ABS: Efficiently Serving Deep Learning Models with Offline CVMs

PACT 2026 Submission **#103** – Confidential Draft – Do NOT Distribute!!

Abstract

Offline Cloud Virtual Machines (CVMs) are a type of CPU compute resources that are easily accessible and have zero operational costs. The two advantages make them attractive for offline deep learning model serving scenarios, where the key objective is to achieve high serving throughputs. However, we observe that the use of offline CVMs in these scenarios poses three key challenges: (1) It is hard to efficiently leverage the available offline CPUs across NUMAs due to the diverse runtime conditions imposed by offline CVMs' parent servers. (2) Offline CVMs lack support for low-precision data formats. (3) The high parallelism required to drive large-scale service deployments places heavy memory burdens on the clients.

To address the above challenges, we build *ABS*, a new system that efficiently serves deep learning models with offline CVMs. *ABS* comprises three key optimizations: (1) It automatically forms balanced pipeline-parallel workers that can exploit CPU resources across multiple NUMA nodes by carefully analyzing the parent servers' runtime conditions. (2) It has an automatic pseudo mixed precision runtime that can transparently and efficiently handle low-precision arithmetic. (3) It virtualizes request payloads on the service clients to significantly reduce their memory footprint. While the first two optimizations improve the runtime and memory efficiency of deep learning models running on offline CVMs, the third one allows us to drive large-scale deployments with fewer client resources.

We evaluate *ABS* in three offline serving scenarios using state-of-the-art deep learning models and a real federated cluster for experiments. Our evaluation shows that *ABS* outperforms baselines that do not possess its key optimizations by 1.66–3.06× in serving throughputs on the three workloads. The serving system is currently deployed in our compute clusters at a scale of 1M CPU cores, and is capable of saving thousands of GPU hours per day.

1 Introduction

Deep learning models form an important class of machine learning algorithms and have made significant advancements in numerous domains such as instruction-tuned language modeling [17, 101, 102] and multi-modal learning (i.e., models that handle a mixture of input types like images and texts and can be used in applications including image classification and captioning [7, 8, 45, 86]). Because of the high demand of using these deep learning models to operate on huge amounts of data, hardware accelerators such as GPUs frequently face shortages [5, 14, 56, 62]. This hence motivates the need to look for alternatives such as offline CVMs that can serve as possible substitutes for these high-performance compute resources.

Offline CVMs are CPU VMs that share the same bare-metal parent servers with multiple online CVMs, and they offer two important benefits: (1) Offline CVMs are widely available and easily accessible: According to our estimates, there are more than 3M offline CPU cores in our compute clusters. Many of these CPU cores are underutilized at late night and early morning, the period which we call *ebb tide*. (2) Offline CVMs are free to use for in-house applications, as the parent servers need to stay operational thanks to the online CVMs. These two important properties make offline CVMs suitable for offline workloads, whose key objective is to achieve high throughputs, and these workloads are also capable of tolerating high latencies, which aligns well with the relatively low compute capabilities of offline CVMs.

While offline CVMs provide the benefits listed above, they encounter significant limitations in efficiency and even practicality when deploying deep learning models. This is because the use of offline CVMs in this scenario poses three key challenges:

(1) *Hard to efficiently leverage the available offline CPUs.* Multi-threading [3, 26, 65, 84] is commonly used to obtain high performance on CPUs. However, we observe that naïvely increasing the number of threads decreases, rather than increases, the serving performance. The reason is two-fold: (1) Modern CPUs typically adopt a multi-NUMA architecture [35], so increasing the number of threads beyond the NUMA boundaries causes CPUs to perform remote memory accesses. (2) Increasing the number of threads used also equally raises the chance of the offline CVM compute jobs' being preempted by those of online CVMs, leading to significant slowdowns (up to 8.35×) in runtime performance.

(2) *Offline CVMs lack support for low-precision data formats.* Low-precision data formats (e.g., bfloat16 [88]) have been the default data type used in many modern deep learning models [6, 17, 101, 102], but offline CVMs do not have the support for hardware intrinsics that allow them to efficiently handle low-precision arithmetic.

(3) *High parallelism and large batch sizes lead to high memory footprint on the service clients.* To maximize the serving throughputs and fully utilize the compute power of the large offline CVM resource pool, we deploy services consisting of a large number of server pods. Each pod is connected to at least one client thread and operates with a large batch size. The high degree of parallelism, along with the large batch sizes, places a significant memory burden on the service clients, especially in the case of serving multi-modal models with service disaggregation, where we separate their submodules into standalone services and have the clients transfer intermediate tensors between these services. The transferring process of service disaggregation incurs more than 200 GB memory footprint on the clients at a medium deployment scale of 400 pods, but it as an optimization is essential for efficient serving, as the disaggregation allows for concurrent execution of multi-modal submodules and averages out ill-performing server pods (which are common in the offline CVM context).

These three challenges hurt the efficiency and the practical use of offline CVMs in many deep learning serving scenarios. To adequately address them, we propose *ABS*, a new holistic system that enables practical and efficient deep learning model serving with offline CVMs. *ABS* is made up of three key optimization techniques:

(1) *ABS* ***automatically forms balanced pipeline-parallel workers*** across multiple NUMA nodes to efficiently leverage the available offline CPUs. It starts with the observation that each NUMA node should be treated as a standalone worker to benefit from its memory locality. However, the fact that every NUMA node of every offline CVM has a unique compute capability (related to its parent server load) makes it impractical to come up with a static parallelization plan that can efficiently orchestrate multiple workers within an offline CVM. We, however, notice that pipeline parallel is a method uniquely suited in this scenario. While it suffers from pipeline stage imbalance when being applied naïvely, the structural consistency of modern deep learning models [34] offers the flexibility of freely adjusting pipeline stages. Based on this observation, for each offline CVM server pod, we carefully analyze the runtime conditions of its parent server and use this analysis to dynamically configure the pipeline stages, so that the compute workloads are balanced between its workers. The balanced workers allow us to efficiently leverage offline CPU resources across multiple NUMAs, increasing the overall service throughputs.

(2) *ABS* incorporates an ***automatic pseudo mixed precision*** runtime that empowers offline CVMs to efficiently handle low-precision data values that are ubiquitous in modern deep learning models. The optimization comes from the insight that even on CPUs that do not have support for low-precision hardware intrinsics (hence the name "*pseudo*"), it is more efficient to perform low-precision in-memory stores and full-precision in-register computes, as the performance benefits brought by fewer memory transfers outweigh the overheads of in-register data type conversions. This idea is generically applicable to common operators that are widely used, such as matrix multiples and paged attention [34], which is why we devise a Python runtime context that allows us to automatically switch to low-level operator implementations with the idea applied for efficiency in both memory and runtime.

(3) *ABS* reduces the memory footprint of the service clients by ***virtualizing request payloads***. We observe that service request payloads are the main reason for the service clients' high memory footprint. These payloads do not have to be present once the requests are issued, but they also cannot be dropped due to the nontrivial failure rates of offline CVM services (around 10–20%), as they have to be reissued when failures happen. Based on this observation, and the fact that more than 99% of the service failures are either immediate connection failures or timeout failures, we devise a virtualization-rematerialization scheme that has service requests release their payloads' memory once they are sent out. In the case when service failures happen, there are two ways for a service request payload to be rematerialized: One is by the ephemeral saved files on disk (for immediate server-client connection failures), and the other is by the service responses (for timeout failures). The scheme significantly reduces the memory consumption of the service clients while making sure that most of the requests can be recovered even if they fail.

Our contributions can be summarized as follows:

- We identify key challenges of efficiently serving modern deep learning models with offline CVMs.
- We build *ABS*, a holistic system that covers both servers and clients. *ABS* adequately addresses those challenges with three key optimizations: (1) It carefully analyzes the runtime conditions of each offline CVM to form balanced pipeline stages. (2) It uses pseudo mixed precision to achieve double wins in memory and performance. (3) It virtualizes payloads of ongoing service requests to reduce the memory consumption of the service clients.
- We evaluate *ABS* on three applications from our data center: image classification, image captioning, and table understanding [41]. These applications operate on proprietary datasets with state-of-the-art deep learning models (Qwen-VL [7] and Llama [17]). We use a real federated cluster and allocate 100K CPU time for experiments. Our evaluation shows that *ABS* is able to outperform baselines that do not have its core optimizations by 1.66–3.06× in service throughputs across different applications. *ABS* is currently deployed in our compute clusters at a scale of 1M CPU cores at ebb tide, and is able to save thousands of GPU hours per day.

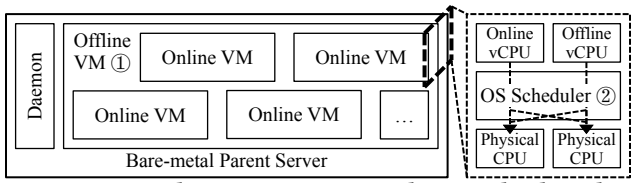


Figure 1: A single parent server can host multiple online and offline VMs. Each VM instance sees only vCPUs, whose processes are managed by the OS kernel scheduler to run on physical CPUs.

2 Background and Motivation

In this section, we present an overview of offline CVMs (Section 2.1) and the challenges of using them to serve state-of-the-art deep learning models (and in particular, multi-modal models [7]) efficiently (Section 2.2). After we present those challenges, we will also demonstrate why prior works [26, 34, 39] and some obvious solutions are inadequate in addressing them (Section 2.3). This hence motivates us to design a holistic system dedicated to efficiently serving deep learning models with offline CVMs.

2.1 Offline CVMs

State-of-the-art deep learning frameworks [1, 10, 59] and serving systems [34, 108] typically use hardware accelerators such as GPUs as their major compute platforms thanks to their powerful compute capabilities. Due to the high demand to execute deep learning workloads, GPU resources offered by cloud vendors are frequently sought after [19], resulting in scarcity and limited cloud user quotas [5, 14, 56, 62]. The shortage of GPU resources hence motivates the need to seek alternative compute resources such as offline CVMs for offline workloads, which are compute jobs characterized by targeting high throughputs and being able to tolerate high latencies.

As is illustrated in Figure 1, offline CVMs are VMs that collocate with online CVMs on the same bare-metal parent server. Specifically, after the physical CPU cores and memory of a parent server have been partitioned to form several mutually exclusive online CVMs and daemon processes, an offline CVM is created (① in Figure 1), sharing the same physical cores with the online ones. Each CVM (whether it is online or offline) sees only virtualized CPUs (vCPUs) and the OS kernel scheduler (②) manages the execution of vCPU processes to have them run on physical CPUs.

