DéjàVu: KV-cache Streaming for Fast, Fault-tolerant Generative LLM Serving

Foteini Strati [1 2] **Sara McAllister** [1 3] **Amar Phanishayee** [4] **Jakub Tarnawski** [4] **Ana Klimovic** [2]

Abstract

Distributed LLM serving is costly and often underutilizes hardware accelerators due to three key challenges: bubbles in pipeline-parallel deployments caused by the bimodal latency of prompt and token processing, GPU memory overprovisioning, and long recovery times in case of failures. DéjàVu addresses all these challenges using a versatile and efficient KV cache streaming library (DéjàVuLib). Using DéjàVuLib, we propose and implement efficient prompt-token disaggregation to reduce pipeline bubbles, microbatch swapping for efficient GPU memory management, and state replication for fault-tolerance. We highlight the efficacy of these solutions on a range of large models across cloud deployments.

1. Introduction

Large Language Models (LLMs) like GPT-3 (Brown et al., 2020) are widely used in chatbots, code generation, and text summarization. Two key trends in generative LLM inference have changed the landscape of ML model serving. First, large model sizes, input sequence lengths, and consequently large intermediate inference state lead to high memory footprint for LLM inference. Figure 1 shows the GPU memory required to serve various generative LLMs with a 2K sequence length; their memory footprint greatly exceeds the capacity of a single GPU, mandating parallelization across many high-end GPUs (including tensor-model and pipeline parallel execution). Second, for low latency serving, these LLMs use a *Key-Value Cache* to store prior computations as individual tokens are generated for each request (Yan et al., 2021; Pope et al., 2022; Kwon et al., 2023; Sheng et al., 2023; Zhang et al., 2023). While ML inference has been traditionally stateless, the use of KV cache makes generative LLM inference *stateful*.

[1]MSR Project Fiddle Intern [2]ETH Zurich [3]Carnegie Mellon University [4]Microsoft Research. Correspondence to: Foteini Strati <foteini.strati@inf.ethz.ch>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024. Copyright 2024 by the author(s).

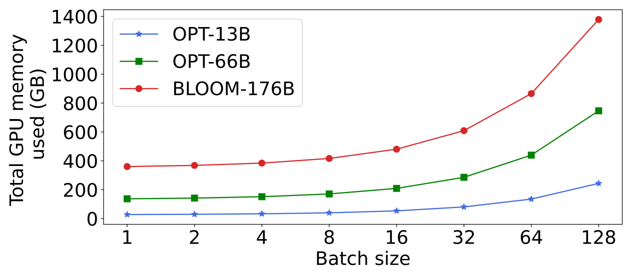


Figure 1. Memory footprint of serving various LLMs with 2K sequence length (input + generated tokens) and half precision (fp16).

Given these trends, we identify three key challenges in stateful, distributed LLM serving. First, we observe a substantial latency discrepancy (up to 2 orders of magnitude) between two phases of LLM serving, which leads to expensive GPU underutilization. Prompt processing, also called the *prefill phase* (Zhong et al., 2024), depends on the input size and is compute bound. Meanwhile, token generation, also called the *decode phase*, is memory bandwidth-bound and the time to generate a token is nearly constant when using the KV cache. Processing both prompts and tokens in the same pipeline introduces *pipeline bubbles*, where GPUs idle.

Second, state-of-the-art LLM serving systems like FasterTransformer vastly *overprovision the KV cache* in pipeline parallel configurations by allocating GPU memory for all microbatches upfront (NVIDIA, 2023b). Since the KV cache is only used by one microbatch at a time, there is an opportunity to allocate GPU memory more efficiently.

Third, existing LLM serving systems do not efficiently *handle failures* or preemptions, which often occur in large-scale GPU deployments (Eisenman et al., 2022). Upon a failure, the LLM serving system crashes and stalls all in-flight requests. When KV cache state is lost, current systems process requests from scratch. These redundant computations severely increase end-to-end request latency.

To address the above challenges for pipeline-parallel distributed inference, we propose DéjàVu, an efficient and fault-tolerant LLM serving system based on KV cache streaming. First, DéjàVu *disaggregates* prompt processing from token generation and optimizes the number of machines for each stage to satisfy GPU memory capacity constraints and avoid GPU idle times. Second, to effectively use GPU memory capacity, DéjàVu *swaps* KV cache state per-microbatch between the GPU and CPU, maximizing GPU memory allocation for each microbatch being processed. Third, DéjàVu *replicates* KV cache state to avoid losing state and employs fast recovery mechanism to minimize lost work on failures.

The core component in DéjàVu that enables all these optimizations is an efficient and versatile KV cache streaming library, DéjàVuLib. We build DéjàVuLib as a modular set of primitives that enable fast streaming for diverse configurations, such as streaming between local or remote machines and for a variety of different KV cache structures.

We evaluate DéjàVu under different use cases. In pipeline parallel configurations without failures, DéjàVu improves LLM serving throughput by up to $2\times$ compared to FasterTransformer. We show that DéjàVu's microbatch swapping can improve throughput by up to $1.8\times$ by accommodating larger batch size for models that already fit in the given deployment. This enables serving even larger models that might not fit using existing state-of-the-art LLM systems. In the presence of system failures, DéjàVu reduces microbatch latency by $1.54\times$ compared to non-fault-tolerant systems. DéjàVu is available at https://github.com/msr-fiddle/dejavu.

2. Background and Motivation

2.1. Generative LLM inference

Generative LLM inference involves two phases: *prompt processing* (or *prefill*) and *autoregessive token generation* (or *decode*). In the prompt processing phase, the model processes a user-defined sentence (i.e., prompt) provided as input and generates a new token. During autoregressive token generation, which spans multiple steps, the model generates new tokens one by one, using the token generated at step i as input for step $i + 1$. This continues until a prespecified number of tokens or until the special EOS token is generated.

A crucial component of an autoregressive LLM is the *attention* mechanism. Upon each step of token generation, each attention layer applies transformations to the input, to extract the *query, key*, and *value* vectors. At each generation step i, the attention mechanism computes the *attention score* and token probability using the query vector at position i, and the key and value vectors at positions $[0, i-1]$. Thus, the computations and output at each step depend on the keys and values of the generated tokens at the previous steps. To avoid recomputing the key and value vectors of all processed tokens at each step, LLM inference frameworks store the vectors in the *KV cache* (Ott et al., 2019; Yan et al., 2021; Pope et al., 2022).

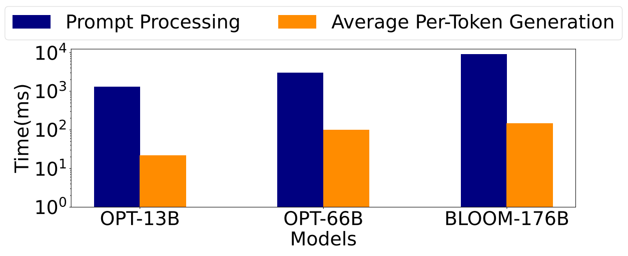


Figure 2. Prompt processing and average per-token generation time on A100 GPUs, using FasterTransformer (with batch size 8 and prompt size 1000). Y-axis is in log scale.

During prompt processing, the key and value vectors of all tokens in the prompt are generated, populating the KV cache. Since all prompt tokens are known, computations during prompt processing use matrix-matrix multiplications and tend to be compute-bound. At each subsequent token generation step, the KV vectors for the newly generated token are appended in the KV cache. This phase is memory-bandwidth-bound (Jin et al., 2023).

The KV cache size depends on the number of layers, hidden units per layer, floating point precision, batch size, and sequence length tokens (Sheng et al., 2023). Larger models, batch sizes, or longer generated sequences lead to a larger KV cache memory footprint. Most LLM inference frameworks preallocate GPU memory for the KV cache for performance and often overprovision for the model's maximum supported sequence length (NVIDIA, 2023b). vLLM (Kwon et al., 2023) proposes *PagedAttention* to dynamically allocate GPU memory for the KV cache.

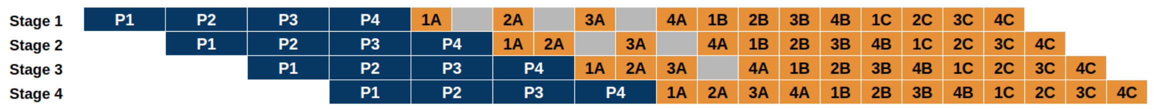
As LLM serving requires 100s of GB of GPU memory (Figure 1), LLM inference is distributed across multiple GPUs, with pipeline and tensor parallelism. Tensor parallelism requires very fast interconnects limiting it to single-node boundaries (Narayanan et al., 2021; Jiang et al., 2024); pipeline parallelism is additionally required for cross-node scaling[1]. With pipeline parallelism, model layers are split across stages, with adjacent stages exchanging activations, and multiple micro-batches used to keep all stages busy.

2.2. Challenges of distributed LLM serving

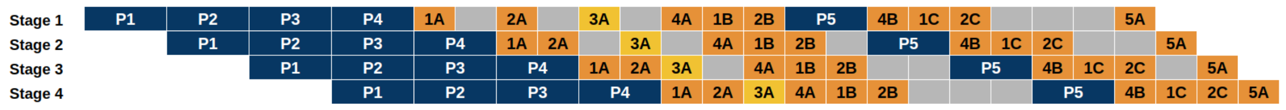
2.2.1. Bimodal prompt vs. token-gen latency

The first challenge in LLM serving comes from the disparity between prompt processing and token generation. Since the number of tokens processed during prompt processing is as large as the input sequence length, the prompt processing

[1]In this paper, we always use a combination of parallelization schemes: tensor-model parallel within a stage (multiple GPUs on a single server) and pipeline parallel across stages (servers)


Stage 1 P1 P2 P3 P4 1A 2A 3A 4A 1B 2B 3B 4B 1C 2C 3C 4C
Stage 2 P1 P2 P3 P4 1A 2A 3A 4A 1B 2B 3B 4B 1C 2C 3C 4C
Stage 3 P1 P2 P3 P4 1A 2A 3A 4A 1B 2B 3B 4B 1C 2C 3C 4C
Stage 4 P1 P2 P3 P4 1A 2A 3A 4A 1B 2B 3B 4B 1C 2C 3C 4C

(a) Baseline


Stage 1 P1 P2 P3 P4 1A 2A 3A 4A 1B 2B P5 4B 1C 2C 5A
Stage 2 P1 P2 P3 P4 1A 2A 3A 4A 1B 2B P5 4B 1C 2C 5A
Stage 3 P1 P2 P3 P4 1A 2A 3A 4A 1B 2B P5 4B 1C 2C 5A
Stage 4 P1 P2 P3 P4 1A 2A 3A 4A 1B 2B P5 4B 1C 2C 5A

(b) Baseline, with request 3 stopping earlier than the rest (at $3A$)

Figure 3. LLM serving with a 4-stage pipeline. A stage is a machine with n GPUs running a set of layers with tensor model parallelism. Px shows prompt processing of microbatch x. X_y shows token generation for token y, microbatch X. For simplicity, in this figure, we assume prompt processing time takes $2\times$ per-token processing time. In reality, the prompt-token difference can be up to $106\times$ (see Appendix A). Grey areas are bubbles due to prompt processing vs. token generation latency discrepancy.

phase usually takes longer than the subsequent token generation phase. Figure 2 shows prompt processing and per-token generation times for various LLMs. Prompt processing can be more than an order of magnitude higher than per-token generation, depending on the model and batch size. In our study, prompt processing latency is up to $106\times$ higher than per-token generation (see Appendix A for details).

With pipeline parallelism, this difference in execution time between the two stages causes *pipeline bubbles*, leaving some stages idle while waiting for others to finish. For example, Figure 3a shows a 4-stage pipeline, with 4 microbatches. Each microbatch consists of the prompt processing step, and multiple token generation steps. We observe bubbles in the pipeline, e.g. for token generation step $3A$ to start at Stage 1 for microbatch 3, Stage 4 must have completed prompt processing (generating the first token) for this microbatch ($P3$). Because prompt processing is slower than token generation, Stage 1 stalls.

The problem of pipeline bubbles becomes even more pronounced with *early stopping* of microbatches, where certain requests complete earlier than others. To keep the pipeline full, existing frameworks like Orca (Yu et al., 2022) and HuggingFace Accelerate (HuggingFace, 2024) will introduce a new microbatch, which will go through the prompt processing phase, disturbing the token generation of unfinished microbatches. Figure 3b shows an example of early stopping, where microbatch 3 finishes earlier than the others (at step $3A$), and is replaced by a new microbatch ($P5$). The difference in processing time between prompt and token steps introduces bubbles in the pipeline.

2.2.2. Inefficient use of GPU memory

In pipeline parallel settings, multiple microbatches should be processed concurrently by the different stages, to keep all stages busy. For example, in Figure 3a, 4 microbatches are in-flight at each stage. While the prompt processing phase happens only once for each microbatch, each microbatch goes through multiple token generation steps. Due to data dependencies between the different stages, the microbatches are processed in a *round-robin* fashion. Each microbatch has its own KV cache.

To increase performance, existing frameworks (NVIDIA, 2023b) preallocate the KV caches of all microbatches in GPU memory. However, since microbatches are processed sequentially at each stage, only the KV cache memory for a single microbatch is used at a time. Hence, memory is overprovisioned.

2.2.3. Statefulness and failure handling

Due to its large memory footprint (see Figure 1), LLM inference typically spans multiple GPUs across multiple nodes. In a distributed setup, failures are inevitable. Industry studies emphasize the prevalence of failures in ML training jobs. Meta reports that 50% of the jobs encounter a failure within less than 16 minutes of execution (Eisenman et al., 2022), while Microsoft notes that training jobs often suffer from hardware or software failures (Jeon et al., 2019). Although these studies focus on iterative training jobs, the causes of software and hardware failures can also affect inference. Due to data dependencies between stages in a pipeline parallel inference setup, a failure in one stage leads all remaining stages to idle, or even results in timeouts and cascading failures, downgrading the throughput of LLM serving.

