Efficient LLM Inference with Activation Checkpointing and Hybrid Caching

Sanghyeon Lee   Hongbeen Kim   Soojin Hwang   Guseul Heo   Minwoo Noh   Jaehyuk Huh

KAIST

Abstract

Recent large language models (LLMs) with enormous model sizes use many GPUs to meet memory capacity requirement incurring substantial costs for token generation. To provide cost-effective LLM inference with relaxed latency constraints, extensive research has focused on expanding GPU memory by leveraging the host memory. However, LLM inference engines that utilize the host memory often face underutilization of GPU compute units, as a considerable portion of inference time is spent in loading the model onto the GPU via host-GPU interconnect. To tackle these challenges of the host memory offloading for LLM, we introduce HybridServe, an LLM inference system with activation checkpointing based on activation caching. The activation cache stores activation checkpoints generated during intermediate inference stages, allowing the fast recomputation of KV cache while model parameters are transferred to GPU from host memory. Unlike conventional methods that recompute the KV cache from scratch using token IDs, the activation cache allows bypassing projection and FFN operations. To balance between the activation recomputation and parameter loading overhead, this study proposes a KV-activation hybrid caching scheme which finds the best ratio of the key-value and activation caches to adjust the recomputation time. Our system achieves 2.19× throughput improvement over the state-of-the-art prior work for offloading both model weights and KV cache.

1 Introduction

Large language models (LLMs) based on the transformer decoder architecture have become prevalent [4, 26, 27, 33]. However, their model sizes have been precipitously increasing, requiring many GPUs to accommodate the weights in the expensive high-bandwidth memory. In addition, the unique characteristic of LLM processing is the KV (Key-Value) cache which stores the keys and values of the prompt and generated tokens [22]. Its size not only accounts for a significant portion of the memory capacity, but also increases as the token sequence length or batch size increases [20, 34]. Such a large memory capacity requirement has been increasing the cost of LLM services as multiple GPUs are used even for inferences [13, 32].

To address the memory capacity requirement, an alternative way is to utilize the slow but large host memory connected to CPUs. Such host memory offloading can reduce the cost of LLM processing drastically at the cost of

increased latencies [2, 7]. A recent study, FlexGen, showed that both of the model weights and KV cache can be offloaded to the host memory, supporting a single GPU LLM computation [24]. Such a single GPU LLM system is suitable for throughput-oriented LLM inferences.

In such a LLM system with the host memory offloading, the main performance bottleneck is the limited communication bandwidth between host memory and GPU via PCIe interconnect. As both weights and KV cache data must be transferred to the small GPU memory, even if the overlapped communication and computation are employed, GPU becomes idle waiting for the next operands to be transferred from the host memory. To reduce the amount of data transfers, FlexGen advocates to use a large batch size to amortize the cost of the weight transfers by increasing the reuse of weights while they are in GPU memory. Alternatively, recent studies investigated filtering techniques for KV cache entries using only part of the KV cache, resulting in approximate outcomes. Although such techniques can reduce the KV cache transfer to the GPU significantly, the approximate nature affects the outcome accuracy [28, 30, 31]. This study opted to avoid such accuracy changes for high fidelity LLM services.

For the host memory offloading without approximation, using a large batch can improve the reuse chance of weights. However, supporting a large batch size with long sequence lengths requires a huge host memory with several hundreds of gigabytes to terabytes for the KV cache alone. The prior work supports a token dynamic recomputation of the KV cache contents to reduce the memory capacity and communication bandwidth consumption [13]. It stores only the actual token IDs of the prompt and generated tokens and dynamically recomputes the keys and values required for the current layer. However, our evaluation shows such *token recomputation* dominates the GPU computation capability.

To address the memory and communication challenges of the host memory offloading, this study proposes a novel *activation checkpointing* scheme based on *Activation caching*. Unlike the token recomputation relying on a full prefill step, activation values of all layers are stored in the memory. Computing the keys and values of a layer from the activation values requires a modest computation capacity. Keeping activation checkpoints instead of keys and values reduce the memory capacity consumption and the communication traffic by half. The proposed technique maintains the same result as the conventional KV caching.

Although replacing the entire KV cache with the Activation cache can reduce the memory usage by half, the amount of computation with activation recomputation also increases as the batch size and/or sequence length increases. Therefore, the system requires a balanced approach to determine the best ratio of KV cache entries and Activation cache entries stored in the host memory. Note that in the limited GPU memory, only Activation cache entries are stored. Our KV-Activation hybrid cache stores parts of the prior KV cache entries as activation representation, and the remaining parts as key and value representation. We propose an algorithm to find the best ratio, and a scheduling tech nique packing requests into mini-batches with the best ratio of KV and Activation entries for computation. Figure 1 presents the existing system with the KV cache (a) and our KV-Activation hybrid cache (b). By using the best ratio of KV and Activation cache sizes, the host-GPU communication latency and the time for activation recomputation and transformer computation are balanced to maximize the throughput.