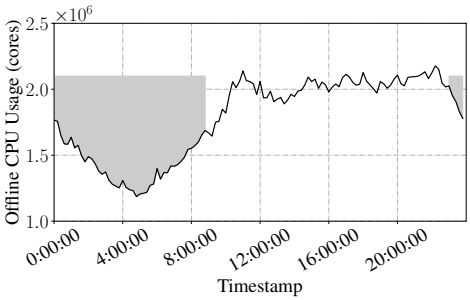


Figure 2: Offline CPU usage of one of our compute clusters, sampled on May 1, 2024. Highlighted in gray is the ebb tide.

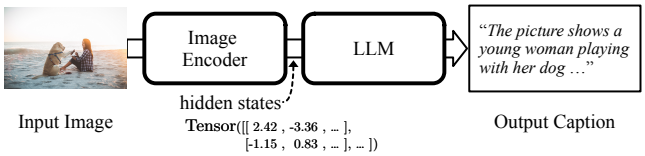


Figure 3: Image caption generation pipeline of Qwen-VL

We highlight that there are two key strengths associated with using offline CVMs: (1) They are easily accessible: We estimate that there are an order of 50K offline CVMs available in our compute clusters, totaling more than 3M available CPU cores. Many of these cores are available at midnight from 11:30 p.m. to 8:30 a.m. when the clusters' workload is light, the period which we refer to as *ebb tide* (see Figure 2). (2) Offline CVMs have zero cost from the viewpoint of Operations and Maintenance for in-house applications, as regardless of whether offline CVMs are used or not, their parent servers have to remain operational due to the presence of online CVMs. Furthermore, the OS scheduler (② in Figure 1) guarantees that offline CVMs have a maximum 1% CPU utilization impact on the online CVMs within the same parent server (even if the offline CVMs aggressively exploit all their CPUs).

2.2 Challenges of Using Offline CVMs

Despite their advantages on accessibility and cost, offline CVMs face numerous challenges in terms of efficiency and even practicability when being used to serve deep learning models. Below we demonstrate the challenges by using Qwen-VL [7] running on offline CVMs as an example. Qwen-VL [7] is a state-of-the-art multi-modal model used in image classification and captioning. Its model architecture is depicted in Figure 3 and consists of two main components: An image encoder that transforms input images as hidden states and a large language model (LLM) that takes the hidden states to generate their corresponding short labels or long captions. To simplify the illustrative example and help better understand the performance impact brought by offline CVMs, we extract from the model a single matrix multiply operator $Y = XW^T$ and use its runtime performance on an offline CVM `testSample` as a microbenchmark[1] (where shapeOf $(X) = (32, 11008)$, shapeOf $(W) = (4096, 11008)$, and `testSample` is a machine equipped with a 192-core Intel Xeon Platinum 8361HC CPU [95]). This is also to show that the challenges listed in the following text apply generically to other types of models, as matrix multiply operators are one of the basic building blocks of deep learning models [46, 47, 68].

[1]See Section 5.1 for the detailed methodology.

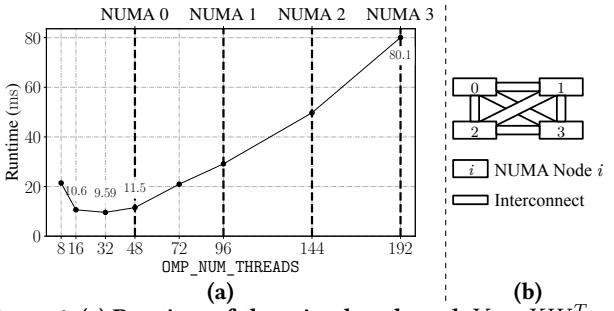


Figure 4: (a) Runtime of the microbenchmark $Y = XW^T$ on `testSample` with various values of `OMP_NUM_THREADS`. Drawn in bold dashed line are the boundaries between NUMA nodes, whose architecture is shown in (b).

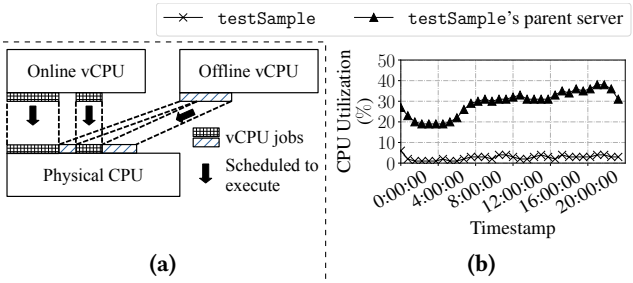


Figure 5: (a) Compute jobs from online VMs have higher weights than those from offline VMs when scheduled to run on physical CPUs. (b) CPU utilization from the perspective of `testSample` versus that from the perspective of its parent server (both sampled on March 12, 2025).

Challenge #1: Hard to efficiently leverage the available offline CPUs. Multi-threading [3, 26, 65, 84] is a commonly used technique to derive high hardware utilization on CPUs, which potentially results in better runtime performance. However, we notice that naïvely increasing the CPU utilization by using more threads can lead to lower runtime performance on offline CVMs. In Figure 4(a) we illustrate how the runtime of the microbenchmark $Y = XW^T$ changes with various values of `OMP_NUM_THREADS` [54], an environment variable that controls how many threads to use for computing programs' parallel regions. We observe from the figure that while the runtime performance improves with the number of threads when its value is below 32, the performance worsens as the value goes beyond this point. The reason for the performance degradation is two-fold:

(1) `testSample` has a 4-NUMA architecture (with 48 cores per NUMA node, see Figure 4(b)), and therefore increasing the number of threads beyond 48 (i.e., a single NUMA) causes some threads to perform remote memory accesses in other NUMA nodes, leading to suboptimal performance [22].

(2) As is introduced in Section 2.1, offline and online CVMs see only virtualized CPUs that are managed by the OS kernel scheduler. To prevent the offline CVMs from undermining the service quality of online CVMs, the scheduler assigns much higher weights to CPU jobs launched from online CVMs than those from offline CVMs (see Figure 5(a)). As a result, even if the CPUs are idle from the

perspective of `testSample`, their utilization can already be above 30% from the perspective of its parent server, as is shown in Figure 5(b). Hence, naïvely increasing the number of threads leads to more contention between offline and online CPUs to fight for the same physical CPUs, raising the chance of the offline compute jobs' being preempted and hurting their runtime performance by up to 8.35×, as Figure 5(a)) shows.

Challenge #2: Offline CVMs lack support for low-precision data formats. With the advent of hardware instruction support, low-precision data formats (e.g., bfloat16 [88]) have been commonly used in training and serving state-of-the-art deep learning models on modern hardware accelerators [6, 7]. For example, CPUs, starting from the Cooper Lake [94] and the Zen-4 [93] micro-architecture, have introduced the AVX512-BF16 instruction [90] for handling bfloat16 [88]-related arithmetic. However, offline CVMs do not have this support. The reason is that: On the one hand, many machines in the cloud data centers were built before the release of those micro-architectures (1949 out of the total 2240 machines in a single cluster that we sample) and hence there is a huge financial cost if they are to be replaced (which we estimate to be 20M$ for the sampled cluster [4]). On the other hand, even if some bare-metal parent servers do have the support for low-precision hardware intrinsics, these features are turned off deliberately by the cloud vendors for offline CVMs, as AVX-512-series instructions (including AVX512-BF16 [90]) have been shown to affect negatively the CPU frequency of the parent servers [9, 16, 38]. Consequently, offline CVMs are incapable of directly supporting commonly used low-precision data formats such as bfloat16 [88] and naïvely using it causes the performance to degrade significantly. Specifically, we evaluate the performance of the microbenchmark $Y = XW^T$ on `testSample` using bfloat16 [88] (with PyTorch ver.2.4.0 [59]+Intel MKL ver.2022.2 [25]) and notice 156× performance slowdown compared with float32. Such performance penalty makes the direct use of bfloat16 [88] impractical for offline CVMs.

Challenge #3: High parallelisms and large batch sizes lead to high memory footprint on the service clients. Not only does the use of offline CVMs pose challenges on the serving workers to efficiently execute deep learning models, but it also presents difficulties for the service clients that issue requests to those workers. To achieve high throughput and unleash the compute capability of the large offline CVM resource pool, we deploy services that constitute many server pods (between 400 and 2000), distributed across multiple offline CVMs. Every server pod corresponds to a replica and hence has at least one client thread connected to it for sending and receiving service requests. Each pod also uses a large batch size, whose value is determined by the point at which the serving model inference throughputs no longer increase.

The high parallelisms combined with the large batch sizes, however, places heavy burden on the clients in terms of memory footprint, especially in the case of serving a multi-modal model such as Qwen-VL [7], where we disaggregate the two image caption generation stages (namely, the image encoder and the LLM) into standalone services and have the clients serve as a transfer station between the image encoder and the LLM service. We illustrate the service disaggregation in Figure 6 and present why it is necessary for efficient serving in Section 2.3. In Figure 7, we show the client

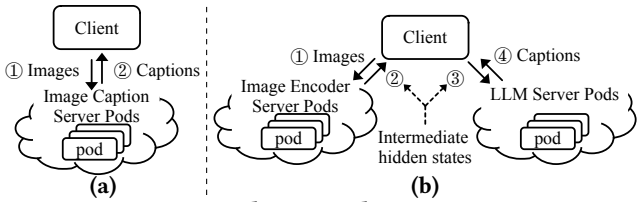


Figure 6: Comparison between the image caption service that deploys the whole model into a single pod (a) and its disaggregated version that separates the image encoder and the LLM module (b). The circled numbers ①–④ denote the service request send and receive order.

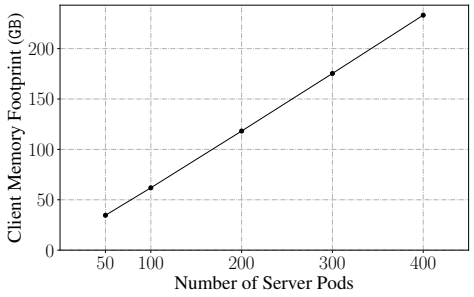


Figure 7: Client memory footprint with respect to the number of server pods (of both the image encoder and the LLM).

memory footprint with respect to the number of server pods (of both the image encoder and the LLM) and observe that the client memory footprint rockets up to 200 `GB` as we deploy only 400 pods. As we cannot use offline CVMs as the clients (since offline CVMs have a limited disk write bandwidth of 5 `MB`/s, which is insufficient for writing back service outputs), it is economical that we use fewer client hardware resources for each deployment, but such a high memory footprint prevents us from further scaling up the services, constraining the throughput that we can derive from the deployable offline CVM nodes.