The impact of these failures in LLM inference is exacerbated by its stateful nature, due to the use of the KV cache. Since the KV cache is typically stored in GPU memory for fast accesses, an accelerator failure would result in loss of cached data for an inference request which in turn would require that all work for that request is redone. Figure 4 shows a toy example of a GPT2-1.5B model serving a request

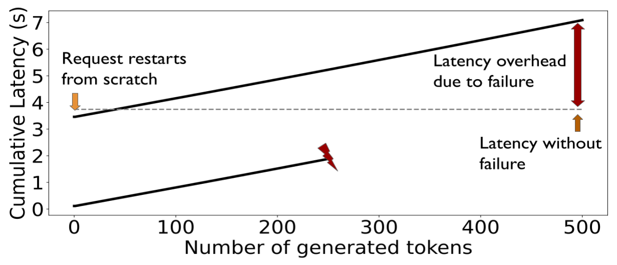


Figure 4. Effect on cumulative latency of an inference request when a failure occurs in today's systems, on a GPT2-1.5B model with a prompt size of 500 tokens, and generating 500 new tokens. After the first 250 tokens are generated, a failure occurs. Existing LLM serving systems lack a fault tolerance mechanism, resorting to restarting the entire request. This involves repopulating the KV cache, thus reprocessing the prompt and regenerating tokens up to the point of failure. In our illustrative example, this approach results in a 1.89× increase in the end-to-end latency of the request. This issue is magnified with pipeline parallelism and multiple requests grouped in microbatches, where a failure in one stage will cause the whole pipeline and the requests across in-flight microbatches to restart from scratch.

3. Proposed Solutions

First, to mitigate pipeline bubbles, we propose *disaggregating* prompt processing from token generation, by allocating separate machines for each task. By avoiding mixing prompt and token tasks, disaggregation helps reduce the pipeline bubbles and leads to higher throughput. However, disaggregation's effectiveness relies on the swift transfer of the prompt KV cache, which can be a bottleneck especially since user-submitted prompts grow in size (Ding et al., 2023), underscoring the need for an efficient KV cache streaming mechanism. Additionally, a key challenge is how to partition the available resources into prompt processing and token generation, to optimize system throughput. While disaggregation has also been recently proposed by concurrent related work (Patel et al., 2023; Zhong et al., 2024), we employ it to mitigate bubbles in pipeline-parallel settings (mandated by large generative models) and our planner is grounded in using a principled approach to optimally partition the available resources to maximize system throughput.

Second, to optimize GPU memory usage, we propose *swapping* the KV cache between GPU and CPU at the microbatch level [2]. The KV caches for all in-flight microbatches are stored in the CPU, and transferred to the GPU only when the respective microbatch is processed. This dramatically reduces GPU memory requirements, enabling larger batch sizes, and facilitating LLM serving under limited hardware (Sheng et al., 2023). However, CPU-GPU transfers through limited-bandwidth PCIe, can be a bottleneck, potentially leaving GPUs idle. Thus, we need an efficient mechanism to swap the KV cache in and out of the GPU.

[2]In contrast, vLLM(Kwon et al., 2023) employs swapping of the KV cache of individual requests, instead of microbatches.

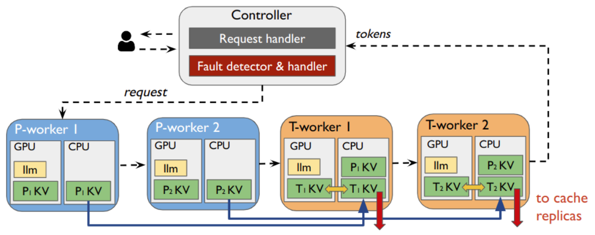


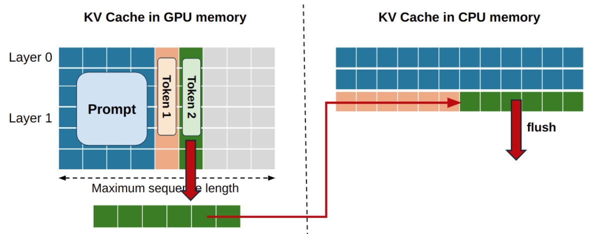
Figure 5. Full DéjàVu system diagram. When disaggregation is enabled, the workers do either only prompt processing (*P-worker*) or token generation (*T-worker*). The blue arrows stand for prompt-token cache exchange, the red arrows for cache replication, and the orange arrows for cache swapping.

Third, for fault tolerance, we propose *replicating* the KV cache in persistent storage or remote CPU memory. In the event of a failure, DéjàVu restores the most recent computed values to the failed GPUs, allowing inference to resume from the last generated token, and decreasing the recovery time compared to other LLM serving systems. To use such a system in practice, we need to minimize the overheads of KV cache streaming to storage or remote memory. We also need to make sure that failures can be detected and mitigated quickly to minimize recovery time.

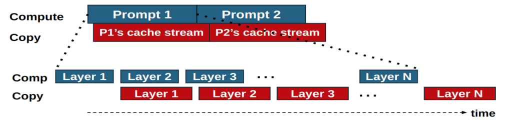
These solutions require a fast and versatile KV cache streaming mechanism. Next, we describe our KV cache streaming library, DéjàVuLib (§4.1), and how our system, DéjàVu, implements the proposed solutions using DéjàVuLib.

4. The DéjàVu LLM serving system

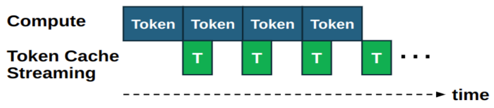
Figure 5 illustrates the DéjàVu system. A centralized controller coordinates inference. Workers are registered with the controller to serve requests. The workers send the tokens to the controller as they are generated. Each DéjàVu Worker has a *cache manager* that handles KV cache streaming. The cache manager is aware of the pipeline configuration (pipeline depths, prompt or token processing, batch sizes, etc). When a Worker needs to stream the KV cache out or into the GPU, the cache manager calls the appropriate DéjàVuLib primitives (4.1.2). We use NCCL (NVIDIA, 2023a) for GPU-GPU communication due to pipeline or tensor parallelism, and gRPC (grpc, 2023) for all control-plane communication (e.g. between workers and controller).



(a) Buffered Copies: We aggregate small updates in a contiguous buffer in the GPU and then copy this buffer out.



(b) Layer-by-layer pipelining of prompt KV cache streaming with computation. We also pipeline the streaming of microbatch i with prompt processing of microbatch $i+1$



(c) Pipelining of token streaming with computation

Figure 6. DéjàVuLib KV cache streaming optimizations

4.1. DéjàVuLib: A KV cache streaming library

4.1.1. Low-level implementation of DéjàVuLib

DéjàVu is built on top of FasterTransformer[3], supporting both tensor and pipeline parallelism. FasterTransformer preallocates GPU memory for the KV cache based on a maximum sequence length (either the maximum length supported by the model or user-defined). Figure 6a provides a simplified 2D representation of the key cache [4], including only the layer and the sequence length dimension.[5] Figure 6a also shows what happens in the key cache after processing a prompt of 4 words (denoted as *prompt*). Prompt processing occurs *layer-by-layer*, populating the respective portion of the cache at each layer. After the prompt has been processed, tokens are generated one by one. Figure 6a depicts key cache contents after 2 subsequent tokens have been generated (denoted as *Token 1* and *Token 2*). After the generation of each token, only a small, non-contiguous part of the Key cache is updated. The need to copy numerous non-contiguous small memory regions results in significant overhead, necessitating optimizations below:

(1) Buffered copies (Figure 6a): Individual token generation leads to multiple non-contiguous small updates in the KV caches. Employing multiple `cudaMemcpy` calls for copying these chunks, results in substantial overhead. Instead, we leverage the high bandwidth of GPU DRAM, and aggregate all updates in a temporary buffer within GPU memory. Once the temporary buffer has been populated, we copy it to the appropriate destination. Since these buffers are reused, the overhead in GPU memory capacity is negligible.

(2) Layer-by-layer prompt cache streaming (Figure 6b): Since prompt processing occurs in a layer-by-layer fashion, we also stream the prompt cache layer-by-layer. This is similar to *wait-free backpropagation* which overlaps backward pass computations with gradient exchange in distributed ML training (Zhang et al., 2017). In a pipeline parallel setup, we further parallelize streaming the prompt of microbatch i with computation of microbatch $i+1$.

(3) Token computation and streaming parallelization (Figure 6c): Unlike prompt processing, token generation for a single request involves multiple steps. In a single-machine configuration, we stream the KV cache for token i, while the generation of token $i+1$ is in progress. In a pipeline parallel configuration, we parallelize the cache streaming of microbatch i, token j, with computation of microbatch $i+1$, token j. Token computation, like prompt processing, occurs layer-by-layer. However, token streaming time can be fully masked behind subsequent token computation, so we do not use layer-by-layer streaming in this case.

We use a background CPU thread that is responsible for cache streaming and CUDA streams to parallelize KV cache streaming with computation on the GPU.

4.1.2. DéjàVuLib primitives

We built DéjàVuLib as a versatile library that handles different configurations, addressing the challenges encountered when developing our solutions described in 3. The source, destination, data volume, and transferring method of KV cache streaming depend on the pipeline setup, and network topology. For instance, when disaggregating prompt processing from token generation, the prompt KV cache is transferred from the prompt processing to the token generation machines. This can occur through various mechanisms, such as GPU-GPU or CPU-CPU copies. Moreover, the prompt and token pipelines might have different pipeline

[3]We chose FasterTransformer since it supports tensor-model and pipeline parallel inference, and the newly released TensorRT-LLM (NVIDIA, 2023c) has not open-sourced their KV cache manager yet. Other frameworks such as vLLM do not support pipeline parallelism at the moment (Kwon et al., 2023).

[4]The value cache follows a similar structure.

[5]In reality, the key cache is a 6D tensor, and the value cache is a 5D tensor. The extra dimensions include the number of attention heads and batch size.

Table 1. DéjàVuLib primitives. The `stream_out`,`stream_in` call `scatter`,`gather`, which call `flush` and `fetch` respectively.

Primitives	Functionality
`stream_out`, `stream_in`	Given a source (or destination) worker, the KV cache, and the inference setup (pipeline depths, batch sizes), find the proper destinations (or sources) for the different chunks of KV cache. This might involve splitting the cache at source or merging cache chunks at destination.
`scatter`, `gather`	Given a non-contiguous region of KV cache, and a local or remote destination (or source), chunk the region to contiguous transfers and orchestrate movement.
`flush`, `fetch`	Copy a contiguous chunk of KV cache. Local copies with CUDA, and remote copies with NCCL (NVIDIA, 2023a), MPI (OpenMPI, 2023), or Boost (Boost, 2021) are supported.

depths and batch sizes, requiring splitting or merging the KV cache at the source and destination respectively. DéjàVuLib aims to account for the diverse set of configurations that require KV cache streaming, abstracting away the implementation details from the high-level handling of the KV cache while offering efficient solutions depending on the type of streaming. We achieve this by offering primitives with different levels of abstraction, shown in table 1.

4.2. Detailed description of the proposed solutions

4.2.1. Prompt-Token disaggregation

Workers are categorized into 2 groups: prompt processing, and token generation (Figure 5). The Controller assigns incoming requests to the prompt workers, which generate the first token, populating the KV cache, which is then transferred to the token generation machines. With disaggregation, we need to ensure that: 1) we optimally allocate resources for each phase, and 2) we transfer the KV cache from prompt to token machines with minimal overheads.

1. Principled allocation of resources: Given a fixed set of machines, we want to partition them into prompt and token processing to satisfy the following requirements: 1) the aggregate memory footprint (model parameters and KV cache), for the active microbatches should fit into the aggregate GPU memory capacity for each pipeline, and 2) the throughput of the disaggregated system should be maximized, and ideally be higher than the throughput of the non-disaggregated system.

Appendix D describes the theoretical foundations of our resource allocation planner. At a high level, for requirement (1), equations 2 and 3 establish a lower limit on the number of machines required for prompt and token processing respectively. For requirement (2), since we want to maximize the throughput of the disaggregated system, we need to ensure that the prompt processing and token generation pipelines have comparable throughput. Assume a system with D machines, with prompt processing time per microbatch Y, token generation time per microbatch t, and N tokens generated per request. The number of machines for token generation D_t, and the number of machines for prompt processing D_p is given by formula 1 (m is the additional prompt processing overhead due to cache streaming).

$$D_t = \frac{D \cdot N \cdot t}{m \cdot Y + N \cdot t} \text{ and } D_p = \frac{D \cdot m \cdot Y}{m \cdot Y + N \cdot t} \quad (1)$$

When the overhead of prompt KV cache streaming is small, the disaggregated system will have higher throughput than the baseline (see Appendix D).

2. Fast prompt KV cache transfers: We use optimizations from 4.1 to pipeline layer-by-layer prompt KV cache streaming with prompt processing. To avoid overloading GPU memory, we transfer the KV cache to local CPU memory, and then to CPU memory of the token machine. Prompt and token pipelines might have different pipeline depths, or different batch sizes. This may require splitting the KV cache to multiple token machines or merging from different prompt machines. The cache manager invokes the `stream_out` primitive which calls the lower-level primitives (Table 1) based on pipeline depth and batch size. Whenever a prompt is needed, the token machines check if there are any prompt KV caches in their local CPU memory. When prompt KV cache becomes available, it is loaded into GPU memory and token generation starts.