We implemented the proposed KV-Activation hybrid caching with activation checkpointing called HybridServe. We extended vLLM 0.4.3 with the host memory offloading capabilities for both weight and KV cache. We evaluate HybridServe with four variants of OPT models, showing the throughput improvement of 2.19× over FlexGen, and 1.35× compared to the Activation cache-only system in geometric mean. The source code will be publicly available after publication. The main contributions of the paper are as follows.

- This paper identifies the communication and computation imbalance problem in a LLM system with the host memory offloading.
- It proposes a novel activation checkpointing technique with Activation cache to reduce the recomputation requirement significantly while cutting down the memory capacity and communication bandwidth consumption.
- It proposes a KV-Activation hybrid caching mechanism. It selects the best ratio of KV and activation representations to balance the communication and recomputation.

2 Background

2.1 Large Language Models

Large Language Models (LLMs) are composed of multiple transformer decoder layers. Each layer applies a series of operations to the input token. Initially, the input token is encoded into an embedding vector (A^0) via embedding lookup and positional encoding:

$$A^0 = pos\_encoding(emb\_lookup(ID_{token})) \quad (1)$$

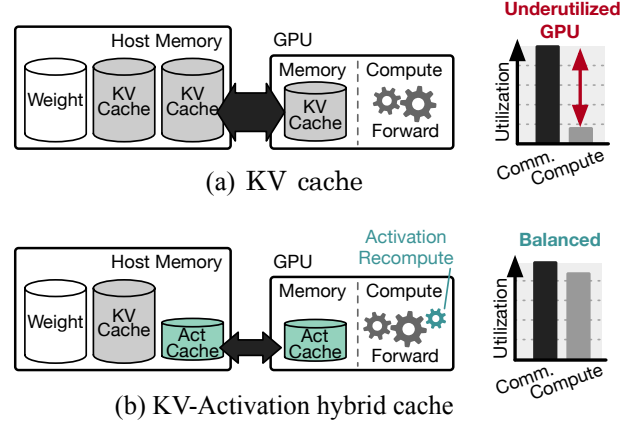Within each decoder layer, the input activation A^i undergoes query, key, and value (QKV) generation, followed by

(a) KV cache

(b) KV-Activation hybrid cache

Figure 1. (a) Existing system with the KV cache, (b) System with KV-Activation hybrid cache.

Input and Activations	
ID_{token}	Token ID
A^i	Input activation of ith decoder
Q^i, K^i, V^i	Query, key, value of ith decoder
K_c^i, V_c^i	KV cache of ith decoder
$AttVal^i$	Multi-head attention output of ith decoder
$Proj^i$	Output of projection of ith decoder
Weights	
W_Q^i, W_K^i, W_V^i	Weights of QKV generation in ith decoder
W_{Proj}^i	Weight of projection in ith decoder
W_{FFN1}^i, W_{FFN2}^i	Weights of feed-forward network in ith decoder

Table 1. Symbols in transformer decoder operations.

concatenation with KV pairs of previous tokens (KV cache):

$$[Q^i \quad K^i \quad V^i] = A^i \times [W_Q^i \quad W_K^i \quad W_V^i] \quad (2)$$

$$[K_c^i \quad V_c^i] = [concat(K_c^i, K^i) \quad concat(V_c^i, V^i)] \quad (3)$$

Next, multi-head attention (MHA) is performed, followed by the projection operation that transforms the attention values for the next step:

$$AttVal^i = softmax(Q^i \times (K^i)^T) \times V^i \quad (4)$$

$$Proj^i = AttVal^i \times W_{Proj}^i \quad (5)$$

Finally, the output of the projection is passed through feed-forward networks (FFNs) to produce the output for the next decoder layer:

$$A^{i+1} = activation((Proj^i) \times W_{FFN1}^i) \times W_{FFN2}^i \quad (6)$$

The LLM inference process is composed of two distinct phases: the prefill phase and the generation phase. The prefill phase encodes user requests (i.e., prompts) into the context, while the generation phase produces the next tokens based on the encoded context. Within the operations in each decoder layer, most can easily reuse weights through batching,
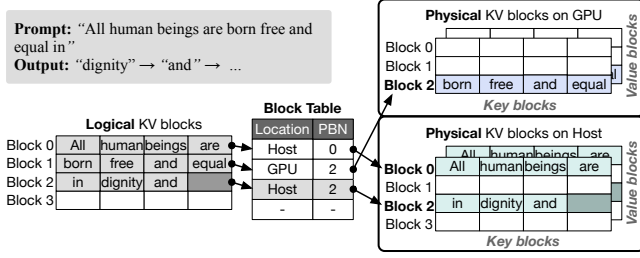
Figure 2. KV cache management with PagedAttention [13].