2.3 Why A New Serving System?

Now that we have presented the key challenges on the practical and efficient aspects of serving deep learning models with offline CVMs, a natural question to ask is why a new serving system is needed to address them. Below, we answer this question by showing why prior works [26, 34, 39] and some obvious solutions fail to resolve these challenges:

Use data-parallel or model-parallel to distribute the services across NUMA nodes. State-of-the-art serving systems [26, 34] suggest replicating the whole service across NUMA nodes and having each replica handle a portion of the incoming service requests (i.e., in a data-parallel manner). Such an approach, however, is not practical for large deep learning models such as Qwen-VL [7], which we measure to use 30 `GB` of memory for its model parameters and workspaces (assuming that the data type used is bfloat16 [88]). This is because duplicating such a large model causes the service memory consumption to at least double, and the resource usage of an offline CVM service is calculated based on **both of its CPU and memory usage**. To alleviate this problem, prior works [39] propose to use model-parallel [77] to split the model across different NUMA nodes. We demonstrate the use of model-parallel [77]

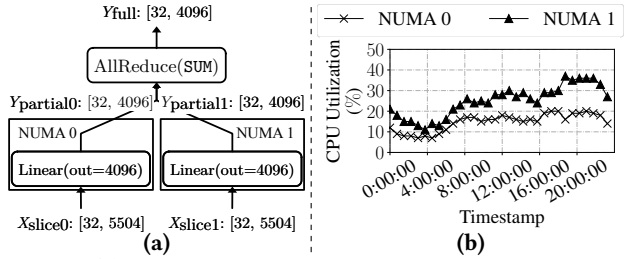


Figure 8: (a) Model-parallel distributes the microbenchmark $Y = XW^T$ into two NUMAs, but under the hood, they have distinct parent server CPU utilization ((b), both sampled on March 12, 2025).

on the microbenchmark $Y = XW^T$ in Figure 8(a) and evaluate its performance on `testSample`. We notice, however, that the performance of model-parallel [77] is 2.09× worse than the baseline that computes within a single NUMA node, and there are two main reasons: (1) As is illustrated in Figure 8(a), model-parallel [77] requires an AllReduce (`SUM`) operator to aggregate the partial compute results, which needs inter-process communication and incurs runtime overheads. (2) Also because of the AllReduce operator, model-parallel [77] expects that the two parallel workers to have similar runtime performance. Such an expectation, however, is hard to meet in the offline CVM context, as different NUMA nodes have distinct parent server CPU utilizations under the hood. We compare the parent server CPU utilization of two CPUs (belonging to different NUMAs) in Figure 8(b) and observe that their CPU utilization could be different by as much as 18%, hindering the efficiency of the model-parallel approach [77].

Convert tensor data types to float32 ahead-of-time or just-in-time. Two workarounds to handle low-precision data formats of modern deep learning models include converting their parameters to float32 by the time when the services are initialized (i.e., ahead-of-time) or at runtime when the parameters are used (i.e., just-in-time). Although both approaches enable the use of efficient hardware intrinsics (e.g., AVX2 [91]), the former approach doubles the amount of service memory consumption, and the latter introduces runtime overheads that are caused by data type conversions. We evaluate the performance of just-in-time conversions with the microbenchmark $Y = XW^T$ on `testSample` and notice that the performance drops by 1.88× compared with the implementation that directly uses float32, indicating the inefficiency of this approach.

Deploy multi-modal models as a whole, or have the disaggregated services directly connect to each other in a bundled manner. Despite the fact that deploying multi-modal models as a whole mitigates the high memory footprint problem on the service clients, it is an inefficient deployment strategy since only one of the multi-modal submodules can be active at a time on each server pod. Although it is feasible to disaggregate the services and have them directly connect to each other in a bundled manner without passing through the clients (which we refer to as *bundled disaggregation* to distinguish between our approach), such an approach has a key weakness that an ill-performing server pod can cascade through its entire bundle. For example, if an image encoder server pod in Qwen-VL [7] is not performant (due to high parent server CPU utilization), its downstream LLM pod will be starved even if the pod

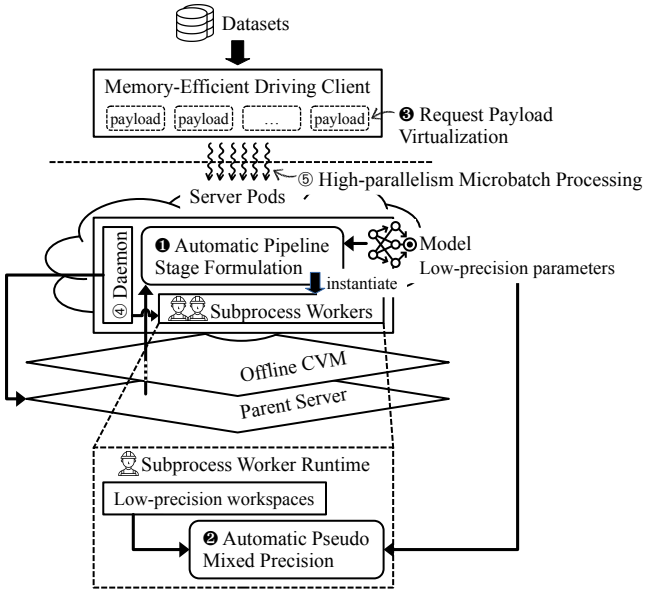


Figure 9: *ABS* System Overview

is idle; and if an LLM pod is not efficient, it will also block its upstream image encoder pod from processing new image requests. In contrast, service disaggregation does not have this problem, as the performance variance of each individual server pod becomes negligible after being averaged across many pods. We compare between the service and the bundled disaggregation deployments in Section 5 and observe that the former has 1.48–1.96× higher service throughputs than the latter, proving that service disaggregation is a better deployment strategy.

3 ABS: Key Ideas

To adequately address the challenges described in Section 2.2, we propose *ABS*, a new system for efficiently serving deep learning models with offline CVMs. We illustrate the system diagram of *ABS* in Figure 9 and highlight its key optimization techniques (❶–❸) in this section (while presenting ④–⑤ in Section 4). Specifically, *ABS* resolves the aforementioned Challenge #1–3 using three key ideas: ❶ It *automatically forms pipeline stages* to efficiently leverage the available offline CPUs across multiple NUMA nodes. ❷ It has an *automatic pseudo mixed precision* runtime context for transparently and efficiently supporting low-precision arithmetic. ❸ On the client side, it *virtualizes request payloads* to significantly reduce the client's memory footprint. We now elaborate on these ideas.

3.1 Automatic Pipeline Stage Formulation

Our first key idea, *automatic pipeline stage formulation*, is based on the observation that we can derive high CPU utilization plus runtime efficiency on a multi-NUMA offline CVM by analyzing its parent server's runtime conditions and forming balanced pipeline-parallel workers across its NUMA nodes. We notice from the runtime performance trend in Figure 4 that while it is evident that we should treat each NUMA node as a standalone worker to make use of its memory locality, the difficulty lies in how we could effectively orchestrate multiple workers within the same offline CVM by parallelizing their executions, especially given that they have different

compute capabilities (due to distinct parent server utilization, exemplified in Figure 8(b)). To make this even more challenging, every parent server in our compute clusters exhibits unique runtime conditions. These intra- and inter-VM diversities make it hard to come up with a static parallelization plan that can efficiently coordinate NUMA node workers across all offline CVMs.

To address this challenge, we consider the commonly used parallelization techniques (including data, model [77], and pipeline parallel) and notice that while the former two suffer from major weaknesses (as is explained in Section 2.3), pipeline parallel has the potential of being well fitted into the diverse runtime environments of offline CVMs. Specifically, we observe that while pipeline parallel also struggles with inefficiencies when being applied naïvely due to imbalanced pipeline stages, it is highly configurable thanks to the homogeneity of modern deep learning models [34] (meaning that they are composed of structurally identical layers with different data values). This provides us with the flexibility of freely adjusting the pipeline partition boundaries. We benefit from this flexibility and carefully analyze the runtime conditions of the underlying parent servers with performance profiling that penetrates through offline CVMs (illustrated in Figure 9 as an '$\rightarrow$' from "*Parent Server*" to ❶). We then use this analysis information to form balanced pipeline stages, each of which corresponds to a subprocess worker. These workers allow us to efficiently leverage offline CPU resources across multiple NUMAs, leading to high service throughputs. We evaluate the performance impact of our automatic pipeline stage formulation in Section 5.2 and obtain a 1.21–1.51$\times$ increase in throughputs, whereas the naïve pipeline parallel implementation degrades the performance by 1.07–1.17$\times$.

3.2 Automatic Pseudo Mixed Precision

Our second key idea, *automatic pseudo mixed precision*, comes from the insight that on CPUs that do not have hardware intrinsic support for low-precision data formats, it is more efficient to do low-precision in-memory stores and full-precision in-register computes. This is because the runtime benefits brought by fewer memory reads and writes outweigh the runtime overheads introduced by data format conversions at the register level. To demonstrate this, we consider again the microbenchmark $Y = XW^T$ in Section 2.2 running on the sample offline CVM `testSample`. We use OpenBLAS [105], a state-of-the-art open-source linear algebra libraries, to carry out the computation and study its runtime performance when using float32 on a single thread. We categorize the runtime into three portions: (1) `COPYX`: the time it takes to copy X from memory to registers, (2) `COPYW`: the time to copy W, and (3) `KERNEL`: the time to compute Y using low-level hardware intrinsics (e.g., AVX2 [91]). We plot the runtime breakdown in Figure 10(a) and observe from the figure that `COPYW` constitutes a significant part of the overall runtime (47%). Therefore, upon switching the data type of W to bfloat16 [88] while preserving others in float32, the time of `COPYW` can be reduced by 2.10$\times$ thanks to fewer memory transfers (as Figure 10(b) shows), contributing to an increase of 1.36$\times$ in performance. We find that this idea of pseudo mixed precision is generically applicable to common deep learning operators, which is why we design a Python runtime context to support automatic and

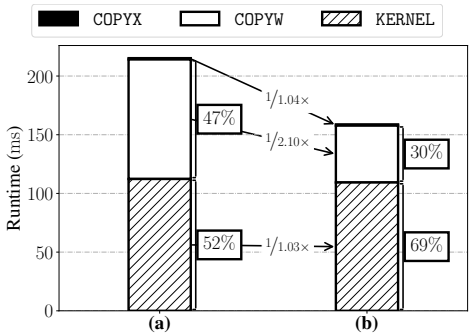


Figure 10: Runtime breakdown of the microbenchmark $Y = XW^T$ on `testSample`, compared between (a) all float32 and (b) W in bfloat16 while others still in float32.

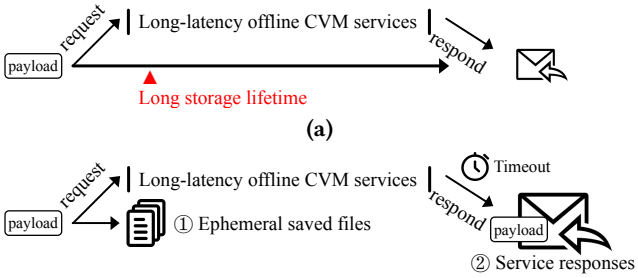


Figure 11: (a) The long latencies of offline CVM services cause request payloads to have long storage lifetime, but (b) they can be virtualized and later rematerialized by local files or service responses.

transparent transition to optimized low-level operator implementations with the idea applied for efficiently handling low-precision model parameters and workspaces that are ubiquitous in modern deep learning models. Our evaluation in Section 5.2 shows that our optimization of pseudo mixed precision significantly reduces the memory consumption by 1.79–1.91$\times$ for server pods running on offline CVMs, which can be converted to an increase of 1.13–1.38$\times$ in service throughputs.