4.2.2. Microbatch swapping

To facilitate microbatch swapping with minimal overhead we leverage all three optimizations presented in 4.1. For a pipeline of depth D, where D microbatches are active at a time, and each microbatch requires M GB, we allocate $D \cdot M$ GB in CPU memory, and $2 \cdot M$ GB in GPU memory. Before token generation step t for microbatch x starts, DéjàVu prefetches the KV cache for this microbatch from CPU to GPU (*swap in*). After step t has finished, the DéjàVu cache manager transfers the updated part of microbatch x's KV cache, corresponding to step t, back to the CPU (*swap out*). See Figure 26 in Appendix G for a visual representation of microbatch swapping over time.

4.2.3. Failure handling

For fault tolerance, we need to ensure that 1) KV cache replication has minimal overheads, and 2) failures are detected and mitigated quickly to minimize recovery time. Figure 24 in appendix E provides a toy example with 4 token generation workers. The DéjàVu Controller is responsible for detecting and mitigating failures. The workers send heartbeats to the controller periodically. If the controller has not received a heartbeat from a worker within a specified timeframe, it identifies the worker as failed, and notifies the rest workers to stop serving requests. To recover from the failure, we need to 1) restore the lost KV caches, and 2) determine the step and microbatch from which the inference should resume.

Each worker x streams its KV cache to worker $(x+1)\%N$ (assuming an N-stage pipeline). For example, in Figure 24, the worker at Stage 1 streams its cache to Stage 2, Stage 2 to Stage 3, etc. The cache is streamed incrementally, as each token is generated, and takes place asynchronously (in parallel) to computation (see 4.1). We define a background thread at each worker that is responsible for receiving the KV cache from its peer. When a worker x fails, both its own KV cache and the replica KV cache of worker $(x-1)\%N$ is lost. During recovery, we make sure the lost caches are repopulated at worker x.

Upon receiving the KV cache update from worker $(x-1)\%N$, for microbatch j and generation step t, worker x sends a message to the controller of the form (x, j, t). Therefore, the controller is aware of the KV cache replication status across all workers. In the event of failure, we follow a four-step process for recovery. First, worker $(x+1)\%N$ sends the replica KV cache it hosts to worker x (repopulating x's lost cache). Second, worker $(x-1)\%N$ sends its KV cache to x (repopulating the lost replica at x). Third, the controller finds the microbatch j and step t, that needs to be re-executed, since the cache of failed worker x has not been replicated up to that point. Finally, as stage x requires input from its preceding stages to re-execute a microbatch, the controller propagates (j, t) to all workers, and Stage 1 resumes inference from microbatch j and step t.

As a concrete example, consider the scenario in Figure 24, where stage 2 fails. First, stage 3 will copy stage 2's KV cache replica back to stage 2. Second, stage 1 will copy its own KV cache to stage 2. Third, the controller identifies the microbatch j and step t that needs to be reexecuted. In that case, $j == 1$ and $t == C$, since stage 2's KV cache for $1C$ had not been replicated before the failure. If stage 2 restarts from $1C$, it needs to get inputs (activations) from stage 1. Thus, finally, all stages execute $1C$.

5. Evaluation

Setup We use VMs with 2 A100-80GB GPUs, and inter-VM network bandwidth of 40 Gbps.
Models We use HuggingFace versions of GPT2, OPT and BLOOM, adapted for FasterTransformer. We use half-precision for all models.
Experiments Section 5.1 evaluates DéjàVuLib with microbenchmars. We evaluate the DéjàVu disaggregation policy, microbatch swapping, and fault-tolerance functionality in sections 5.2.1, 5.2.2 and 5.2.3 respectively.
Baselines We compare DéjàVu with FasterTransformer, as it supports pipeline parallelism for LLM inference. The original FasterTransformer follows a very rigid paradigm, where all microbatches must finish generating tokens before any new microbatches are introduced. This avoids mixing prompt and token processing, but it introduces bubbles in the pipeline since not enough microbatches are now in-flight to fully utilize the pipeline. Since works such as Orca (Yu et al., 2022) allow more flexible scheduling, we modified FasterTransformer to allow scheduling at the microbatch level. Whenever a microbatch completed in any stage, it can be replaced by the next available microbatch. Appendix H.1 shows that our modified FasterTransformer baseline outperforms the original.

5.1. Microbenchmarks

We evaluate the DéjàVuLib streaming mechanism, using requests with a prompt size of 500 tokens, generating 500 new tokens. We measure the time to complete a batch of requests, without streaming, and with streaming to local SSD, and remote CPU. Here, we use only tensor parallelism when more than 1 GPU is employed (no pipeline parallelism). The DéjàVuLib streaming slowdown is within 2% for local SSD and remote CPU memory (Appendix F). The negligible overhead of DéjàVuLib is due to the optimizations described in 4.1. Figure 7 provides a breakdown of the performance achieved by each of the optimizations. *Baseline* stands for transferring all contiguous memory regions one by one. *Buffered Copies* (Optimization (1) in 4.1) has 95× improvement compared to baseline. The other two DéjàVuLib optimizations further improve streaming performance by 1.4×.

5.2. End-To-End Performance

5.2.1. Performance without failures

We now evaluate the performance of our disaggregated DéjàVu system compared to the non-disaggregated baseline. We configure all our requests to a fixed prompt size (1000 tokens for Figure 8), and we sample the number of newly generated tokens from the LMSys dataset (Zheng et al., 2023), assuming all requests within a microbatch generate the same number of tokens. We use one client, which submits requests following a Poisson distribution in an open loop, with varying request rates. We use microbatch size of 8 in all cases. Similarly to Orca (Yu et al., 2022) and vLLM (Kwon et al., 2023) we report normalized latency

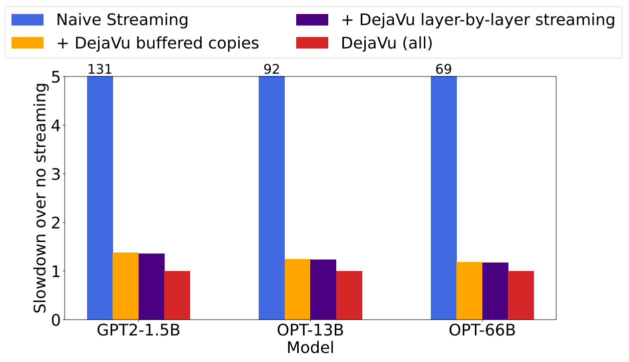


Figure 7. Single-batch latency slowdown (with batch size 8) caused by KV streaming to remote CPU memory, when gradually applying the optimizations proposed by DéjàVuLib.

(seconds/token) for each request rate. To compute the normalized latency for each request, we divide its end-to-end latency by the number of generated tokens. Figure 8 shows the median normalized latency for OPT-66B and BLOOM-175B. Since DéjàVu targets pipeline parallel configurations, we employ pipeline parallelism using multiple machines with a few GPUs each. Each pipeline stage is a VM with 2 GPUs running tensor model parallelism. The legends in Figure 8 show the pipeline parallelism depth.

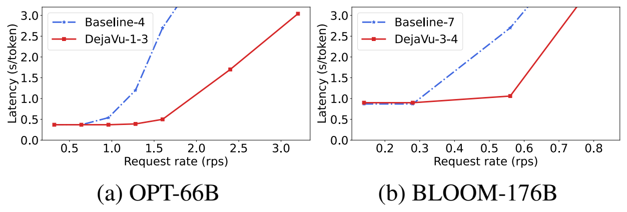


Figure 8. E2E performance of the OPT-66B and BLOOM-176B model in the LMSys dataset. Baseline-X means that X machines were used in the pipeline. DejaVu-X-Y means that X machines were used for prompt processing, and Y for token generation.

As we increase the input request rate, the normalized latency increases, due to the systems' inability to maintain that high request rate, leading to queueing effects. DéjàVu sustains low latency with up to 1.88×, and 2× higher throughput than FasterTransformer baseline for the OPT-66B and BLOOM-176B models respectively. Since microbatches generate a variable number of tokens, new prompts are being injected, introducing bubbles in the pipeline of the baseline case, where all machines are dedicated to both prompt and token processing. DéjàVu addresses this issue by allocating separate pipelines for prompt and token processing. We select the number of prompt processing and token generation pipelines as explained in section 4.2.1, to ensure that the token generation machines do not idle waiting for prompts to be processed. Disaggregation benefits are more noticeable with larger prompt sizes. Larger prompts result in extended prompt processing time, leading to larger bubbles in the baseline case, thereby downgrading performance. Despite the larger amount of data that needs to be streamed from the prompt processing to token generation machines with larger prompt sizes, DéjàVu streaming optimizations (sec 4.1) manage to fully hide the prompt streaming overhead. In appendix B we use our planner and simulator to evaluate various scenarios. Overall, DéjàVu scales better than the baseline, leading to shorter makespan and cost.

5.2.2. Performance with microbatch swapping

Swapping reduces the amount of GPU memory required for the KV cache, allowing larger batch sizes, and increasing system throughput. Figure 9 illustrates this. In this experiment, we also evaluate OPT-30B on VMs with 2 V100-16GB GPUs, and inter-VM network bandwidth of 32 Gbps. For each model and set of GPUs (x-axis), we get the achieved system throughput with the largest feasible batch size without swapping B, and the achieved throughput with swapping enabled and batch size $2 \cdot B$. By accommodating larger batch size, we increase throughput by up to 1.8×. However, the main bottleneck of swapping is the time needed to bring the KV cache back into the GPU. This depends on the size of the cache and the CPU-GPU PCIe bandwidth. Since each token generation step takes 10s-100s ms, the KV cache transferring should also be very fast. In Appendix G, we formalize the benefits of microbatch swapping and evaluate the mechanism varying the sequence length and batch size.

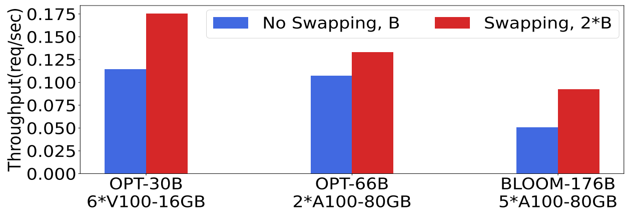


Figure 9. Benefit of microbatch swapping

5.2.3. Performance with failures

We now evaluate the performance of DéjàVu in the event of failures. We serve the OPT-66B model in a cluster of 4 machines, using pipeline parallelism with 4 stages, and microbatch size 8. Each stage in the pipeline does both prompt processing and token generation. Each request has a prompt size of 500 tokens and generates 1000 extra tokens. We incur a pipeline stage failure at token generation step 1200. Figure 10 depicts the cumulative latency of one of the active microbatches at the moment of failure. A single failure led

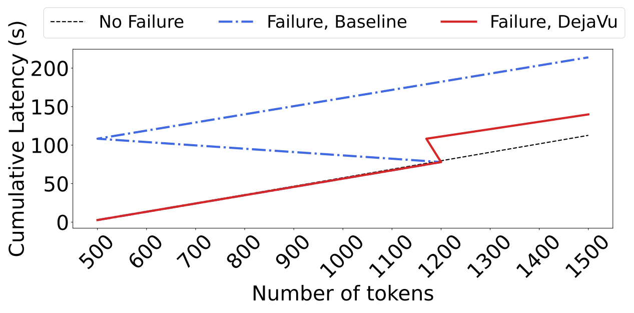


Figure 10. Effect on cumulative latency for a single microbatch when a failure occurs at token generation step 1200.

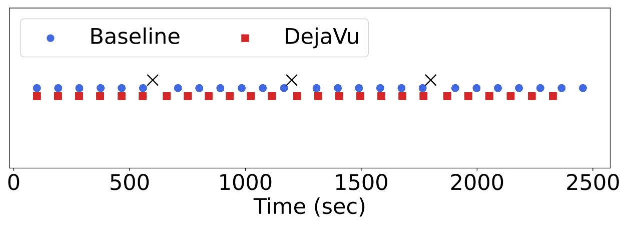


Figure 11. Request completions over time. We introduce failures after 600, 1200, and 1800 sec (marked with black X).

to 1.91× increase in the latency of a set of microbatches. In contrast, with DéjàVu the increase due to failure is 1.24×.

In Figure 11 we introduce failures at various timestamps while serving a set of requests. In the case of baseline, all workers need to restart, and the processing of the microbatches that were active at the point of failure starts from scratch. With DéjàVu, due to the lightweight cache streaming protocol, token generation just restarts from the latest replicated step, leading to 1.16× shorter runtime.

6. Related Work

Serving Systems for LLMs The widespread adoption of LLMs led to multiple LLM serving systems, such as FasterTransformer, TensorRT-LLM (NVIDIA, 2023c), and DeepSpeed Inference (Aminabadi et al., 2022). Orca (Yu et al., 2022) introduces iteration-level scheduling allowing requests at a batch to be at different phases (prompt processing or token generation), but overlooks the potential negative impact on request latency caused by the difference between prompt processing and per-token generation time. vLLM (Kwon et al., 2023) reduces KV cache overprovisioning by using dynamic memory allocation, and swapping KV cache blocks to the CPU under GPU memory pressure for individual requests. In contrast to vLLM that does not support pipeline parallelism, DéjàVu targets pipeline parallel inference and employs swapping at the level of microbatches.

Differences in prompt and token processing Some recent works, developed concurrently with DéjàVu, also address the discrepancy in prompt and token processing times. Sarathi (Agrawal et al., 2023) proposes partitioning prefill requests into smaller chunks and merging them with decodes. Splitwise (Patel et al., 2023) and DistServe (Zhong et al., 2024) propose separating prompt from token processing, to reduce power consumption and cost, by using heterogeneous GPUs for each phase independently. Splitwise is primarily simulation-based and supports execution modes with limited parallelism; models fit on a single GPU or run tensor-model parallel across GPUs, and they do no consider pipeline parallel serving (required for recent massive LLM models). DistServe employs distinct batching and parallelism for the prompt processing and token generation, based on model characteristics and simulation findings. DéjàVu uses disaggregation to minimize bubbles in pipeline parallel configurations and optimizes machine allocation for prompt and token pipelines to maximize system throughput.