increasing arithmetic intensity (i.e. number of arithmetic operations per unit of memory traffic). However, multi-head attention exhibits different characteristics. Even when batched, the key and value tensors are not shared across requests, which restricts reuse and prevents efficient tensor flattening [6]. To overcome this limitation, Orca proposes selective batching, which applies batching to all operations except multi-head attention. This improves GPU throughput by enhancing on-chip weight reuse [32].

2.2 Memory Management for KV Cache

To generate the next token in the generation phase, attention is performed between the current token and all previous tokens. Modern LLM inference frameworks use a *KV cache* (Key-Value cache), storing keys and values of previous tokens in memory for subsequent token generation steps. With the KV cache, the QKV generation operation for previous tokens can be skipped in each generation step. Since the token generation of an LLM does not terminate until the special <end-of-sequence> token is generated, the size of the KV cache also continues increasing during inference.

Inference engines typically reserve a large amount of memory for the KV cache to avoid the overhead of frequent memory allocations for every token generation. However, such coarse-grained memory reservation can lead to memory fragmentation when the actual length of generated tokens is smaller than the reserved capacity. To tackle this challenge, vLLM introduced PagedAttention, a demand paging technique for managing KV cache [13]. The primary objective of PagedAttention is to reduce memory fragmentation by enabling continuous KV cache allocation across non-contiguous memory regions. To achieve this goal, PagedAttention applies two main techniques illustrated in Figure 2: (1) It uses *KV blocks* as the unit of memory allocation, grouping KV cache tensors for a set number of tokens, and (2) it adds a *block table* on top of the GPU virtual memory to categorize KV blocks into virtual and physical blocks. Virtual blocks are allocated continuously for each request, ensuring the appearance of contiguous memory, while physical blocks are mapped non-contiguously on the actual memory.

2.3 Throughput-oriented LLM tasks.

There are two types of key performance metrics for LLM serving systems: latency and throughput. Latency is measured as the total elapsed time for generating tokens, which includes the time taken to generate the first token (Time-To-First-Token) and the time taken to generate each subsequent token (Time-Between-Token) [1, 20, 37]. The throughput is defined as the number of tokens generated per unit of time. The appropriateness of these metrics varies depending on the type of LLM tasks. While real-time serving tasks like chatbots are highly sensitive to latency [20], there are still LLM tasks with less stringent latency requirements [5, 15, 17, 18]. These tasks primarily include relatively non-interactive jobs such as evaluating LLMs on large datasets or summarizing extensive documents. Given their lower interaction with users, longer latency can be tolerated, and throughput becomes a more crucial performance metric.

2.4 LLM Inference Serving with Host Memory

Host memory offloading provides a cost-effective solution for managing workloads with relaxed latency requirements. It enables the use of model weights and KV cache that exceed GPU memory capacity. This approach allows LLM inference in resource-constrained environments by offloading weights and KV cache to host memory and streaming them to the GPU as needed, effectively balancing computational demands and memory resources [2, 7, 14, 24, 25, 35]. It reduces the substantial costs and dependence on high-performance interconnects in existing systems, where all tensors must fully reside in GPU memory [10–12, 23].

Batched LLM inference. In LLM with host-memory offloading, the key objective is to maximize the overlap between token generation and data transfer. While multi-GPU LLM serving systems that aim to meet service-level objectives (SLOs) focus on balancing latency metrics [1, 20, 37] rather than increasing batch size [13, 16, 19, 32], throughput-centric LLM systems with host memory offloading mostly focus on increasing batch size [2, 24]. Larger batch sizes improve weight reuse, hiding data transfer overhead under computation and thus enhancing throughput. In throughput-oriented LLM tasks, this shifts the focus from latency optimization to maximizing throughput.

DeepSpeed-Inference [2] is an early work that concentrates on increasing the batch size. It offloads most of the weight parameters to the host memory and loads them to the GPU memory during inference in a streaming, layer-granular manner. This approach increases batch size, enabling more efficient reuse of weights in GPU memory. However, when the size of intermediate tensors and activations during computation exceeds the GPU's memory capacity, further batch size expansion becomes limited.