3.3 Request Payload Virtualization

Our third key idea, *request payload virtualization*, originates from the analyses that service request payloads are the main reason that attributes to the high memory footprint of the service clients. We find that unlike services that use high-performance hardware accelerators where requests can be completed within seconds, offline CVM services require much higher latencies to handle incoming requests (from tens to even hundreds of seconds). Consequently, the payloads associated with these requests have to stay in memory for a long period of time (as Figure 11(a) shows), resulting in humongous memory consumption when the service deployment scale (and hence also the number of concurrent requests) goes up.

We observe that request payloads do not have to be kept alive in memory once the requests have been issued. Nevertheless, directly releasing them leads to inefficiencies in offline CVM services that apply service disaggregation. This is because offline CVM services have nontrivial failure rates (around 10–20% by our measurements), and if an intermediate request (e.g., ③ in Figure 6(b))

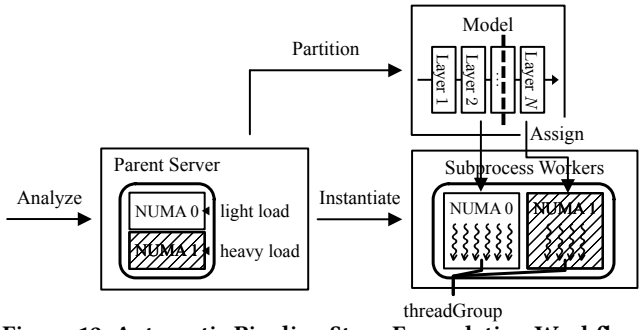


Figure 12: Automatic Pipeline Stage Formulation Workflow

fails and is released, it will have to be reproduced by the disaggregated service that initially generates it (i.e., the image encoder), causing unnecessary recomputations. To handle the memory footprint challenge while avoiding these recomputations, we devise a request payload virtualization and rematerialization scheme as in Figure 11(b). OOur rematerialization approaches are based on the observation that service request failures fall into the following three major categories: (1) The service-client connection has been rejected (due to the server pods' being evicted by K8s [31]). (2) The request processing time exceeds the timeout limit (which we configure to be 15 min). (3) The server pods are suddenly terminated (e.g., due to the instability of Tritonserver [20, 74]). From this observation, we rematerialize failed service requests in (1) by reloading them from ephemeral saved files on disk (① in Figure 11(b)), and those in (2) by having server pods transfer the payloads back in the form of service responses (②). We evaluate the effect of our virtualization scheme and obtain a 39.19× reduction in memory footprint compared with the baseline that does not implement virtualization at the same deployment scale. Furthermore, 99.04% of the failed requests can be rematerialized, allowing us to practically benefit from the performance benefits brought by service disaggregation.

4 Implementation Details

As is depicted in Figure 9, *ABS* is a holistic serving system whose key optimization techniques span both the server and the client side. In this section, we dive deep into the implementation details of both sides, with special focus on modules with circled numbers in Figure 9. After that, we also present details on the deployment strategy for fast service setup at ebb tide.

4.1 Automatic Pipeline Stage Formulation

In Figure 12, we illustrate the workflow of our automatic pipeline stage formulation process. The workflow is invoked every time when we launch a deep learning model service on a server pod. It starts by carefully analyzing the runtime environments of the parent server that lie beneath the offline CVM, the details of which are listed in in Algorithm 1. Specifically, we start by filelocking an offline CVM-wide volume `/abs_perf` to prevent simultaneous profiling from other server pods (line 1); we then apply a stress test workload on all the useable CPU cores to evaluate their CPU utilization (line 6) [18]; from the CPU utilization, we select the top-P cores (default to 10%) within each NUMA node to form a thread group (line 9–10), each corresponding to a subprocess worker of *omp* threads; we measure the compute capability of

Algorithm 1: Parent Server Performance Profiling

Input : number of pipeline stages to form *numStages*
Parameters: top CPU cores filter threshold *topP*; minimum OMP value *minOMP*
Inout : CVM-wide volume `/abs_perf`
Output : a mapping {*numaNode*: (*omp*, *flops*)}

1 filelock `/abs_perf`;
2 *useableCPUs* ← process.schedAffinity \ `/abs_perf/subscribed_cpus`;
3 *cpuNodes* ← {numaNode : {$c \mid \forall c \in$ *useableCPUs* and $c \in$ numaNode}};
4 launch synthetic workload *stressTest* on all *useableCPUs*;
5 **for** (*thread*, *coreId*) ∈ zip(*stressTest.threads*, *useableCPUs*) **do**
6 *cpuCoreUtils*[*coreId*] ← *stressTest*.cpuPercentt × (*thread*.systemTime + *thread*.userTime) × $1/\sum$(*stressTest*.cpuTimes);
7 **end**
8 **for** *node* ∈ *cpuNodes* **do**
9 *topCPUCores*[*node*] ← {$c \mid \forall c \in$ *node*.cores and *cpuCoreUtils*[c] $\geq (1 - topP) \times \max(cpuCoreUtils[\{c \mid \forall c \in node.\text{cores}\}])$};
10 *omp* ← max(len(*topCPUCores*[*node*]), *minOMP*);
11 launch FLOPS measuring workload *blasTest* with NUMA bound to *node* and `OMP_NUM_THREADS`= *omp*;
12 *perfResults*[*node*] = (*omp*, *testBLAS*.flops);
13 **end**
14 **for** *node* ∈ (sorted$_{flops}$(*topCPUCores*)) [:*numStages*] **do**
15 mark as *perfResults*[*node*].omp CPU cores as subscribed in `/abs_perf/subscribed_cpus`;
16 **end**
17 release the filelock on `/abs_perf`;
18 **return** *perfResults*;

every worker in FLOPS with a BLAS test (line 11); finally, we mark CPUs in the thread groups as subscribed and release the filelock (line 14–17). Given the *flops*es that are obtained in Algorithm 1 and the model that is to be served, we assign model layers to each subprocess worker based on the formula below (expressed in the form of intervals) to instantiate the worker:

$$\left[N_0 \frac{\sum_{i=0}^{rank} flops}{\sum flops}, \min\left(N_0 \frac{\sum_{i=0}^{rank+1} flops}{\sum flops}, numLayers \right) \right)$$

where *numLayers* is the number of hidden layers in the model, *rank* is the subprocess rank, and N_0 is set to *numLayers* + 2 if the model has a large projection layer (which is commonly used in LLM services to project their final hidden states to a large vocabulary size) or *numLayers* otherwise. The objective of the layer assignment is to ensure that all workers have similar latencies upon receiving the same service request.

After the subprocess workers have been instantiated, a daemon process will be launched to constantly health-check the workers' status (④ in Figure 9). Additionally, it will also monitor the CPU utilization of the parent server in a 1-hour period to determine if

pipeline stage re-formulation is required, which will happen if the utilization has changed by 10% since its previous checkpoint.

4.2 Automatic Pseudo Mixed Precision

We apply the idea of pseudo mixed precision by focusing on operators that matter the most in the server pods' memory consumption, as efficiently supporting these operators allows us to derive most of the memory and the performance benefits. We notice that model parameters and KVCache [21] consume more than 90% of the total memory in the deep learning models that we serve [7, 17]. While the former is the input to the matrix multiply operator, the latter is a data structure that is used in LLM serving and serves as inout to the paged attention operator [34]. From these observations, we implement new operators on top of the `gemm` kernels of OpenBLAS [105] and the paged attention operator of vLLM [34]. The new operators perform their core operations in float32, and every time when they read or write a low-precision memory region, we leverage the AVX2 hardware intrinsic [91] to make the conversion efficient. To be more specific on the `gemm` operators, we write new `INCOPY/ITCOPY/ONCOPY/OTCOPY` kernels tailored to each microarchitecture in our compute clusters (namely, Haswell [91] and Zen [92][2]). The kernels correspond to reading the left (`I`) or the right (`O`) hand side operand in normal (`N`) or transposed (`T`) layout. We integrate our new operators as part of the PyTorch machine learning framework [59] and the vLLM serving system [34]. To allow for minimal changes at the model source code level, we design a Python runtime context that facilitates the switch in the low-level operator implementations to enable transparent handling of low-precision model parameters and KVCache storage [21].

4.3 Memory Efficient Driving Client

Request Payload Virtualization. We implement the request payload virtualization in Triton client [52] and aiohttp [2]. Whenever a service request is `POST`ed and configured to be virtualized, its payload will be serialized to a temporary file on disk for the first way of rematerialization. We set a short lifetime for the saved files (5 seconds) to prevent them from piling up when the number of concurrent requests goes up, which will not only consume hundreds of `GB` of disk storage, but will also cause CPUs to waste time on disk I/Os that could otherwise be used for sending out service requests or receiving service responses (note that if the files are ephemeral, they will not trigger long-latency disk operations thanks to the memory buffer mechanism [69] that is supported by modern operating systems). To support the second way of rematalization, we configure the behavior of the server pods so that they embed service payloads as part of the service responses when the requests' processing time has reached the preset timeout limit.

In the case when a service failure happens, we start by examining the disk storage to see whether its associated saved file is present or not. If the file is present, we can directly recover the payload to make another request attempt. Otherwise, if the file is absent but the failure is a timeout, we read from the service response to rematerialize the request. If neither of the above applies, which could happen when the server pods are abruptly terminated (e.g., due to their being evicted by K8s [32]), the clients will have to reconstruct the payload according to its unique identifier. In the case of serving multi-modal models (e.g., Qwen-VL [7]) with service disaggregation, a failed and un-rematerializable request to the LLM service will have to be reconstructed from the image encoder service. We observe, however, that since eviction happens rarely, these types of requests constitute less than 1% out of all the requests issued according to our measurements.

[2]Although some machines in our compute clusters are of the Skylake [89] or the Cooper Lake [94] microarchitecture, they are treated as Haswell [91] with their AVX-512 features [89] disabled.