LLM serving on preemptible resources SpotServe (Miao et al., 2023) is a framework for serving LLMs over spot cloud resources. SpotServe utilizes the grace period (e.g. 30 sec in AWS) before a VM is preempted, to optionally migrate KV cache contents to avoid restarting inference from scratch. However, this approach cannot protect from sudden failures which can harm request latency (see Figure 10). In contrast, DéjàVu uses a token-level KV cache replication strategy with minimal overhead, offering continuous fault tolerance and seamless recovery from any failure.

Overall, DéjàVu stands out by being comprehensive in addressing the key LLM-serving challenges we highlight in the paper; DéjàVuLib, its KV cache streaming library, is designed for high-performance and versatility to address these challenges. Unlike other systems that are myopic in their focus of a challenge, DéjàVu offers high-throughput, fault-tolerance, and seamless recovery upon failures, as well as opportunities for memory savings in LLM inference.

7. Conclusion

DéjàVu is a system for efficient and fault-tolerant LLM serving at scale. It decouples prompt processing from token generation to mitigate pipeline bubbles caused by the differences in prompt processing and per-token generation times. Additionally, it optimizes memory utilization in pipeline parallel configurations by implementing microbatch-level swapping to and from CPU memory. Finally, it employs cache replication and failure handling mechanisms to provide seamless recovery and minimize redundant work in the event of failures. DéjàVu leverages DéjàVuLib, a library that allows KV cache streaming under various setups with minimal overhead. DéjàVu improves LLM serving throughput by up to 2× compared to state-of-the-art systems.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none of which we feel must be specifically highlighted here.

Acknowledgements

We thank the anonymous reviewers for their valuable feedback. Foteini Strati was supported in part by the Swiss National Science Foundation (Project Number 200021_204620). Sara McAllister was supported in part by a NDSEG fellowship.

References

Agrawal, A., Panwar, A., Mohan, J., Kwatra, N., Gulavani, B. S., and Ramjee, R. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills, 2023.

Aminabadi, R. Y., Rajbhandari, S., Zhang, M., Awan, A. A., Li, C., Li, D., Zheng, E., Rasley, J., Smith, S., Ruwase, O., and He, Y. Deepspeed inference: Enabling efficient inference of transformer models at unprecedented scale, 2022.

Bhardwaj, A., Phanishayee, A., Narayanan, D., Tarta, M., and Stutsman, R. Packrat: Automatic reconfiguration for latency minimization in cpu-based dnn serving, 2023.

Boost. Boost.asio. https://www.boost.org/doc/libs/1_78_0/doc/html/boost_asio.html, 2021.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H. (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 1877–1901. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

Ding, J., Ma, S., Dong, L., Zhang, X., Huang, S., Wang, W., Zheng, N., and Wei, F. Longnet: Scaling transformers to 1,000,000,000 tokens, 2023.

Eisenman, A., Matam, K. K., Ingram, S., Mudigere, D., Krishnamoorthi, R., Nair, K., Smelyanskiy, M., and Annavaram, M. Check-N-Run: a checkpointing system for training deep learning recommendation models. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pp. 929–943, Renton, WA, April 2022. USENIX Association. ISBN 978-1-939133-27-4. URL https://www.usenix.org/conference/nsdi22/presentation/eisenman.

grpc. grpc: A high performance, open source universal rpc framework. https://grpc.io/, 2023.

HuggingFace. Huggingface accelerate. https://huggingface.co/docs/accelerate/en/index, 2024.

Jeon, M., Venkataraman, S., Phanishayee, A., Qian, J., Xiao, W., and Yang, F. Analysis of Large-Scale Multi-Tenant GPU clusters for DNN training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pp. 947–960, Renton, WA, July 2019. USENIX Association. ISBN 978-1-939133-03-8. URL https://www.usenix.org/conference/atc19/presentation/jeon.

Jiang, Y., Yan, R., Yao, X., Zhou, Y., Chen, B., and Yuan, B. Hexgen: Generative inference of large-scale foundation model over heterogeneous decentralized environment, 2024.

Jin, Y., Wu, C.-F., Brooks, D., and Wei, G.-Y. S^3: Increasing gpu utilization during generative inference for higher throughput, 2023.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, pp. 611–626, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702297. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

Li, Y., Phanishayee, A., Murray, D., Tarnawski, J., and Kim, N. S. Harmony: overcoming the hurdles of gpu memory capacity to train massive dnn models on commodity servers. *Proc. VLDB Endow.*, 15(11):2747–2760, jul 2022. ISSN 2150-8097. doi: 10.14778/3551793.3551828. URL https://doi.org/10.14778/3551793.3551828.

Miao, X., Shi, C., Duan, J., Xi, X., Lin, D., Cui, B., and Jia, Z. Spotserve: Serving generative large language models on preemptible instances, 2023.

Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V. A., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., Phanishayee, A., and Zaharia, M. Efficient large-scale language model training on gpu clusters using megatron-lm, 2021.

NVIDIA. Nvidia collective communications library (nccl). https://developer.nvidia.com/nccl, 2023a.

NVIDIA. Nvidia fastertransformer. https://github.com/NVIDIA/FasterTransformer, 2023b.

NVIDIA. Tensorrt-llm. https://github.com/NVIDIA/TensorRT-LLM, 2023c.

OpenMPI. Open mpi: Open source high performance computing. https://www.open-mpi.org/, 2023.

Ott, M., Edunov, S., Baevski, A., Fan, A., Gross, S., Ng, N., Grangier, D., and Auli, M. fairseq: A fast, extensible toolkit for sequence modeling. In Ammar, W., Louis, A., and Mostafazadeh, N. (eds.), *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations)*, pp. 48–53, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-4009. URL https://aclanthology.org/N19-4009.

Patel, P., Choukse, E., Zhang, C., Íñigo Goiri, Shah, A., Maleki, S., and Bianchini, R. Splitwise: Efficient generative llm inference using phase splitting, 2023.

Pope, R., Douglas, S., Chowdhery, A., Devlin, J., Bradbury, J., Levskaya, A., Heek, J., Xiao, K., Agrawal, S., and Dean, J. Efficiently scaling transformer inference. *CoRR*, abs/2211.05102, 2022. doi: 10.48550/ARXIV.2211.05102. URL https://doi.org/10.48550/arXiv.2211.05102.

Rajbhandari, S., Ruwase, O., Rasley, J., Smith, S., and He, Y. Zero-infinity: breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384421. doi: 10.1145/3458817.3476205. URL https://doi.org/10.1145/3458817.3476205.

Sheng, Y., Zheng, L., Yuan, B., Li, Z., Ryabinin, M., Fu, D. Y., Xie, Z., Chen, B., Barrett, C., Gonzalez, J. E., Liang, P., Ré, C., Stoica, I., and Zhang, C. Flexgen: High-throughput generative inference of large language models with a single gpu, 2023.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., and Lample, G. Llama: Open and efficient foundation language models, 2023.

Workshop, B. Bloom: A 176b-parameter open-access multilingual language model, 2023.

Yan, Y., Hu, F., Chen, J., Bhendawade, N., Ye, T., Gong, Y., Duan, N., Cui, D., Chi, B., and Zhang, R. Fastseq: Make sequence generation faster. In *Annual Meeting of the Association for Computational Linguistics*, 2021. URL https://api.semanticscholar.org/CorpusID:235376968.

Yu, G.-I., Jeong, J. S., Kim, G.-W., Kim, S., and Chun, B.-G. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 521–538, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/yu.

Zhang, H., Zheng, Z., Xu, S., Dai, W., Ho, Q., Liang, X., Hu, Z., Wei, J., Xie, P., and Xing, E. P. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pp. 181–193, Santa Clara, CA, July 2017. USENIX Association. ISBN 978-1-931971-38-6. URL https://www.usenix.org/conference/atc17/technical-sessions/presentation/zhang.

Zhang, S., Roller, S., Goyal, N., Artetxe, M., Chen, M., Chen, S., Dewan, C., Diab, M., Li, X., Lin, X. V., Mihaylov, T., Ott, M., Shleifer, S., Shuster, K., Simig, D., Koura, P. S., Sridhar, A., Wang, T., and Zettlemoyer, L. Opt: Open pre-trained transformer language models, 2022.

Zhang, Z., Sheng, Y., Zhou, T., Chen, T., Zheng, L., Cai, R., Song, Z., Tian, Y., Ré, C., Barrett, C., Wang, Z., and Chen, B. H_2o: Heavy-hitter oracle for efficient generative inference of large language models, 2023.

Zheng, L., Chiang, W.-L., Sheng, Y., Li, T., Zhuang, S., Wu, Z., Zhuang, Y., Li, Z., Lin, Z., Xing, E. P., Gonzalez, J. E., Stoica, I., and Zhang, H. Lmsys-chat-1m: A large-scale real-world llm conversation dataset, 2023.

Zhong, Y., Liu, S., Chen, J., Hu, J., Zhu, Y., Liu, X., Jin, X., and Zhang, H. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving, 2024.

A. Prompt processing and token generation time

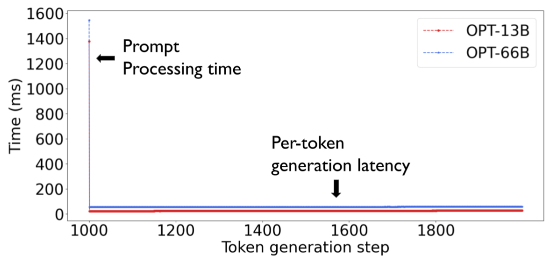


Figure 12. Prompt processing and token generation time for OPT-13B and OPT-66B serving with batch size 8

Figure 12 shows the per-token generation time when serving the OPT-13B and OPT-66B model, with prompt size 1000, and generating 1000 extra tokens. We observe that the time to generate the first token (i.e. prompt processing time) is much higher than the time to generate each subsequent token (which is nearly constant).

Figures 13, 14, and 15 show average per-token generation time, and prompt processing time with different prompt sizes and batch sizes. Prompt processing time scales almost linearly with batch size and prompt length while being up to 106× higher than per-token generation latency. We observe similar patterns in other model generations such as Llama (Touvron et al., 2023).

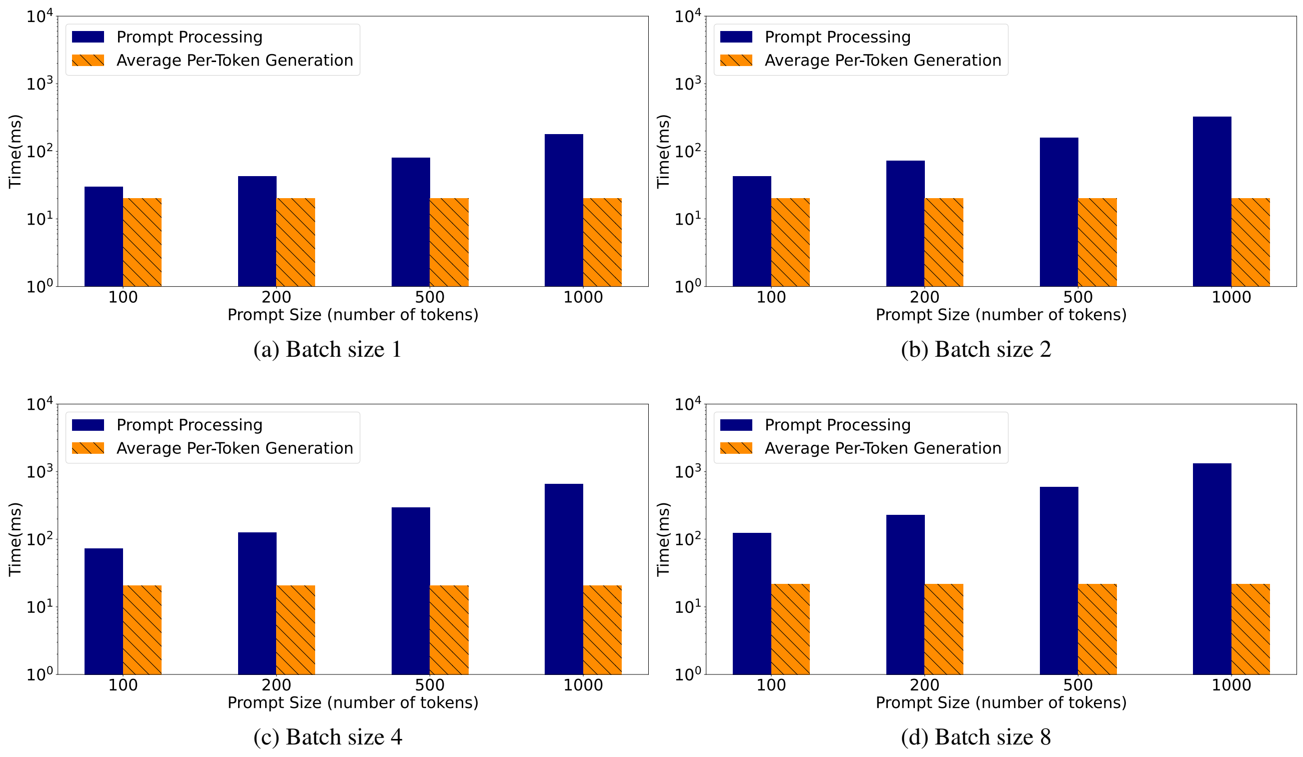
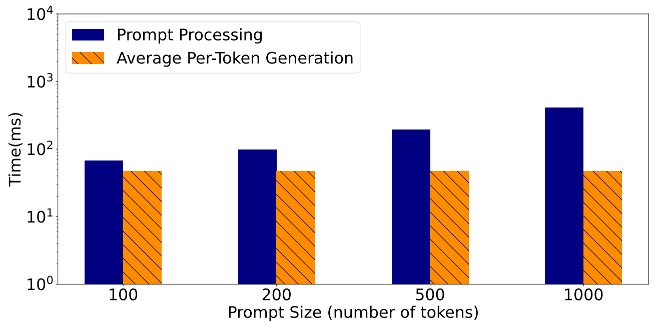
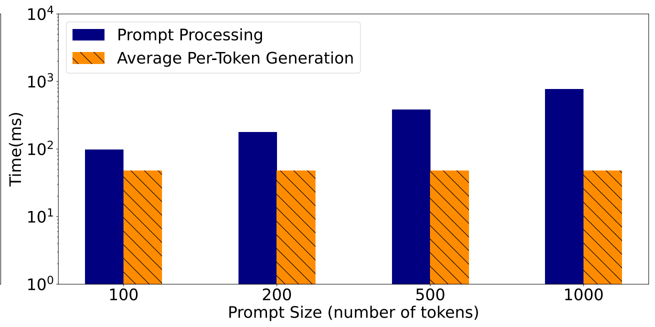