Table 1: List of Offline CVM Specifications

CPU Model	Cores	Memory	NUMAs
Intel Xeon Platium 8255C [13]	84	48/64	2
Intel Xeon Platium 8361HC [95]	192	128	4
Intel Xeon Platium 8374C [95]	132	32/64	2
AMD EPYC 7K62 [57]	180	64	2
AMD EPYC 7K83 [58]	232	64	2

High-parallelism Microbatch Processing (⑤ in Figure 9). To derive high parallelisms on the clients for large-scale service deployments, we combine Python multi-processing [64] with asyncio [63]. For each server pod, we assign one coroutine to it for sending and receiving service requests. In the case of serving an LLM, more coroutines per pod are required to keep a large batch size for high-throughput execution. This is because the execution time of LLM is data-dependent (e.g., in image captioning, different images could generate captions of various lengths, resulting in distinct serving latencies). Therefore, we compose a large batch size using several small microbatches and have each client coroutine responsible for managing a single microbatch, so that new microbatches can replenish the server pods in case some of their microbatches exit early. For example, in the case of Qwen-VL [7] where we set a batch size of 32 for each server pod, we have 8 coroutines per pod, with each processing the service requests of 4 images, leading to 16K coroutines concurrently running on the service clients when the deployment scale is 2000 server pods. These coroutines are distributed evenly among multiple processes, the number of which on a service client is equal to the number of CPU cores that the client has.

5 Evaluation

5.1 Methodology

Infrastructure. Our compute platform is the offline CVMs that are available in our compute clusters. In Table 1 we list the specifications of the top-5 most common machines by sampling 30K machines in our clusters. We use K8s [32] to deploy our services on a federated cluster that is composed of 19 physical clusters and allocate a fixed CPU time budget of 100K for deployments, with 32 CPU time and a maximum of 32 `GB` memory assigned to each server pod, We employ a machine that is equipped with a 28-core Intel Xeon E5-2680 v4 CPU [23] and 256 `GB` memory as the client. To ensure fairness in comparisons, all experiments are done within the 9 a.m. to 9 p.m. interval (i.e., non-ebb-tide period).

For the software infrastructure, we use PyTorch ver.2.4.0 [59] built with Intel MKL ver.2022.2 [25] and OpenBLAS ver.0.3.29 [105] for executing basic deep learning operators, vLLM ver.0.5.4 [34] for serving LLM modules, and Tritonserver ver.22.04 [53] paired with Triton client ver.2.47.0 [52] for setting up server-client connections.

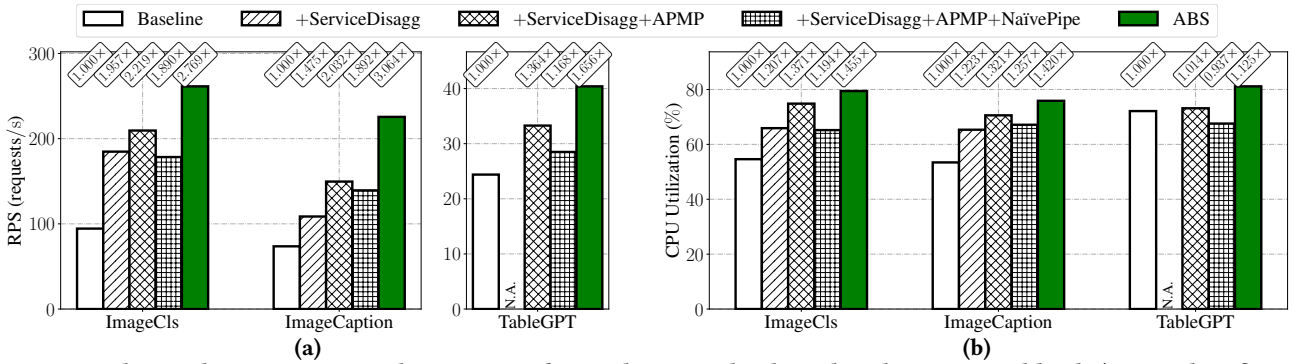


Figure 13: End-to-end RPS comparison between *Baseline* and *ABS* on the three deep learning workloads (image classification (ImageCls), image captioning (ImageCaption), and table understanding (TableGPT)).

Applications. We evaluate our new serving system *ABS* on two state-of-the-art deep learning models: Qwen-VL [7] and Llama-8b-Instruct [17]. While the former is used in the image classification and captioning workloads (with different model parameters), the latter corresponds to the table understanding tasks [41]. We use sample traces in our data center as input data to those services. For image captioning and table understanding [41], which are workloads that involve generating long sequences of various lengths, we set a maximum context length of 32K.

Baselines. We define the implementation that uses the basic OpenMP and NUMA settings as *Baseline*, where by "*basic*" we mean that *Baseline* binds its service to the most performant NUMA and sets a fixed `OMP_NUM_THREADS` value of 32. As we have shown in Figure 4, without these basic settings the runtime performance can be $8.35\times$ worse. Additionally, *Baseline* converts all tensor data types to float32 ahead-of-time and uses bundled disaggregation for serving multi-modal models (where a bundle is formed by grouping one server pod from each multi-modal submodule). To show the performance benefits of each of *ABS*'s key optimizations, we add on top of *Baseline* the ability to perform service disaggregation and automatic pseudo mixed precision, denoted as *ServiceDisagg* and *APMP* respectively, and *ABS* further extends beyond the two by incorporating the ability to automatically form pipeline stages.

Metrics. We show results on (1) the end-to-end service request processing capabilities, measured in requests per second (RPS, higher is better) and obtained by averaging a 10-minute interval after the service being measured is stabilized, (2) the CPU utilization (higher is better), measured in % and sampled from the perspective offline CVMs (hence affected by the steal time of online CVMs), and (3) the server pods' and the client's memory consumption, measured in GB (lower is better). In the motivational examples in Section 2.2, where we measure the runtime performance of $Y = XW^T$ on `testSample`, we run for 128 iterations and report the median.

5.2 End-to-End Serving Performance

Figure 13 shows the end-to-end RPS comparison between *Baseline* and *ABS* on the three deep learning workloads. We observe from the figure that the key optimization techniques of *ABS* bring noticeable speedups across diverse workloads. Specifically:

(1) On multi-modal models where service disaggregation is applicable, deploying multi-modal submodules as standalone services significantly increases the RPS by 1.48–$1.96\times$. As is explained in Section 2.3, the main reason for this performance speedup is that service disaggregation does not have the problem as in bundled disaggregation where an ill-performing server pod starves others that are in its bundle. This can be verified by comparing the server pods' CPU utilization between *Baseline* and *ServiceDisagg* in Figure 13(b), where we notice that average CPU utilization increases by $1.2\times$.

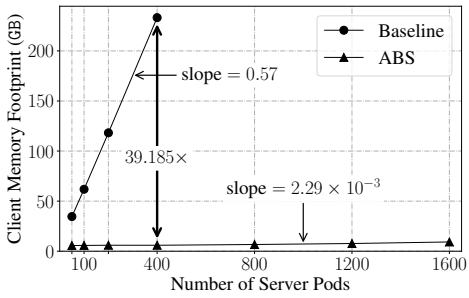


Figure 14: Comparison of the client memory footprint between *Baseline* and *ABS*.

Note that the application of service disaggregation is not practical without implementing request payload virtualization on the service client. Similar to Figure 7, in Figure 14 we compare the client memory footprint between *Baseline* and *ABS* and observe that *ABS* significantly reduces the client's memory footprint by $39.19\times$ when the deployment scale is 400 server pods, allowing us to further scale up the service. We measure the rematerialization details and observe that while 13.07% of the service requests are rematerialized (due to service failures), 11.78% of them are rematerialized from the ephemeral saved files on disk (due to network errors that are caused by K8s pod evictions [31]) and 87.26% of them are rematerialized from the service request responses (due to service timeouts), leaving only a small fraction ($< 1\%$) that cannot be rematerialized (due to the instability of Tritonserver [53], as is reported in several online issues [20, 74]). None of the above causes of failures is specific to our cloud infrastructure.

(2) *APMP* adds on top of *ServiceDisagg* a boost of 1.13–$1.38\times$ in RPS. This is because it drastically reduces the memory consumption of the server pods. We evaluate the memory footprint of the three applications and observe that enabling *APMP* reduces the server pods' memory size by $1.79\times$, $1.83\times$, and $1.91\times$ respectively under the same batch sizes. Such a large memory footprint reduction brings two major benefits: I. The smaller memory sizes make it possible for the K8s scheduler [30] to seek CVM nodes with more offline

Workload	RPS	Ratio to *ABS* (with 100K CPU)
ImageCls	6.57	1/39.78
ImageCaption	3.41	1/66.06
TableGPT	1.10	1/36.72

Table 2: RPSes of the three workloads on an NVIDIA HGX H20 GPU, and are compared with the ones in Figure 13.

CPUs. As we have shown in Figure 13(b), the CPU utilization increases by 1.01–1.14× on the three workloads. II. *APMP* allows the maximum batch size to double under the same workspace memory budget, leading to higher throughputs. For example, LLM services such as TableGPT [41] are initialized by allocating a fixed amount of memory for their KVCache [21], a data structure whose size scales proportionally with their data type and determines the maximum number of sequences that can run concurrently. Therefore, with the efficient support of bfloat16 [88], *APMP* increases the maximum number of running sequences from 16 to 32, contributing to the 1.36× RPS increase.

(3) Finally, *ABS*'s automatic pipeline stage formulation further increases the RPSes by 1.21–1.51× beyond those achieved by the combination of *ServiceDisagg* and *APMP*. This is due to its ability to automatically form thread groups and partition model layers across different NUMA nodes, which results in the efficient use of offline CPUs on the multi-NUMA architecture. As we have shown in Table 1, this optimization is applicable to all the top-5 most common machines in our compute clusters, raising the CPU utilization by 1.04–1.08× compared to the implementations that bind the service to a single NUMA node. As a control experiment, we also show the performance numbers obtained by replacing the automatic pipeline formulation scheme of *ABS* with a naïve scheme that does not balance the pipeline stages (denoted as *NaïvePipe* in Figure 13) and notice a 1.07–1.17× drop in serving performance, indicating that the naïve scheme fails to be efficient in the diverse runtime environments of offline CVMs.

5.3 Comparison with Modern GPUs

To demonstrate *ABS*'s potential as possible substitutes for high-performance hardware accelerators, we compare its performance on the three applications with that of modern GPUs. We evaluate the three workloads on a machine that is equipped with an NVIDIA HGX H20 GPU [83]. The software stack that we are using is PyTorch ver.2.6.0+cu124 [59] and vLLM ver.0.8.2 [34]. In Table 2, we show the RPSes that are obtained by increasing the request batch sizes until the service throughputs saturate, and compare these values with the ones that are illustrated in Figure 13. We observe from the comparison that the processing capability of *ABS* (when deployed at a scale of 100K CPU) is roughly equal to 39.78×, 66.06×, and 36.72× of that of a single H20 GPU [83] on the three workloads (image classification, image captioning, and table understanding [41]) respectively. After examining the timeline traces, we notice that the reason why the image captioning figure stands out is that the combination of the image encoding and the long LLM text generation pipeline is unfavorable for GPUs, as while the former is compute-intensive, the latter is memory-intensive. These two stages are coupled to have the same resource allocation and parallelization plans, causing them to interfere with each other and yielding inefficient serving throughputs on GPUs [78, 109]. Given the above comparisons, we conclude that, as *ABS* is currently deployed in our compute clusters at a scale of 1M CPU at ebb tide every day, the serving system saves thousands of GPU hours per day with offline CVMs, a solid justification of its economical values.