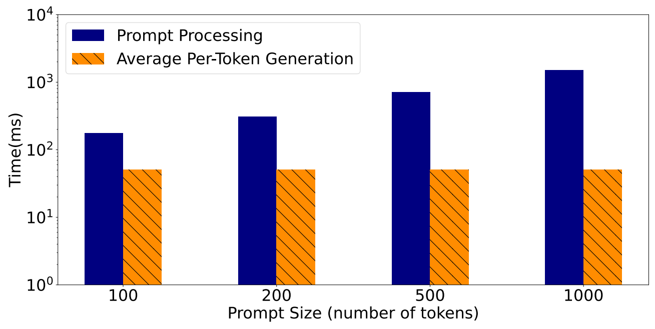
Figure 13. Prompt processing and token generation times for the OPT-13B model



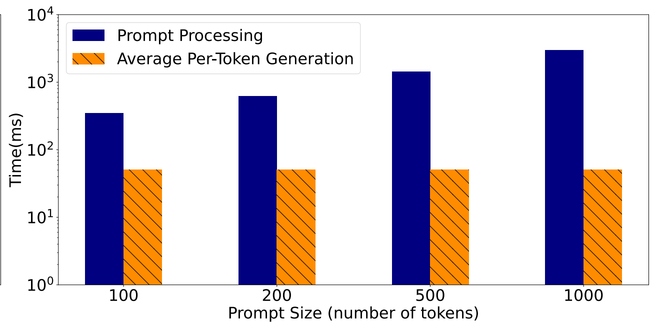
(a) Batch size 1



(b) Batch size 2

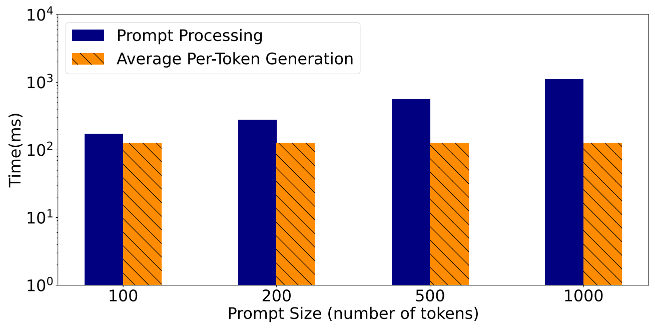


(c) Batch size 4

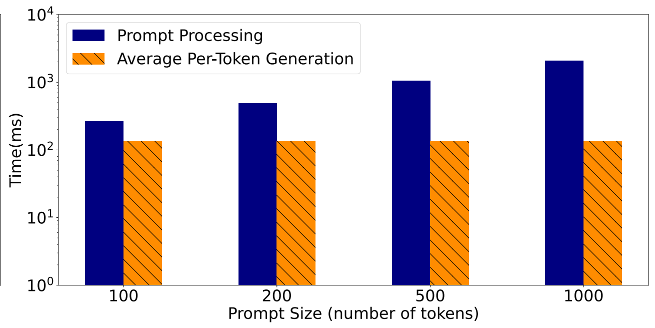


(d) Batch size 8

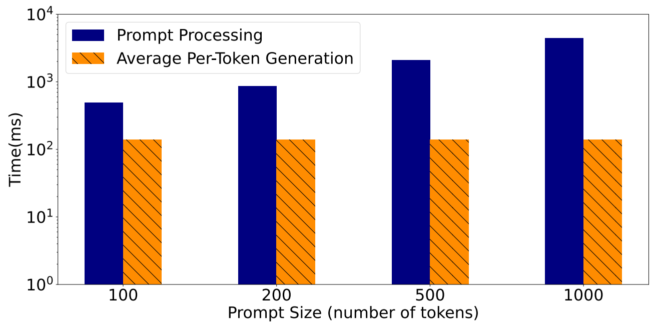
Figure 14. Prompt processing and token generation times for the OPT-66B model



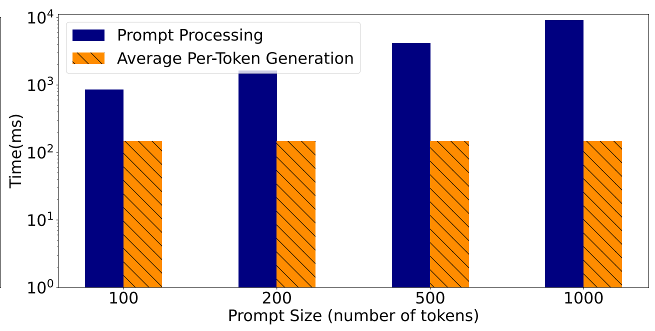
(a) Batch size 1



(b) Batch size 2



(c) Batch size 4



(d) Batch size 8

Figure 15. Prompt processing and token generation times for the BLOOM-176B model

B. DéjàVu Planner

In this section, we evaluate how disaggregation performs under various scenarios. Due to limited budget, we use our simulator to model a large number of machines. We use different types of models (OPT (Zhang et al., 2022) and BLOOM (Workshop, 2023)), different types of GPUs (V100-16GB, and A100-80GB) and different number of GPUs per machine (i.e. varying degrees of tensor parallelism). As in section 5.2.1, we use a modified version of the LMSys dataset for the number of generated tokens per microbatch (Zheng et al., 2023), and keep the prompt size equal to 1000 tokens.

Figures 16-19 show the makespan and normalized total cost (with respect to the hourly cost of a single VM) for various numbers of available machines for our trace. For D available machines, we simulate 3 different scenarios:

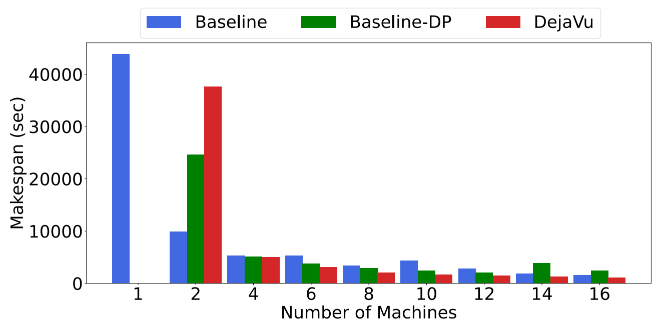
1. **Baseline (Tensor Model + Pipeline Parallel)**: all D machines run pipeline parallelism (all GPUs within each machine run Tensor Model Parallelism). We vary the microbatch size b, and pick the microbatch size that leads to the shortest makespan for our trace of requests.
2. **Baseline-DP (Tensor Model + Pipeline + Data Parallel)**: we use d pipelines, serving requests in parallel. Each pipeline has depth $\frac{D}{d}$. We also vary the microbatch size b, that each pipeline is using. We pick the combination d-b that leads to the shortest makespan.
3. **DéjàVu (Tensor Model + Pipeline Parallel)**: we use D_p machines for prompt processing (with pipeline depth D_p), and D_t machines for token generation (with pipeline depth D_t), where $D = D_p + D_t$. We also vary the microbatch size b, that each pipeline is using. We pick the combination D_p-b that leads to the shortest makespan.

Tables 2-5 show the best configurations found for each use case. For the *Baseline*, (Yp, Zb) indicates that a pipeline of depth Y was used, with microbatch size Z. For the *Baseline-DP*, (Xd, Yp, Zb) indicates that X pipelines were used, each with depth Y and microbatch size Z. For *DéjàVu*, $((Y_1 d, Z_1 b), (Y_2 d, Z_2 b))$ means that the pipeline used for prompt processing has depth Y_1 and microbatch size Z_1, and the pipeline used for token generation has depth Y_2 and microbatch size Z_2.

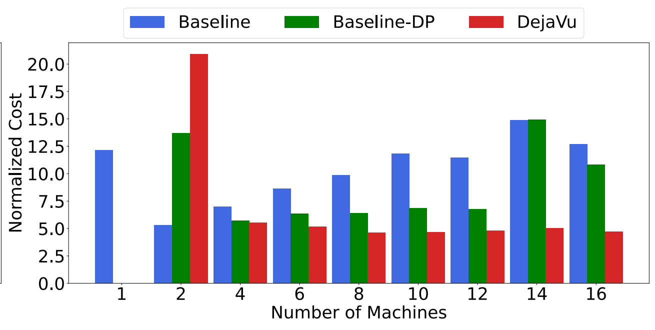
Overall, *Baseline* (i.e. using tensor model and pipeline parallelism) has a shorter makespan with a larger number of available machines (and subsequently the pipeline depth) and microbatch size (which can also be explained from formula 4). However, we observe 3 trends: First, the scalability of *Baseline* diminishes as pipeline depth increases, i.e. an Y$\times$ increase in the pipeline depth does not correspond to a Y$\times$ decrease in makespan. This becomes evident from the normalized cost Figures 16-19, where the cost increases with the number of machines. Second, with a real-world trace such as the LMSys (Zheng et al., 2023) dataset, the number of ”non-overlapping” early stops, and their impact on the pipeline depends on the trace and the pipeline depth. For example, in Figure 20, we plot the makespan for *Baseline* for the BLOOM model, with and without early stops for a fixed microbatch size. Without early stops, as we increase the number of machines, the makespan decreases. With early stops, we see sudden increases in makespan with 6,10, and 14 machines, due to a higher number of non-overlapping early stops. Third, although a larger microbatch size can reduce the makespan, it also increases proportionally the time for prompt computation (Y). However, the time per token only slightly increases (see Figures 13-15). Consequently, larger microbatch size leads to larger differences between prompt processing and token generation time, which increases the pipeline bubbles in the case of requests exiting early. Thus, a larger microbatch size is not always beneficial, as shown in Figure 21.

Introducing Data Parallelism is beneficial for performance. Overall, *Baseline-DP* outperforms *Baseline* by 2.29$\times$. However, the varying number of generated token at the LMSys trace can cause imbalance to the different data parallel pipelines, both in the total number of tokens they generate, and also in the number of early stops they encounter.

DéjàVu can alleviate all of these issues by disaggregating prompt processing from token generation, and eliminating the impact of early stops. Overall, when dedicating the same number of machines to all baselines, DéjàVu leads to 4.2$\times$ and 2.22$\times$ shorter makespan compared to *Baseline*, and *Baseline-DP* respectively. We also observe that DéjàVu's makespan speedup increases as the cluster size increases. Thus, we expect that with ever larger models and deployments, DéjàVu's performance benefits will be even more crucial.

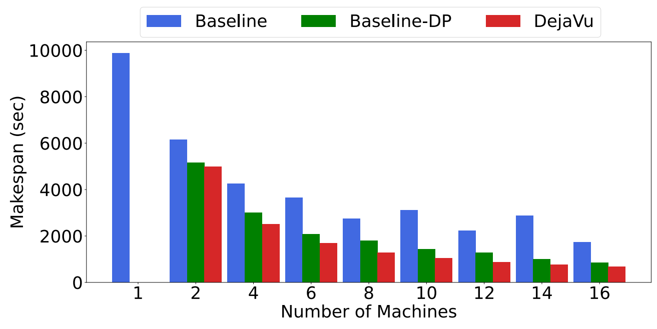


(a) Makespan

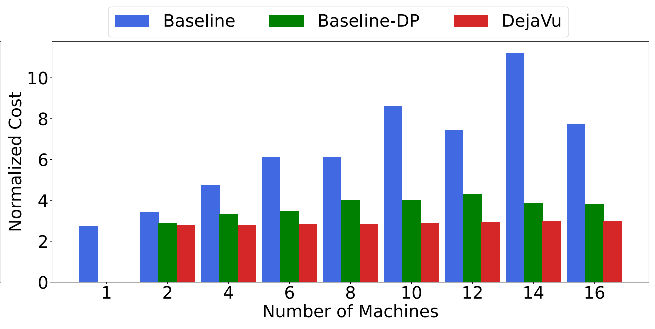


(b) Normalized Cost

Figure 16. Makespan and cost for the best configurations found with different policies for the OPT-66B model on 2-A100-80GB machines. The best configurations found for each use case are shown in table 2

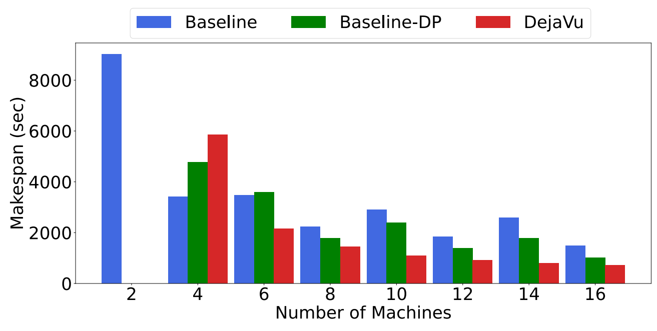


(a) Makespan

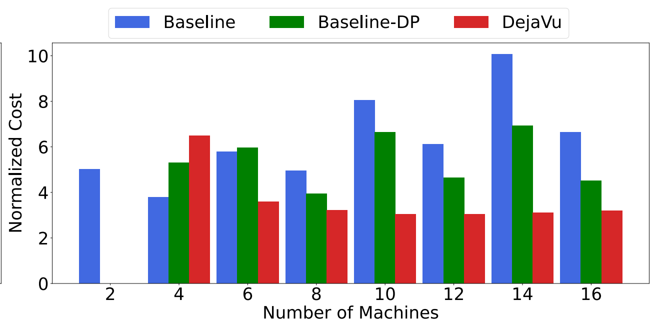


(b) Normalized Cost

Figure 17. Makespan and cost for the best configurations found with different policies for the OPT-66B model on 4-A100-80GB machines. The best configurations found for each use case are shown in table 3

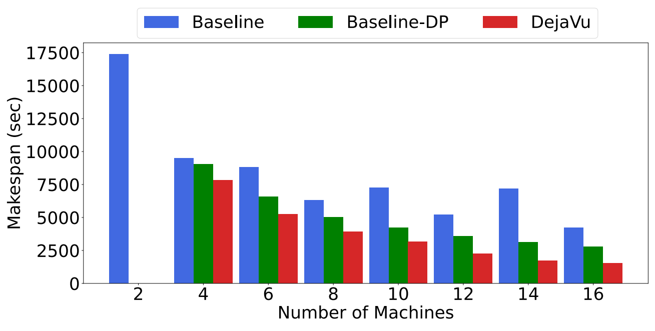


(a) Makespan

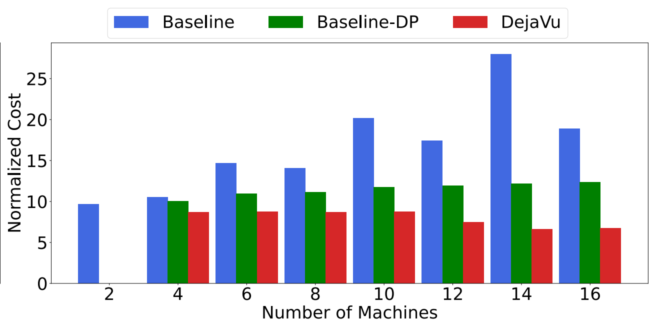


(b) Normalized Cost

Figure 18. Makespan and cost for the best configurations found with different policies for the OPT-30B model on 4-V100-16GB machines. The best configurations found for each use case are shown in table 4

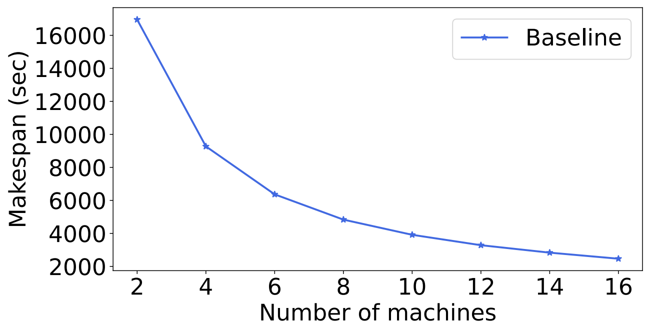


(a) Makespan

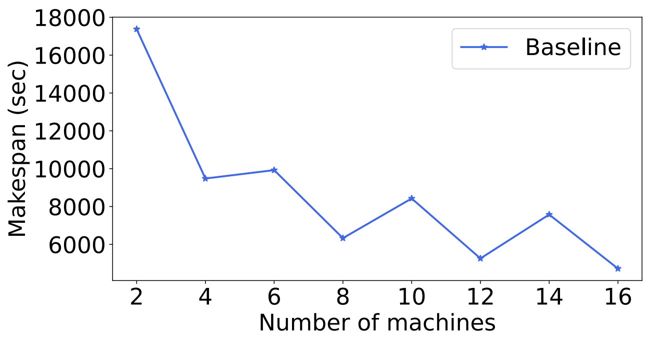


(b) Total Cost

Figure 19. Makespan and cost for the best configurations found with different policies for the BLOOM-176B model on 4-A100-80GB machines. The best configurations found for each use case are shown in table 5

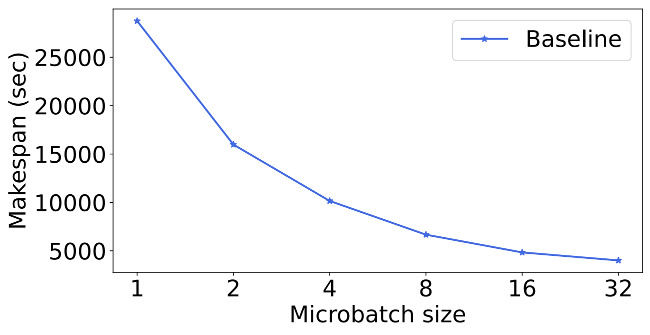


(a) Without early stopping

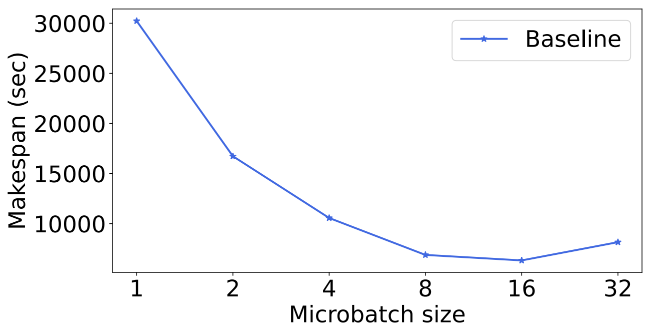


(b) With early stopping

Figure 20. Makespan for BLOOM-176B with and without early stopping, varying the number of available machines. We keep the microbatch size constant at 16 requests.



(a) Without early stopping



(b) With early stopping

Figure 21. Makespan for BLOOM-176B on 8 machines, with and without early stopping, varying the microbatch size.

Table 2. Best configurations found for Figure 16.

Number of machines	Baseline	Baseline-DP	DéjàVu
1	$(1p, 4b)$		
2	$(2p, 16b)$	$(2d, 1p, 4b)$	$((1p, 4b), (1p, 4b))$
4	$(4p, 16b)$	$(2d, 2p, 16b)$	$((2p, 2b), (2p, 2b))$
6	$(6p, 16b)$	$(3d, 2p, 16b)$	$((2p, 16b), (4p, 16b))$
8	$(8p, 16b)$	$(2d, 4p, 16b)$	$((3p, 16b), (5p, 16b))$
10	$(10p, 16b)$	$(5d, 2p, 16b)$	$((4p, 16b), (6p, 16b))$
12	$(12p, 16b)$	$(3d, 4p, 16b)$	$((5p, 16b), (7p, 16b))$
14	$(14p, 16b)$	$(7d, 2p, 16b)$	$((5p, 16b), (9p, 16b))$
16	$(16p, 8b)$	$(4d, 4p, 16b)$	$((6p, 16b), (10p, 16b))$

Table 3. Best configurations found for Figure 17.

Number of machines	Baseline	Baseline-DP	DéjàVu
1	$(1p, 32b)$		
2	$(2p, 32b)$	$(2d, 1p, 32b)$	$((1p, 32b), (1p, 32b))$
4	$(4p, 16b)$	$(4d, 1p, 32b)$	$((2p, 32b), (2p, 32b))$
6	$(6p, 32b)$	$(6d, 1p, 32b)$	$((3p, 32b), (3p, 32b))$
8	$(8p, 16b)$	$(8d, 1p, 32b)$	$((4p, 32b), (4p, 32b))$
10	$(10p, 32b)$	$(10d, 1p, 32b)$	$((5p, 32b), (5p, 32b))$
12	$(12p, 16b)$	$(12d, 1p, 32b)$	$((6p, 32b), (6p, 32b))$
14	$(14p, 32b)$	$(14d, 1p, 32b)$	$((7p, 32b), (7p, 32b))$
16	$(16p, 8b)$	$(16d, 1p, 32b)$	$((8p, 32b), (8p, 32b))$

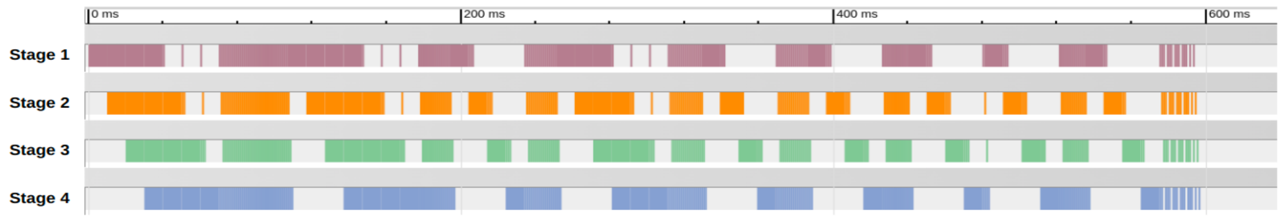
Table 4. Best configurations found for Figure 18.

Number of machines	Baseline	Baseline-DP	DéjàVu
2	$(2p, 8b)$		
4	$(4p, 16b)$	$(2d, 2p, 8b)$	$((2p, 8b), (2p, 8b))$
6	$(6p, 16b)$	$(3d, 2p, 8b)$	$((2p, 16b), (4p, 16b))$
8	$(8p, 16b)$	$(2d, 4p, 16b)$	$((3p, 16b), (5p, 16b))$
10	$(10p, 16b)$	$(5d, 2p, 8b)$	$((4p, 16b), (6p, 16b))$
12	$(12p, 16b)$	$(3d, 4p, 16b)$	$((5p, 16b), (7p, 16b))$
14	$(14p, 16b)$	$(7d, 2p, 8b)$	$((6p, 16b), (8p, 16b))$
16	$(16p, 8b)$	$(4d, 4p, 16b)$	$((7p, 16b), (9p, 16b))$

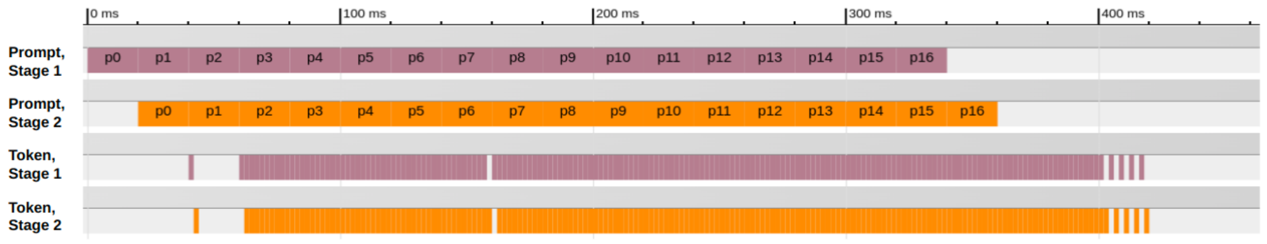
Table 5. Best configurations found for Figure 19.

Number of machines	Baseline	Baseline-DP	DéjàVu
2	$(2p, 16b)$		
4	$(4p, 16b)$	$(2d, 2p, 16b)$	$((2p, 16b), (2p, 16b))$
6	$(6p, 32b)$	$(3d, 2p, 16b)$	$((3p, 16b), (3p, 16b))$
8	$(8p, 16b)$	$(2d, 4p, 16b)$	$((4p, 16b), (4p, 16b))$
10	$(10p, 32b)$	$(5d, 2p, 16b)$	$((5p, 16b), (5p, 16b))$
12	$(12p, 16b)$	$(6d, 2p, 16b)$	$((6p, 32b), (6p, 32b))$
14	$(14p, 32b)$	$(7d, 2p, 16b)$	$((8p, 32b), (6p, 32b))$
16	$(16p, 8b)$	$(4d, 4p, 16b)$	$((10p, 32b), (6p, 32b))$

C. Traces of disaggregation



(a) Prompt processing and token generation in the same pipeline.



(b) Using a different pipeline for prompt processing and token generation.

Figure 22. Comparison between dedicating 4 machines to both prompt processing and token generation, vs using 2 machines for prompt processing and 2 machines for token generation. Prompt processing latency is 10 $\times$ higher than per-token generation latency.

D. Principled allocation of resources for prompt processing and token generation

Assume we are given D machines, each with aggregate GPU memory capacity of M GB. Assume a model has L layers. For simplification, we consider only the memory requirements of the attention layers W_i. We also assume each layer's prompt KV cache footprint is C_i. The requirements that we need to satisfy are: 1) the aggregate memory footprint (model parameters and KV cache), for the active microbatches should fit into the aggregate GPU memory capacity for each pipeline, and 2) the throughput of the disaggregated system should be maximized, and ideally be higher than the throughput of the non-disaggregated system.

We first aim to satisfy requirement (1) for the prompt processing pipeline, i.e. find the prompt pipeline depth D_p. P_n is the number of attention layers per stage. Assuming each machine corresponds to a pipeline stage, the following inequality should hold:

$$M \geq P_n \cdot (C_0 + W_0) \implies P_n \leq \lfloor \frac{M}{C_0 + W_0} \rfloor$$

Since $D_p = \lceil \frac{L}{P_n} \rceil$:

$$D_p \geq \lceil \frac{L \cdot (C_0 + W_0)}{M} \rceil \tag{2}$$

Similarly, we need to satisfy requirement (1) for the token generation pipeline, i.e. find the token generation depth D_t. Each layer's token KV cache footprint is K_i. T_n is the number of attention layers per stage. Since token generation involves multiple steps, and at least D_t microbatches need to be on-the-fly at any given step, we have:

$$M \geq T_n \cdot W_0 + D_t \cdot (C_i + K_i) \implies M \geq T_n \cdot (W_0 + (C_0 + K_0) \cdot D_t)$$

Since $D_t = \lceil \frac{L}{T_n} \rceil$:

$$M \geq T_n \cdot (W_0 + (C_0 + K_0) \cdot \lceil \frac{L}{T_n} \rceil)$$

For simplicity, we assume T_n divides L:

$$M \geq T_n \cdot W_0 + L \cdot (C_0 + K_0) \Longrightarrow M \geq \frac{L}{D_t} \cdot W_0 + L \cdot (C_0 + K_0)$$

$$D_t \geq \frac{L \cdot W_0}{M - L \cdot (C_0 + K_0)} \tag{3}$$

For requirement (2), we need to compute the throughput of the non-disaggragated and the disaggregated setups. For simplicity, in the following formulas, we work with the *inverse throughput* of the pipelines. Thus, we would like the disaggregated case to have *lower* inverse throughput than the non-disaggregated one. Assume, for simplicity, that prompt processing of each microbatch with D machines lasts Y ms, and each token generation step for a single microbatch takes t ms. Since we dedicate D_p machines to prompt processing and D_t machines to token generation, each machine will host a larger number of layers. Thus, $Y_{dis} = \frac{D}{D_p} \cdot Y$, and $t_{dis} = \frac{D}{D_t} \cdot t$. Assume, also, N new tokens are generated per microbatch.