6 Related Works

ABS addresses key challenges in making deep learning model serving practical and efficient with offline CVMs. It is orthogonal to research works that compile deep learning models and generate efficient executables on CPU platforms [11, 36, 40, 55, 75, 82, 107] and aligns with the philosophy of research works that leverage low-cost compute resources to perform complex compute tasks, such as SpotServe [48] and Junkyard Computing [80].

P/D disaggregation is a technique used in LLM serving to split the prefill and the decode phase of an LLM model into separate device workers. Prior works of P/D disaggregation resolve its challenges in terms of runtime efficiency and KVCache management [28, 60, 66, 78, 79, 109]. We notice that while P/D disaggregation and service disaggregation share some similarity, the former offers the flexibility to tune SLA requirements (time-to-first-token and inter-token-latency) separately and the latter is used to average out offline CVM server pods that perform poorly due to high parent server utilization. This is why *ABS* implements disaggregation at the service-level granularity, rather than in a bundled manner.

Low precision data formats (e.g., bf16 [88], fp16 [49], fp8 [15, 37, 50, 73, 98, 104], fp6 [99], fp4 [44, 87], int8 [24, 96, 100, 103, 110], and int4 [42, 43, 76, 97]) have been commonly used in modern deep learning models to accelerate training and/or inference. There are also research works that focus on efficient code generation across a variety of low-precision data types [51, 81, 82, 85]. Most of these works leverage hardware-accelerated low-precision compute capabilities, particularly those inherent in modern GPU architectures. In *ABS*, we efficiently support low-precision data formats on offline CVMs while maintaining full model fidelity, without relying on native hardware support for low-precision arithmetic.

Memory virtualization and rematerialization. As modern deep learning models become large, GPU memory optimizations are usually applied to avoid hitting the memory capacity wall. Virtualization (through offloading to CPUs or NVMe) [67, 70–72] and rematerialization (through activation recomputation) [12, 27, 29, 33, 61, 106] are commonly used techniques especially in training modern deep learning models. These techniques are related to the request payload virtualization of *ABS*, but *ABS* focuses on the memory regions of the service request payloads rather than those of the deep learning models' data values.

7 Conclusion

We build *ABS*, a new system that efficiently serves deep learning models with offline CVMs. On the three state-of-the-art deep learning models evaluated, *ABS* outperforms baselines that do not possess its key optimizations by 1.66–3.06× in serving throughputs. System researchers and machine learning practitioners can all benefit from *ABS*, as it unleashes the compute capability of offline CVMs and saves high-performance compute resources such as GPUs for workloads that have stringent SLA requirements.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, Kimberly Keeton and Timothy Roscoe (Eds.). USENIX Association, 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[2] aio-libs. 2025. Async http client/server framework. https://github.com/aio-libs/aiohttp accessed on April 11, 2025.

[3] AMD. 2020. High Performance Computing (HPC) Tuning Guide for AMD EPYC™ 7002 Series Processors. https://intel.github.io/intel-extension-for-pytorch/cpu/latest/tutorials/performance_tuning/tuning_guide.html accessed on April 11, 2025.

[4] AMD. 2023. AMD EPYC™ 9754. https://www.amd.com/en/products/processors/server/epyc/4th-generation-9004-and-8004-series/amd-epyc-9754.html

[5] avivsham. 2023. GCP A100 Shortage. https://www.googlecloudcommunity.com/gc/Infrastructure-Compute-Storage/GCP-A100-Shortage/m-p/637252

[6] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. 2023. Qwen Technical Report. *CoRR* abs/2309.16609 (2023). https://doi.org/10.48550/ARXIV.2309.16609 arXiv:2309.16609

[7] Jinze Bai, Shuai Bai, Shusheng Yang, Shijie Wang, Sinan Tan, Peng Wang, Junyang Lin, Chang Zhou, and Jingren Zhou. 2023. Qwen-VL: A Frontier Large Vision-Language Model with Versatile Abilities. *CoRR* abs/2308.12966 (2023). https://doi.org/10.48550/ARXIV.2308.12966 arXiv:2308.12966

[8] Shuai Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Sibo Song, Kai Dang, Peng Wang, Shijie Wang, Jun Tang, Humen Zhong, Yuanzhi Zhu, Ming-Hsuan Yang, Zhaohai Li, Jianqiang Wan, Pengfei Wang, Wei Ding, Zheren Fu, Yiheng Xu, Jiabo Ye, Xi Zhang, Tianbao Xie, Zesen Cheng, Hang Zhang, Zhibo Yang, Haiyang Xu, and Junyang Lin. 2025. Qwen2.5-VL Technical Report. *CoRR* abs/2502.13923 (2025). https://doi.org/10.48550/ARXIV.2502.13923 arXiv:2502.13923

[9] BeeOnRope. 2019. SIMD instructions lowering CPU frequency. https://stackoverflow.com/a/56861355/6320608

[10] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. JAX: Composable Transformations of Python+NumPy Programs. http://github.com/jax-ml/jax

[11] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[12] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training Deep Nets with Sublinear Memory Cost. *CoRR* abs/1604.06174 (2016). arXiv:1604.06174 http://arxiv.org/abs/1604.06174

[13] CPU-World. 2025. Intel Xeon 8255C specifications. https://www.cpu-world.com/CPUs/Xeon/Intel-Xeon%208255C.html accessed on April 11, 2025.

[14] crinix. 2024. Launching p5.48xlarge (8xH100). https://www.reddit.com/r/aws/comments/1fb8zt8/launching_p548xlarge_8xh100/?utm_source=share&utm_medium=web3x&utm_name=web3xcss&utm_term=1&utm_content=share_button

[15] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, and Wangding Zeng. 2024. DeepSeek-V3 Technical Report. *CoRR* abs/2412.19437 (2024). https://doi.org/10.48550/ARXIV.2412.19437 arXiv:2412.19437

[16] Travis Downs. 2020. Ice Lake AVX-512 Downclocking. https://travisdowns.github.io/blog/2020/08/19/icl-avx512-freq.html

[17] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurélien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Rozière, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Graeme Nail, Grégoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel M. Kloumann, Ishan Misra, Ivan Evtimov, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, and et al. 2024. The Llama 3 Herd of Models. *CoRR* abs/2407.21783 (2024). https://doi.org/10.48550/ARXIV.2407.21783 arXiv:2407.21783

[18] Gabe and Florent Thiery. 2014. CPU usage per thread. https://stackoverflow.com/questions/26238184/cpu-usage-per-thread

[19] Erin Griffith. 2023. The Desperate Hunt for the A.I. Boom's Most Indispensable Prize.

[20] Kanchan Gupta. 2023. Stub process is unhealthy and it will be restarted. https://github.com/triton-inference-server/server/issues/6480

[21] Hugging Face. 2025. KV cache strategies. https://huggingface.co/docs/transformers/main/en/kv_cache accessed on April 11, 2025.

[22] Intel. 2015. Intel Guide for Developing Multithreaded Application. https://www.intel.com/content/www/us/en/developer/articles/guide/guide-for-developing-multithreaded-applications.html

[23] Intel. 2022. Intel® Xeon® Processor E5-2680 v4. https://www.intel.com/content/www/us/en/products/sku/91754/intel-xeon-processor-e52680-v4-35m-cache-2-40-ghz/specifications.html

[24] Intel. 2023. INT8 Quantization for x86 CPU in PyTorch. https://pytorch.org/blog/int8-quantization/

[25] Intel. 2025. Intel® oneAPI Math Kernel Library (oneMKL). https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html#gs.kvuqu3 accessed on April 11, 2025.

[26] Intel. 2025. Performance Tuning Guide. https://intel.github.io/intel-extension-for-pytorch/cpu/latest/tutorials/performance_tuning/tuning_guide.html accessed on April 11, 2025.

[27] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Kurt Keutzer, Ion Stoica, and Joseph Gonzalez. 2020. Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization. In *Proceedings of the Third Conference on Machine Learning and Systems, MLSys 2020, Austin, TX, USA, March 2-4, 2020*, Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze (Eds.). mlsys.org. https://proceedings.mlsys.org/paper_files/paper/2020/hash/0b816ae8f06f8dd3543dc3d9ef196cab-Abstract.html

[28] Yibo Jin, Tao Wang, Huimin Lin, Mingyang Song, Peiyang Li, Yipeng Ma, Yicheng Shan, Zhengfan Yuan, Cailong Li, Yajing Sun, Tiandeng Wu, Xing Chu, Ruizhi Huan, Li Ma, Xiao You, Wenting Zhou, Yunpeng Ye, Wen Liu, Xiangkun Xu, Yongsheng Zhang, Tiantian Dong, Jiawei Zhu, Zhe Wang, Xijian Ju, Jianxun Song, Haoliang Cheng, Xiaojing Li, Jiandong Ding, Hefei Guo, and Zhengyong Zhang. 2024. P/D-Serve: Serving Disaggregated Large Language Model at Scale. *CoRR* abs/2408.08147 (2024). https://doi.org/10.48550/ARXIV.2408.08147 arXiv:2408.08147

[29] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. 2021. Dynamic Tensor Rematerialization. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=Vfs_2RnOD0H

[30] Kubernetes. 2024. Kubernetes Scheduler. https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/

[31] Kubernetes. 2024. Node-pressure Eviction. https://kubernetes.io/docs/concepts/scheduling-eviction/node-pressure-eviction/

[32] Kubernetes. 2025. Kubernetes. https://kubernetes.io/ accessed on April 11, 2025.