First, we compute the inverse throughput (I) of the baseline. Figure 23 illustrates a toy example of a pipeline with 3 stages. In the general case of a pipeline with D stages, and D active microbatches at each point in time, the inverse throughput is given by:

$$I_c = \frac{S_1 + S_2 + S_3}{D}$$

where:

$$S_1 = D \cdot Y$$

$$S_2 = (D-1) \cdot Y$$

$$S_3 = (N-1) \cdot D \cdot t + t$$

Thus:

$$I_c = \frac{D \cdot Y + (D-1) \cdot Y + (N-1) \cdot D \cdot t + t}{D}$$

$$I_c = \frac{D \cdot Y + (D-1) \cdot Y + N \cdot D \cdot t - D \cdot t + t}{D}$$

$$I_c = \frac{D \cdot Y + (D-1) \cdot Y + N \cdot D \cdot t - (D-1) \cdot t}{D}$$

$$I_c = Y + N \cdot t + \frac{(D-1)(Y-t)}{D}$$

$$I_c = \frac{(D-1)(Y-t)}{D} + Y + N \cdot t \tag{4}$$

In steady case, the token generation pipeline with D_t machines will have inverse throughput:

$$I_t = \frac{N \cdot D_t \cdot t_{dis}}{D_t} = N \cdot t_{dis} = \frac{N \cdot D \cdot t}{D_t}$$

The prompt generation pipeline with D_p stages will have inverse throughput:

$$I_p = \frac{m \cdot Y_{dis} \cdot D_p}{D_p} = m \cdot Y_{dis} = \frac{m \cdot D \cdot Y}{D_p}$$

where m is the additional overhead due to cache streaming (i.e. $m \geq 1$).

The performance of the disaggregated system (I_{dis}) depends on the performance of the prompt processing and token generation pipelines, i.e. $I_{dis} = max(I_t, I_p)$. Since we have D machines, and we partition them into the 2 phases, allocating more machines to prompt processing, i.e. decreasing its inverse throughput, would lead to fewer machines for token generation, i.e. increasing its inverse throughput. Since we are minimizing a $\max$ function, the ideal case will be when $I_t = I_p$. Thus, we want:

$$I_{dis} = I_t = I_p < I_c$$

From $I_t = I_p$ (and the fact that $D_t + D_p = D$), we get that:

$$\frac{N \cdot D \cdot t}{D_t} = \frac{m \cdot D \cdot Y}{D_p} \implies D_t = \frac{D \cdot N \cdot t}{m \cdot Y + N \cdot t}$$

Given this D_t, the throughput of the disaggregated system will be higher than the throughput of the non-disaggragated system if

$$I_{dis} = I_t < I_c \implies \frac{Y}{t} > \frac{D-1}{D \cdot (2-m) - 1} \tag{5}$$

Eq. 5 holds if $m \in [1, 2)$. If $m \in [1, 2)$, we have:

$$D_t = \frac{D \cdot N \cdot t}{m \cdot Y + N \cdot t} \tag{6}$$

and

$$D_p = D - D_t = \frac{D \cdot m \cdot Y}{m \cdot Y + N \cdot t} \tag{7}$$

Formulas 5, 6 and 7 lead to a couple of observations. First, as expected, given D, Y, and t, with $Y > t$, the benefits of disaggregation depend on the overheads of the prompt KV cache streaming. If the streaming overhead is too high (i.e. $m \geq 2$), there will be no benefits from disaggregation. DéjàVuLib employs multiple optimizations to ensure that the prompt KV cache streaming overhead is minimized. Second, the larger N is, i.e. a lot of new tokens are generated, D_t is increasing, i.e. we need to dedicate more machines to token generation. In contrast, when $\frac{Y}{t}$ increases, D_p is increasing, thus more machines need to be dedicated for prompt processing. Moreover, as $\frac{Y}{t}$ increases, i.e. with larger prompts, the disaggregated setup becomes more beneficial, as can be seen from inequality 5.

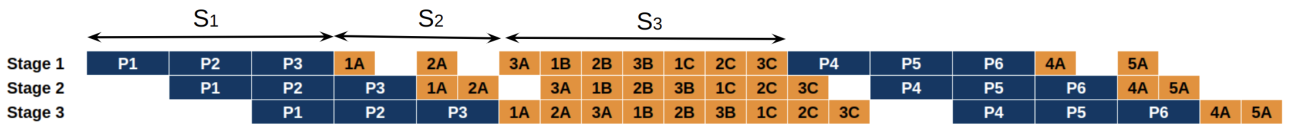
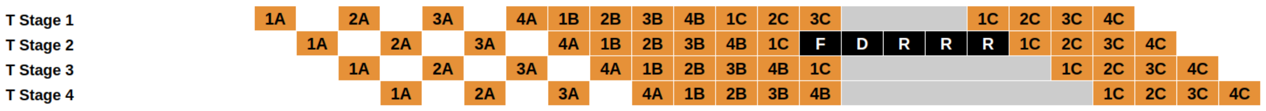

S1
S2
S3
Stage 1
Stage 2
Stage 3
P1 P2 P3 1A 2A 3A 1B 2B 3B 1C 2C 3C P4 P5 P6 4A 5A
P1 P2 P3 1A 2A 3A 1B 2B 3B 1C 2C 3C P4 P5 P6 4A 5A
P1 P2 P3 1A 2A 3A 1B 2B 3B 1C 2C 3C P4 P5 P6 4A 5A

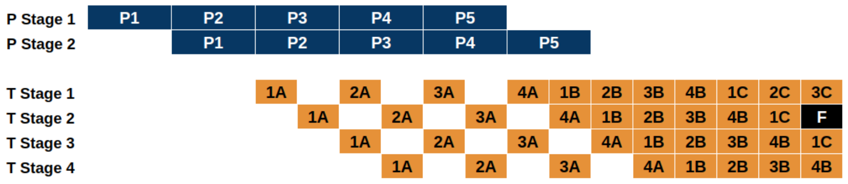
Figure 23. Illustration of the different phases of a 3-stage pipeline with prompt processing and token generation

E. Fault tolerance example

Figure 24 provides a toy example with 4 token generation workers, depicting our fault-tolerance mechanism. Refer to section 4.2.3 for a description of the fault-tolerance protocol (with references to this example).



(a) A failure occurs (F)

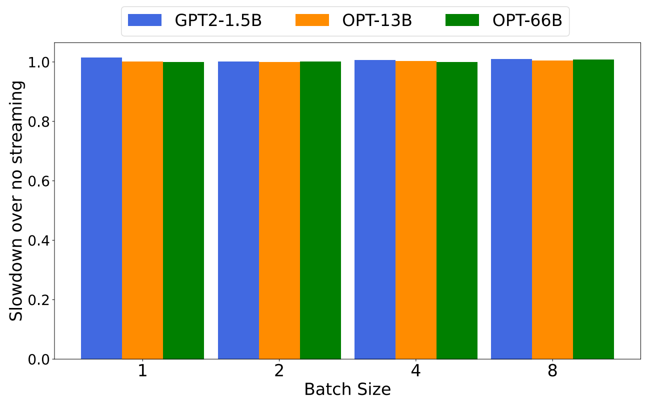


(b) The failure is detected and repaired

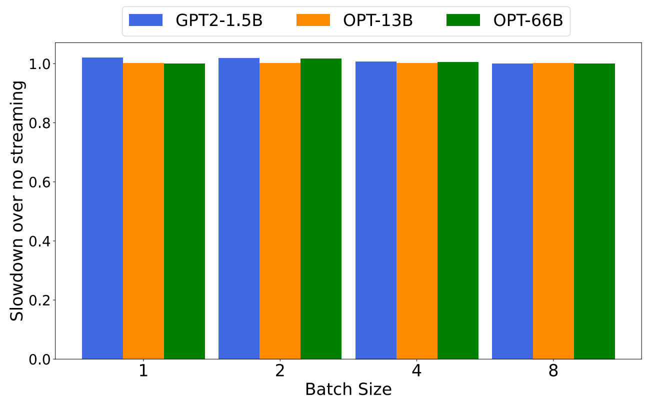
Figure 24. Example of a pipeline failure and recovery. Token stage 2 fails (F), and is detected by the DéjàVu Controller (D). The pipeline is repaired (R), which includes copying the KV caches around appropriately. After repair is done, inference continues.

F. Microbenchmarks

Figure 25 shows the slowdown of DéjàVuLib when streaming to remote CPU memory for a single batch (i.e. no pipeline parallelism) that contains requests with prompt size 500, and generating 500 extra tokens. The slowdown compared to no streaming is always within 2%. DéjàVuLib might cause some slowdown in prompt processing time, due to streaming that cannot be hidden without pipeline parallelism. However, since many tokens are generated, this slowdown is negligible.



(a) DéjàVu slowdown when streaming to remote CPU



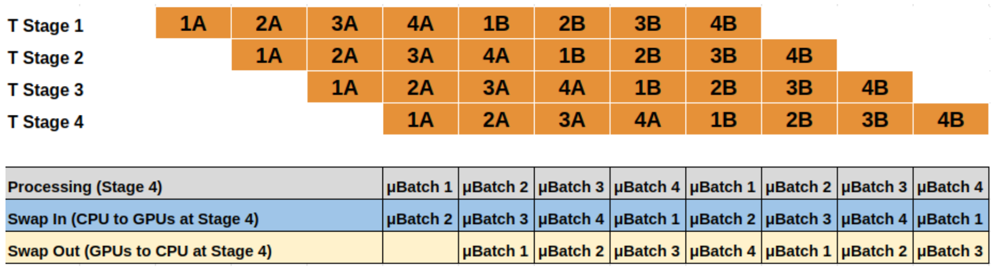
(b) DéjàVu slowdown when streaming to local SSD

Figure 25. Single-batch slowdown of DéjàVu KV cache streaming

G. Microbatch swapping

Figure 26 shows KV cache swapping on a microbatch level, for a pipeline of 4 stages, focusing on Stage 4. Whenever a microbatch is processed, we make sure its KV cache resides in GPU memory, while, in parallel, swapping other microbatches in and out of GPU. When Stage 4 generates a token for microbatch 1 (e.g. step $T1_1$), it swaps in the KV cache for the next microbatch to be processed, i.e. microbatch 2. When the processing of microbatch 1 has finished, the newly added contents to the KV cache of microbatch 1 are swapped out of the GPU. The same procedure follows for all microbatches: assuming a pipeline with N stages, when microbatch x is processed, microbatch $(x + 1)\%N$ is swapped in, and microbatch

$(x-1)\%N$ is swapped out.



T Stage 1	1A	2A	3A	4A	1B	2B	3B	4B			
T Stage 2		1A	2A	3A	4A	1B	2B	3B	4B		
T Stage 3			1A	2A	3A	4A	1B	2B	3B	4B	
T Stage 4				1A	2A	3A	4A	1B	2B	3B	4B

Processing (Stage 4)	μBatch 1	μBatch 2	μBatch 3	μBatch 4	μBatch 1	μBatch 2	μBatch 3	μBatch 4
Swap In (CPU to GPUs at Stage 4)	μBatch 2	μBatch 3	μBatch 4	μBatch 1	μBatch 2	μBatch 3	μBatch 4	μBatch 1
Swap Out (GPUs to CPU at Stage 4)		μBatch 1	μBatch 2	μBatch 3	μBatch 4	μBatch 1	μBatch 2	μBatch 3

Figure 26. Microbatch KV cache swapping over time for a 4-stage token generation pipeline. We show microbatch swapping for Stage 4, but all other stages follow similar pattern.

G.1. Understanding the benefits of microbatch swapping

The performance of microbatch swapping heavily depends on the time needed to swap the KV cache back in the GPU. In this section, we formalize the performance of pipeline parallel inference with and without microbatch swapping and investigate where it makes sense to use swapping.

Since the amount of GPU memory needed for the KV cache, when the microbatch swapping optimization is enabled, is smaller than without swapping, we can use larger batch sizes, which are beneficial for the inference throughput. Assume a case where with a given set of GPUs, we can fit a maximum microbatch size B without swapping, and microbatch size $2 \cdot B$ with swapping. We also assume that the time for token generation t is constant with both microbatch sizes (which has been validated experimentally). The time for prompt processing with microbatch size B is P, while for microbatch size $2 \cdot B$ is $2 \cdot P$.

Microbatch swapping will lead to higher throughput when:

$$2 \cdot (P + N \cdot t) \geq 2 \cdot P + \sum_{i=p}^{i=N} \max(t, transf_i)$$

$$2 \cdot N \cdot t \geq \sum_{i=p}^{i=N+p} \max(t, transf_i)$$

where $transf_i$ stands for the time needed to transfer the KV cache back in GPU from host memory, and N is the number of generated tokens. We have that $transf_i = \frac{i \cdot B \cdot C_i}{pciebw}$, where C_i is the single-token, single-request KV cache size, and $pciebw$ is the PCIe bandwidth. Thus, we have that:

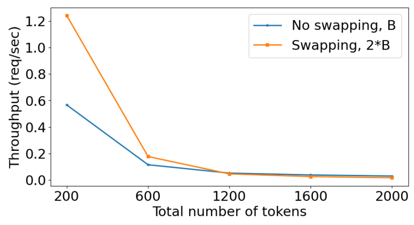
$$2 \cdot N \cdot t \geq \sum_{i=p}^{i=N+p} \max(t, \frac{i \cdot B \cdot C_i}{pciebw})$$

G.2. Experimental Analysis

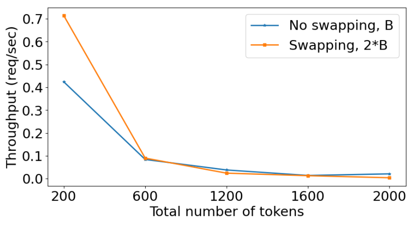
For a given model, the main factors that affect the performance of microbatch swapping are the batch size and the number of generated tokens. For our analysis, we use OPT-30B, OPT-66B, BLOOM-176B, and 2 types of GPUs: V100-16GB, and A100-80GB. As expected, with larger batch sizes or sequence lengths, the overhead of streaming the KV cache becomes high, and swapping is not beneficial anymore.

G.2.1. Constant batch size, vary the number of generated tokens

In Figure 27, We keep the batch size constant, vary the sequence length, and measure the throughput in each setup.



(a) OPT-30B, 6 V100-16GB, B=1

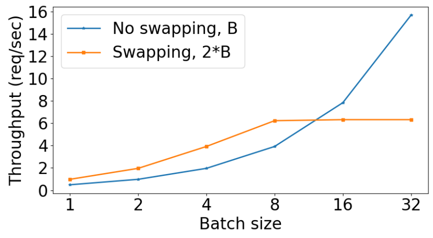


(b) OPT-66B, 2 A100-80GB, B=1

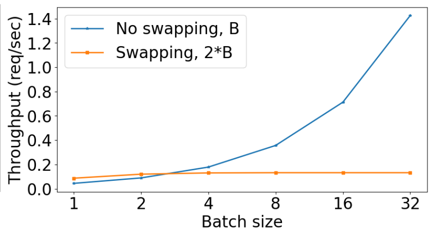
Figure 27. Throughput with and without swapping, varying the sequence length

G.2.2. Constant number of generated tokens, vary batch size

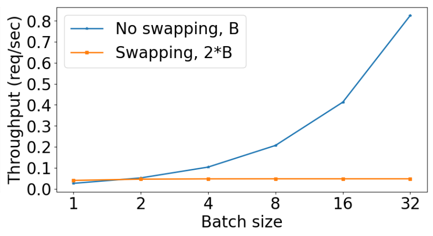
Figures 28, 29, and 30 show results in simulation, where we keep the total number of generated tokens constant, and vary the batch size (in these figures, N is the number of generated tokens). We want to see whether in cases where the maximum batch size that can fit without swapping is B, having batch size $2 \cdot B$, with swapping, provides any performance benefits.



(a) N=200

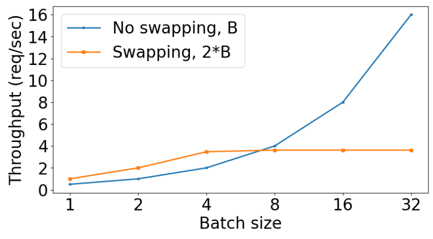


(b) N=1200

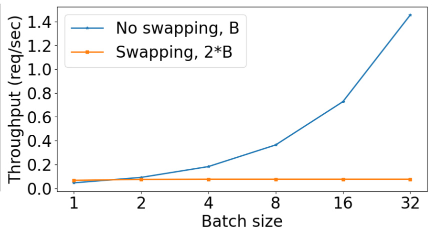


(c) N=2000

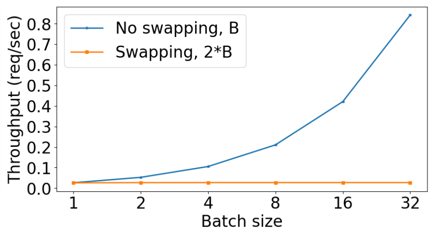
Figure 28. Throughput for OPT-30B, 6 V100-16GB



(a) N=200

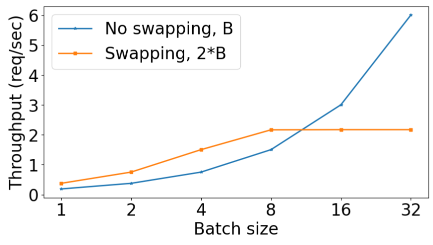


(b) N=1200

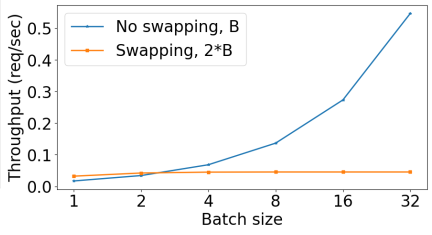


(c) N=2000

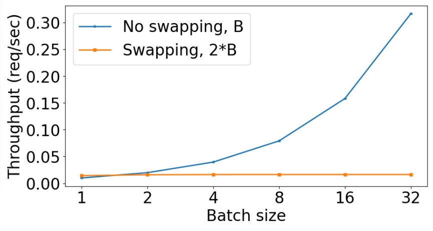
Figure 29. Throughput for OPT-66B, 2 A100-80GB



(a) N=200



(b) N=1200



(c) N=2000

Figure 30. Throughput for BLOOM-176B, 5 A100-80GB

H. Discussion

H.1. Comparison with original FasterTransformer

We modified FasterTransformer to allow for more flexible scheduling of microbatches, i.e. whenever a microbatch is completed in any stage, it can be replaced by the next available microbatch. In Figure 31 we compare our modified FasterTransformer version with the original version, highlighting that our modified FasterTransformer version outperforms the original. We use this modified version of FasterTransformer as a baseline in our paper for comparison with DéjàVu.

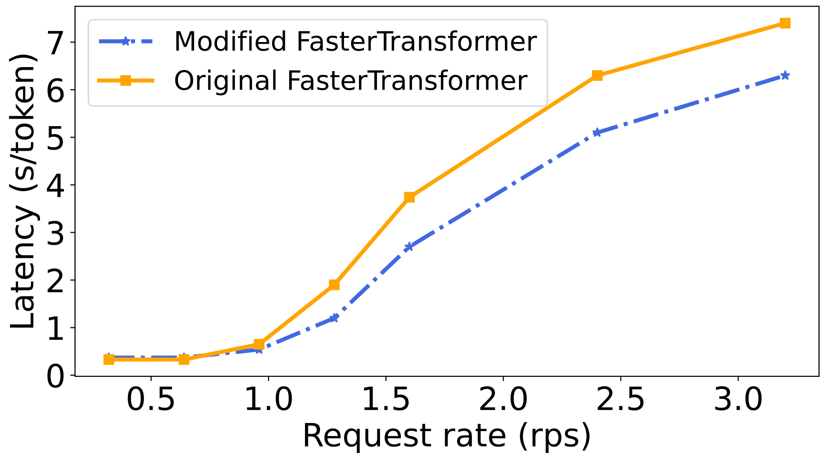


Figure 31. Comparison between original and modified FasterTransformer for the OPT-66B model.

H.2. Relation to PagedAttention and other swapping techniques

Recently, vLLM (Kwon et al., 2023) proposed *Paged Attention*, a mechanism to dynamically reserve GPU memory as more tokens are generated. Paged Attention has helped alleviate memory capacity pressure due to the KV cache in non-pipeline-parallel configurations, by reserving only the necessary amount of KV cache in GPU, organizing memory in blocks, and swapping blocks to CPU memory under GPU memory pressure.

In this work, we identify another source of memory overprovisioning in pipeline parallel setups. As explained in section 2.2.2, current frameworks that support pipeline parallel LLM inference such as FasterTransformer (NVIDIA, 2023b) preallocate the KV caches for all microbatches in GPU memory. However, only one microbatch is processed at a time, thus a large part of GPU memory remains unused. To alleviate this issue and enable inference with limited hardware, we proposed using our DéjàVuLib streaming mechanism to swap the KV cache in and out of the GPU. Compared to paged attention and vLLM, since vLLM does not currently support pipeline parallelism, it is unclear how paged attention is applied to pipeline parallel cinfigurations, and whether it resolves the memory overprovisioning issue we identify in this work. We plan to combine our swapping (and DéjàVuLib streaming mechanism) with paged attention to enable even further memory savings.

It is also important to acknowledge that swapping to CPU memory (or disk) has been widely used in DNN training to facilitate training under limited GPU memory (Li et al., 2022; Rajbhandari et al., 2021).

H.3. Discussion about FasterTransformer, and comparison with other baselines

We chose FasterTransformer as our baseline since our exploratory evaluations showed that it was the most performant system out of the systems that supported pipeline parallelism ((Aminabadi et al., 2022) and (HuggingFace, 2024)) when we started working on this project.

TensorRT-LLM (NVIDIA, 2023c) is a newly released successor to FasterTranformer (NVIDIA, 2023b). We observe that the same issues we identify in the FasterTransofrmer framework also apply to TensorRT-LLM. For example, we still see a very large difference in prompt processing and token generation time (14× in an 2-A100-80GB machine for OPT-66B with batch size 2), while there is still no fault-tolerance support.

Given the close relationship between FasterTransformer and TensorRT-LLM, we are exploring the implementation of DéjàVu (disaggregation, state replication, microbatch swapping) on top of TensorRT-LLM to address these issues. Since it already supports pipeline parallelism, TensorRT-LLM serves as a stronger foundation (compared to others that do not support pipeline parallelism). However, TensorRT-LLM's KV cache management source code is not open-sourced (yet) and hence such an integration will take longer than what one might expect from integrating with fully open-source implementations.

H.4. Effects of network bandwidth on disaggregation

Although disaggregation can reduce pipeline bubbles, increasing GPU utilization, KV cache transferring between the prompt processing and token generation workers can easily become a bottleneck, especially in slow networks. DéjàVu takes this into account, by 1) optimizing the cache streaming time, to exchange the KV cache as fast as possible, and 2) by incorporating the cache streaming time in the resource planner's decision mechanism (m in formula 5 is the cache streaming overhead). Our planner can make informed decisions and determine the cases where disaggregation would be beneficial or not, depending on the available network bandwidth and compute time. Naturally, the lower the network bandwidth, the larger the streaming overhead, so disaggregation would not be beneficial (see formula 5).

To demonstrate how network bandwidth affects request latency, in table 6, we simulate the p50 latency when serving BLOOM-176B in a cluster of 7 machines (3 used for prompt processing, and 4 for token generation) with microbatch size 8. We issue requests with a Poisson distribution. All requests have a prompt size of 1000 tokens, and generate a different number of tokens, following the LMSys dataset (Zheng et al., 2023).

We vary the network bandwidth, which affects the time needed for cache transferring. Since DéjàVuLib performs a range of optimizations to hide the streaming overheads as much as possible, we define as *extra time*, the streaming time of the KV cache which cannot be hidden by computation. From table 6, we observe that network bandwidth up to 20 Gbps is enough to sustain low latency in this setup. However, with lower network bandwidths, the cache streaming overhead cannot be tolerated. This can be seen from formula 5, where for $m > 2$ ($m = \frac{Cache\ Transfer\ Time}{Prompt\ time}$), the non-disaggregated baseline is better than DéjàVu. Also, as batch size increases, both the prompt computation time and KV cache transfer time increase proportionally, thus we expect similar trends.

Table 6. Effects of network bandwidth in the performance of a disaggregated DéjàVu inference setup (simulation-based). We serve BLOOM-176B in a cluster of 7 machines (3 used for prompt processing, and 4 for token generation). The KV cache per microbatch is 10.7 GB.

BW (Gbps)	Transfer time (sec)	Prompt time (sec)	Extra time (sec)	Latency/Token (sec)
100	0.856	3.1	0	1.29
80	1.07	3.1	0	1.29
60	1.43	3.1	0	1.29
40	2.14	3.1	0	1.29
20	4.28	3.1	1.18	1.35
10	8.56	3.1	5.46	1.5
1	85.6	3.1	82.5	3.65

H.5. Dynamic reconfiguration

So far, we have considered a *static* resource allocation planning setup, where the planner defines a plan once (e.g. amount of workers used for prompt processing or token generation), which is followed throughout model serving. An interesting extension would be to dynamically adjust the number of machines and parallelism degrees for prompt and token generation phases, based on the average input and output lengths of requests over time, and average rps. We now propose a way to adapt DéjàVu to work on this dynamic environment:

First, we would need to integrate a monitoring component that detects workload changes (changes in burstiness of requests, average prompt sizes, and number of generated tokens - while avoiding overreacting to minor temporal blips; detecting "stable" changes in workloads is a well-studied area in web services). Second, DéjàVu 's planner would have to be invoked upon such "stable" workload changes, to adjust the number of machines allocated for each phase. The time required for the planner to come up with an optimal allocation plan is negligible, so this will not add any overheads. Third, we will need necessary orchestration infrastructure in the worker processes to be able to adapt configuration on-the-fly, i.e. transitioning between prompt processing to token generation workers, and repartitioning the model if needed, based on the new degrees of parallelism. Here, reconfiguring from a prior state to a new state for each pipeline, can be done after flushing active requests, or in the case of aggressive reconfigurations can be done without flushing the pipelines (in which case, DéjàVu 's KV cache streaming primitives can be used to re-partition the cache along with model weights). Such reconfiguration can also be performed in a rolling fashion as done in some serving systems (for non-generative workloads) such as Packrat (Bhardwaj et al., 2023).