[33] Ravi Kumar, Manish Purohit, Zoya Svitkina, Erik Vee, and Joshua R. Wang. 2019. Efficient Rematerialization for Deep Networks. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 15146–15155. https://proceedings.neurips.cc/paper/2019/hash/ffe10334251de1dc98339d99ae4743ba-Abstract.html

[34] 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, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace (Eds.). ACM, 611–626. https://doi.org/10.1145/3600006.3613165

[35] Christoph Lameter. 2013. An overview of non-uniform memory access. *Commun. ACM* 56, 9 (2013), 59–54. https://doi.org/10.1145/2500468.2500477

[36] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques A. Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021*, Jae W. Lee, Mary Lou Soffa, and Ayal Zaks (Eds.). IEEE, 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[37] Joonhyung Lee, Shmulik Markovich-Golan, Daniel Ohayon, Yair Hanani, Gunho Park, Byeongwook Kim, Asaf Karnieli, Uri Livne, Haihao Shen, Tai Huang, Se Jung Kwon, and Dongsoo Lee. 2025. Faster Inference of LLMs using FP8 on the Intel Gaudi. arXiv:2503.09975 [cs.AR] https://arxiv.org/abs/2503.09975

[38] Daniel Lemire and Travis Downs. 2018. AVX-512: when and how to use these new instructions. https://lemire.me/blog/2018/09/07/avx-512-when-and-how-to-use-these-new-instructions/

[39] Jiang Li. 2025. [Hardware] [Intel] Enable Multiprocessing and tensor parallel in CPU backend and update documentation. https://github.com/vllm-project/vllm/pull/6125#

[40] Jianhui Li, Zhennan Qin, Yijie Mei, Jingze Cui, Yunfei Song, Ciyong Chen, Yifei Zhang, Longsheng Du, Xianhang Cheng, Baihui Jin, Yan Zhang, Jason Ye, Eric Lin, and Dan Lavery. 2024. oneDNN Graph Compiler: A Hybrid Approach for High-Performance Deep Learning Compilation. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2024, Edinburgh, United Kingdom, March 2-6, 2024*, Tobias Grosser, Christophe Dubach, Michel Steuwer, Jingling Xue, Guilherme Ottoni, and ernando Magno Quintão Pereira (Eds.). IEEE, 460–470. https://doi.org/10.1109/CGO57630.2024.10444871

[41] Peng Li, Yeye He, Dror Yashar, Weiwei Cui, Song Ge, Haidong Zhang, Danielle Rifinski Fainman, Dongmei Zhang, and Surajit Chaudhuri. 2024. Table-GPT: Table Fine-tuned GPT for Diverse Table Tasks. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 176. https://doi.org/10.1145/3654979

[42] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. 2024. AWQ: Activation-aware Weight Quantization for On-Device LLM Compression and Acceleration. In *Proceedings of the Seventh Annual Conference on Machine Learning and Systems, MLSys 2024, Santa Clara, CA, USA, May 13-16, 2024*, Phillip B. Gibbons, Gennady Pekhimenko, and Christopher De Sa (Eds.). mlsys.org. https://proceedings.mlsys.org/paper_files/paper/2024/hash/42a452cbafa9dd64e9ba4aa95cc1ef21-Abstract-Conference.html

[43] Yujun Lin, Haotian Tang, Shang Yang, Zhekai Zhang, Guangxuan Xiao, Chuang Gan, and Song Han. 2024. QServe: W4A8KV4 Quantization and System Co-design for Efficient LLM Serving. *arXiv preprint arXiv:2405.04532* (2024).

[44] Shih-Yang Liu, Zechun Liu, Xijie Huang, Pingcheng Dong, and Kwang-Ting Cheng. 2023. LLM-FP4: 4-Bit Floating-Point Quantized Transformers. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, 592–605. https://doi.org/10.18653/V1/2023.EMNLP-MAIN.39

[45] Haoyu Lu, Wen Liu, Bo Zhang, Bingxuan Wang, Kai Dong, Bo Liu, Jingxiang Sun, Tongzheng Ren, Zhuoshu Li, Hao Yang, Yaofeng Sun, Chengqi Deng, Hanwei Xu, Zhenda Xie, and Chong Ruan. 2024. DeepSeek-VL: Towards Real-World Vision-Language Understanding. *CoRR* abs/2403.05525 (2024). https://doi.org/10.48550/ARXIV.2403.05525 arXiv:2403.05525

[46] Peter Mattson, Christine Cheng, Gregory F. Diamos, Cody Coleman, Paulius Micikevicius, David A. Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David Brooks, Dehao Chen, Debo Dutta, Udit Gupta, Kim M. Hazelwood, Andy Hock, Xinyuan Huang, Daniel Kang, David Kanter, Naveen Kumar, Jeffery Liao, Deepak Narayanan, Tayo Oguntebi, Gennady Pekhimenko, Lillian Pentecost, Vijay Janapa Reddi, Taylor Robie, Tom St. John, Carole-Jean Wu, Lingjie Xu, Cliff Young, and Matei Zaharia. 2020. MLPerf Training Benchmark. In *Proceedings of the Third Conference on Machine Learning and Systems, MLSys 2020, Austin, TX, USA, March 2-4, 2020*, Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze (Eds.). mlsys.org. https://proceedings.mlsys.org/paper_files/paper/2020/hash/411e39b117e885341f25efb8912945f7-Abstract.html

[47] Peter Mattson, Hanlin Tang, Gu-Yeon Wei, Carole-Jean Wu, Vijay Janapa Reddi, Christine Cheng, Cody Coleman, Greg Diamos, David Kanter, Paulius Micikevicius, David A. Patterson, and Guenther Schmuelling. 2020. MLPerf: An Industry Standard Benchmark Suite for Machine Learning Performance. *IEEE Micro* 40, 2 (2020), 8–16. https://doi.org/10.1109/MM.2020.2974843

[48] Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. 2024. SpotServe: Serving Generative Large Language Models on Preemptible Instances. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir (Eds.). ACM, 1112–1127. https://doi.org/10.1145/3620665.3640411

[49] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory F. Diamos, Erich Elsen, David García, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. 2018. Mixed Precision Training. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=r1gs9JgRZ

[50] Paulius Micikevicius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, Naveen Mellempudi, Stuart F. Oberman, Mohammad Shoeybi, Michael Y. Siu, and Hao Wu. 2022. FP8 Formats for Deep Learning. *CoRR* abs/2209.05433 (2022). https://doi.org/10.48550/ARXIV.2209.05433 arXiv:2209.05433

[51] NVIDIA. 2025. TensorRT-LLM: A TensorRT Toolbox for Optimized Large Language Model Inference. https://github.com/NVIDIA/TensorRT-LLM accessed on April 11, 2025.

[52] NVIDIA. 2025. Triton Client Libraries and Examples. https://github.com/triton-inference-server/client accessed on April 11, 2025.

[53] NVIDIA. 2025. Triton Inference Server: An Optimized Cloud and Edge Inferencing Solution. https://github.com/triton-inference-server/server accessed on April 11, 2025.

[54] OpenMP. 2018. OpenMP Application Programming Interface. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf

[55] OpenVINO™ Toolkit. 2025. OpenVINO™: Open-source software toolkit for optimizing and deploying deep learning models. https://github.com/openvinotoolkit/openvino accessed on April 11, 2025.

[56] pagggga. 2024. No A100 resource availability. https://www.reddit.com/r/googlecloud/comments/1cb8ddu/no_a100_resource_availability/?utm_source=share&utm_medium=web3x&utm_name=web3xcss&utm_term=1&utm_content=share_button

[57] PassMark Software. 2025. AMD EPYC 7K62. https://www.cpubenchmark.net/cpu.php?cpu=AMD+EPYC+7K62&id=4859 accessed on April 11, 2025.

[58] PassMark Software. 2025. AMD EPYC 7K83. https://www.cpubenchmark.net/cpu.php?cpu=AMD+EPYC+7K83&id=4932&cpuCount=2 accessed on April 11, 2025.

[59] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 8024–8035. https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html

[60] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient Generative LLM Inference Using Phase Splitting. In *51st ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2024, Buenos Aires, Argentina, June 29 - July 3, 2024*. IEEE, 118–132. https://doi.org/10.1109/ISCA59077.2024.00019

[61] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. 2020. Capuchin: Tensor-based GPU Memory Management for Deep Learning. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, James R. Larus, Luis Ceze, and Karin Strauss (Eds.). ACM, 891–905. https://doi.org/10.1145/3373376.3378505

[62] Peter. 2024. Cannot use p4 instances. https://repost.aws/questions/QUSyWmoKjASo-6lSqBRqjLmA/cannot-use-p4-instances

[63] Python. 2025. asyncio — Asynchronous I/O. https://docs.python.org/3/library/asyncio.html accessed on April 11, 2025.

[64] Python. 2025. multiprocessing — Process-based parallelism. https://docs.python.org/3/library/multiprocessing.html accessed on April 11, 2025.

[65] PyTorch. 2024. Performance Tuning Guide. https://pytorch.org/tutorials/recipes/recipes/tuning_guide.html

[66] Ruoyu Qin, Zheming Li, Weiran He, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2024. Mooncake: A KVCache-centric Disaggregated Architecture for LLM Serving. *CoRR* abs/2407.00079 (2024). https://doi.org/10.48550/ARXIV.2407.00079 arXiv:2407.00079

[67] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. ZeRO-infinity: breaking the GPU memory wall for extreme scale deep learning. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*, Bronis R. de Supinski, Mary W. Hall, and Todd Gamblin (Eds.). ACM, 59. https://doi.org/10.1145/3458817.3476205

[68] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. 2020. MLPerf Inference Benchmark. In *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Virtual Event / Valencia, Spain, May 30 - June 3, 2020*. IEEE, 446–459. https://doi.org/10.1109/ISCA45697.2020.00045

[69] RedHat. 2024. Chapter 5. Memory Usage and Page Cache. https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/5/html/tuning_and_optimizing_red_hat_enterprise_linux_for_oracle_9i_and_10g_databases/chap-oracle_9i_and_10g_tuning_guide-memory_usage_and_page_cache#sect-Oracle_9i_and_10g_Tuning_Guide-Memory_Usage_and_Page_Cache-Checking_the_Memory_Usage

[70] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. ZeRO-Offload: Democratizing Billion-Scale Model Training. In *Proceedings of the 2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, Irina Calciu and Geoff Kuenning (Eds.). USENIX Association, 551–564. https://www.usenix.org/conference/atc21/presentation/ren-jie

[71] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. 2016. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*. IEEE Computer Society, 18:1–18:13. https://doi.org/10.1109/MICRO.2016.7783721

[72] Minsoo Rhu, Mike O'Connor, Niladrish Chatterjee, Jeff Pool, Youngeun Kwon, and Stephen W. Keckler. 2018. Compressing DMA Engine: Leveraging Activation Sparsity for Training Deep Neural Networks. In *IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24-28, 2018*. IEEE Computer Society, 78–91. https://doi.org/10.1109/HPCA.2018.00017

[73] Bita Darvish Rouhani, Ritchie Zhao, Ankit More, Mathew Hall, Alireza Khodamoradi, Summer Deng, Dhruv Choudhary, Marius Cornea, Eric Dellinger, Kristof Denolf, Dusan Stosic, Venmugil Elango, Maximilian Golub, Alexander Heinecke, Phil James-Roxby, Dharmesh Jani, Gaurav Kolhe, Martin Langhammer, Ada Li, Levi Melnick, Maral Mesmakhosroshahi, Andres Rodriguez, Michael Schulte, Rasoul Shafipour, Lei Shao, Michael Y. Siu, Pradeep Dubey, Paulius Micikevicius, Maxim Naumov, Colin Verilli, Ralph Wittig, Doug Burger, and Eric S. Chung. 2023. Microscaling Data Formats for Deep Learning. *CoRR* abs/2310.10537 (2023). https://doi.org/10.48550/ARXIV.2310.10537 arXiv:2310.10537

[74] s. 2022. Stub process is unhealthy and it will be restarted. https://github.com/triton-inference-server/server/issues/5144

[75] Amit Sabne. 2020. XLA : Compiling Machine Learning for Peak Performance.

[76] Dave Salvator, Hao Wu, Milind Kulkarni, and Niall Emmart. 2019. Int4 Precision for AI Inference. https://developer.nvidia.com/blog/int4-for-ai-inference/

[77] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *CoRR* abs/1909.08053 (2019). arXiv:1909.08053 http://arxiv.org/abs/1909.08053

[78] Gursimran Singh, Xinglu Wang, Ivan Hu, Timothy Yu, Linzi Xing, Wei Jiang, Zhefeng Wang, Xiaolong Bai, Yi Li, Ying Xiong, Yong Zhang, and Zhenan Fan. 2025. Efficiently serving large multimedia models using EPD Disaggregation. *CoRR* abs/2501.05460 (2025). https://doi.org/10.48550/ARXIV.2501.05460 arXiv:2501.05460

[79] Foteini Strati, Sara Mcallister, Amar Phanishayee, Jakub Tarnawski, and Ana Klimovic. 2024. DéjàVu: KV-cache Streaming for Fast, Fault-tolerant Generative LLM Serving. arXiv:2403.01876 [cs.DC] https://arxiv.org/abs/2403.01876

[80] Jennifer Switzer, Gabriel Marcano, Ryan Kastner, and Pat Pannuto. 2023. Junkyard Computing: Repurposing Discarded Smartphones to Minimize Carbon. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift (Eds.). ACM, 400–412. https://doi.org/10.1145/3575693.3575710

[81] tile ai. 2025. Tile Language. accessed on April 11, 2025.

[82] Philippe Tillet, Hsiang-Tsung Kung, and David D. Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2019, Phoenix, AZ, USA, June 22, 2019*, Tim Mattson, Abdullah Muzahid, and Armando Solar-Lezama (Eds.). ACM, 10–19. https://doi.org/10.1145/3315508.3329973

[83] Viperatech. 2025. NVIDIA HGX H20 Enterprise 96GB. https://viperatech.com/shop/nvidia-hgx-h20/ accessed on April 11, 2025.

[84] vLLM. 2025. CPU. https://docs.vllm.ai/en/latest/getting_started/installation/cpu/index.html

[85] Lei Wang, Lingxiao Ma, Shijie Cao, Quanlu Zhang, Jilong Xue, Yining Shi, Ningxin Zheng, Ziming Miao, Fan Yang, Ting Cao, Yuqing Yang, and Mao Yang. 2024. Ladder: Enabling Efficient Low-Precision Deep Learning Computing through Hardware-aware Tensor Transformation. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, Ada Gavrilovska and Douglas B. Terry (Eds.). USENIX Association, 307–323. https://www.usenix.org/conference/osdi24/presentation/wang-lei

[86] Peng Wang, Shuai Bai, Sinan Tan, Shijie Wang, Zhihao Fan, Jinze Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Yang Fan, Kai Dang, Mengfei Du, Xuancheng Ren, Rui Men, Dayiheng Liu, Chang Zhou, Jingren Zhou, and Junyang Lin. 2024. Qwen2-VL: Enhancing Vision-Language Model's Perception of the World at Any Resolution. *CoRR* abs/2409.12191 (2024). https://doi.org/10.48550/ARXIV.2409.12191 arXiv:2409.12191

[87] Ruizhe Wang, Yeyun Gong, Xiao Liu, Guoshuai Zhao, Ziyue Yang, Baining Guo, Zhengjun Zha, and Peng Cheng. 2025. Optimizing Large Language Model Training Using FP4 Quantization. *CoRR* abs/2501.17116 (2025). https://doi.org/10.48550/ARXIV.2501.17116 arXiv:2501.17116

[88] Shibo Wang and Pankaj Kanwar. 2019. BFloat16: The secret to high performance on Cloud TPUs. https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus

[89] WikiChip. 2022. Skylake (server) - Microarchitectures - Intel. https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server)

[90] WikiChip. 2023. AVX-512 BFloat16 Instructions (BF16) - x86. https://en.wikichip.org/wiki/x86/avx512_bf16

[91] WikiChip. 2023. Haswell - Microarchitectures - Intel. https://en.wikichip.org/wiki/intel/microarchitectures/haswell

[92] WikiChip. 2023. Zen - Microarchitectures - AMD. https://en.wikichip.org/wiki/amd/microarchitectures/zen

[93] WikiChip. 2023. Zen 4 - Microarchitectures - AMD. https://en.wikichip.org/wiki/amd/microarchitectures/zen_4

[94] WikiChip. 2024. Cooper Lake - Microarchitectures - Intel. https://en.wikichip.org/wiki/intel/microarchitectures/cooper_lake

[95] Wikipedia. 2025. List of Intel Xeon processors (Ice Lake-based). https://en.wikipedia.org/wiki/List_of_Intel_Xeon_processors_(Ice_Lake-based) accessed on April 11, 2025.

[96] Hao Wu, Patrick Judd, Xiaojie Zhang, Mikhail Isaev, and Paulius Micikevicius. 2020. Integer Quantization for Deep Learning Inference: Principles and Empirical Evaluation. *CoRR* abs/2004.09602 (2020). arXiv:2004.09602 https://arxiv.org/abs/2004.09602

[97] Xiaoxia Wu, Cheng Li, Reza Yazdani Aminabadi, Zhewei Yao, and Yuxiong He. 2023. Understanding Int4 Quantization for Language Models: Latency Speedup, Composability, and Failure Cases. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 37524–37539. https://proceedings.mlr.press/v202/wu23k.html

[98] Haocheng Xi, Han Cai, Ligeng Zhu, Yao Lu, Kurt Keutzer, Jianfei Chen, and Song Han. 2024. COAT: Compressing Optimizer states and Activation for Memory-Efficient FP8 Training. *CoRR* abs/2410.19313 (2024). https://doi.org/10.48550/ARXIV.2410.19313 arXiv:2410.19313

[99] Haojun Xia, Zhen Zheng, Xiaoxia Wu, Shiyang Chen, Zhewei Yao, Stephen Youn, Arash Bakhtiari, Michael Wyatt, Donglin Zhuang, Zhongzhu Zhou, Olatunji Ruwase, Yuxiong He, and Shuaiwen Leon Song. 2024. Quant-LLM: Accelerating the Serving of Large Language Models via FP6-Centric Algorithm-System Co-Design on Modern GPUs. In *Proceedings of the 2024 USENIX Annual Technical Conference, USENIX ATC 2024, Santa Clara, CA, USA, July 10-12, 2024*, Saurabh Bagchi and Yiying Zhang (Eds.). USENIX Association, 699–713. https://www.usenix.org/conference/atc24/presentation/xia

[100] Guangxuan Xiao, Ji Lin, Mickaël Seznec, Hao Wu, Julien Demouth, and Song Han. 2023. SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 38087–38099. https://proceedings.mlr.press/v202/xiao23c.html

[101] An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, Jianxin Yang, Jin Xu, Jingren Zhou, Jinze Bai, Jinzheng He, Junyang Lin, Kai Dang, Keming Lu, Keqin Chen, Kexin Yang, Mei Li, Mingfeng Xue, Na Ni, Pei Zhang, Peng Wang, Ru Peng, Rui Men, Ruize Gao, Runji Lin, Shijie Wang, Shuai Bai, Sinan Tan, Tianhang Zhu, Tianhao Li, Tianyu Liu, Wenbin Ge, Xiaodong Deng, Xiaohuan Zhou, Xingzhang Ren, Xinyu Zhang, Xipin Wei, Xuancheng Ren, Xuejing Liu, Yang Fan, Yang Yao, Yichang Zhang, Yu Wan, Yunfei Chu, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, Zhifang Guo, and Zhihao Fan. 2024. Qwen2 Technical Report. *CoRR* abs/2407.10671 (2024). https://doi.org/10.48550/ARXIV.2407.10671 arXiv:2407.10671

[102] An Yang, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoyan Huang, Jiandong Jiang, Jianhong Tu, Jianwei Zhang, Jingren Zhou, Junyang Lin, Kai Dang, Kexin Yang, Le Yu, Mei Li, Minmin Sun, Qin Zhu, Rui Men, Tao He, Weijia Xu, Wenbiao Yin, Wenyuan Yu, Xiafei Qiu, Xingzhang Ren, Xinlong Yang, Yong Li, Zhiying Xu, and Zipeng Zhang. 2025. Qwen2.5-1M Technical Report. *CoRR* abs/2501.15383 (2025). https://doi.org/10.48550/ARXIV.2501.15383 arXiv:2501.15383

[103] Zhewei Yao, Reza Yazdani Aminabadi, Minjia Zhang, Xiaoxia Wu, Conglong Li, and Yuxiong He. 2022. ZeroQuant: Efficient and Affordable Post-Training Quantization for Large-Scale Transformers. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (Eds.). http://papers.nips.cc/paper_files/paper/2022/hash/adf7fa39d65e2983d724ff7da57f00ac-Abstract-Conference.html

[104] Jeffrey Yu, Kartik Prabhu, Yonatan Urman, Robert M. Radway, Eric Han, and Priyanka Raina. 2024. 8-bit Transformer Inference and Fine-tuning for Edge Accelerators. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir (Eds.). ACM, 5–21. https://doi.org/10.1145/3620666.3651368

[105] Xianyi Zhang, Qian Wang, and Yunquan Zhang. 2012. Model-driven Level 3 BLAS Performance Optimization on Loongson 3A Processor. In *18th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2012, Singapore, December 17-19, 2012*. IEEE Computer Society, 684–691. https://doi.org/10.1109/ICPADS.2012.97

[106] Bojian Zheng, Nandita Vijaykumar, and Gennady Pekhimenko. 2020. Echo: Compiler-based GPU Memory Footprint Reduction for LSTM RNN Training. In *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Virtual Event / Valencia, Spain, May 30 - June 3, 2020*. IEEE, 1089–1102. https://doi.org/10.1109/ISCA45697.2020.00092

[107] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 863–879. https://www.usenix.org/conference/osdi20/presentation/zheng

[108] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark W. Barrett, and Ying Sheng. 2024. SGLang: Efficient Execution of Structured Language Model Programs. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (Eds.). http://papers.nips.cc/paper_files/paper/2024/hash/724be4472168f31ba1c9ac630f15dec8-Abstract-Conference.html

[109] 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. *CoRR* abs/2401.09670 (2024). https://doi.org/10.48550/ARXIV.2401.09670 arXiv:2401.09670

[110] Neta Zmora, Hao Wu, and Jay Rodge. 2021. Achieving FP32 Accuracy for INT8 Inference Using Quantization Aware Training with NVIDIA TensorRT. https://developer.nvidia.com/blog/achieving-fp32-accuracy-for-int8-inference-using-quantization-aware-training-with-tensorrt/