FlexGen [24] mitigates the limitation of batch size through a technique called zig-zag scheduling. FlexGen does not require all activations for an entire batch to be loaded into the GPU at once. Instead, requests are divided into smaller mini-batches that fit within the GPU's memory capacity. This strategy enhances weight reuse and improves throughput by enabling larger batch sizes, even when the activation tensor footprint exceeds the available GPU memory.

Reducing data transfer via sparsification. Several prior studies have reduced the data transfer bottleneck of host memory offloading by lossy sparsification [14, 25, 35]. Based on the observation that not all data used in LLM computation always significantly contributes to the generation of the next token [29, 34], these works exploit only critical (i.e., important) data for the token generation. ALISA [35] and InfiniGen [14] selectively use a subset of the KV cache for multi-head attention. On the other hand, PowerInfer [25] selectively loads critical weight parameters, and applies CPU-GPU hybrid attention, performing computations with less important weights in host CPU.

By adopting weight and KV cache sparsification, host memory offloading becomes viable for real-time inference by significantly reducing latency compared to conventional batch-size-centric offloading approaches. However, sparsification methods present potential risks to correctness and safety, as they rely on only a subset of data for computation [28, 30, 31]. For throughput-oriented tasks that tolerate higher latency but have strict accuracy requirements, minimizing the degree of approximation introduced by sparsification is essential to ensure reliable performance.

3 Motivation

3.1 Challenges for Increasing the Batch Size

As discussed in Section 2.4, with model weights offloaded to host memory, increasing the batch size is a common strategy to enhance generation throughput. However, simply increasing the batch size does not consistently scale throughput. Figure 3 (a) shows the relationship between the batch size and the generation throughput in FlexGen [24]. Throughput scales linearly with batch size smaller than 256, but eventually saturates and even degrades when the batch size keeps growing. This is due to the corresponding increase in KV cache transfer volume. Although larger batches improve weight reuse during token generation, the KV cache, essential for attention operations, cannot be shared across batched requests. Consequently, PCIe transfer volume grows with the sum of the context lengths, leading to GPU underutilization. As shown in Figure 3 (b), the KV cache transfer volume increases linearly to the batch size when the token length is reserved. With a batch size of 16, total KV cache traffic is 21 GB per token generation, and the traffic rises sharply to 168 GB at a batch size of 128. Due to the PCIe bottleneck incurred
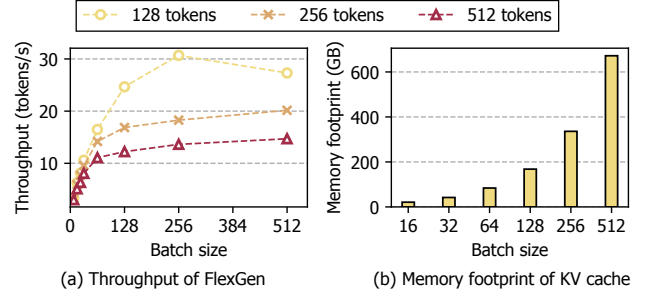

Figure 3. Performance evaluation of FlexGen [24] with OPT-30B: (a) token generation throughput with varying input prompt lengths, and (b) memory footprint of the KV cache with 1024 input tokens.

Prompt	B=1	B=8	B=16	B=64	B=256	B=1024
128 tokens	3.96	5.87	6.83	6.86	6.76	5.77
256 tokens	3.93	5.76	6.42	7.02	7.14	7.15
512 tokens	3.49	5.57	6.35	7.27	6.99	6.28

Table 2. Token generation throughput of PowerInfer [25] with various input prompt lengths and batch sizes in LLaMA2-70B [27]. B denotes batch size.

by the massive size of KV cache, the FlexGen reports GPU utilization of only 7.4% with the batch size of 128.

Even a state-of-the-art approach of host memory offloading, PowerInfer [25], faces throughput saturation due to the increasing KV cache volume. PowerInfer mitigates single-GPU limitations by prioritizing critical model parameters on the GPU and distributing attention operations between the CPU and GPU. However, as shown in Table 2, the increased traffic of KV cache also leads to throughput saturation as the batch size grows, similar to what is shown for FlexGen in Figure 3 (a). While KV cache reuse is possible for requests with shared prefixes, it can only be applied in limited cases, as even a slight token variations impede KV cache reuse [16, 36].

3.2 Limitation of KV Cache Recomputation

KV cache recomputation. One way to reduce the memory footprint in LLM serving is to recompute KV cache from every token ID, instead of storing it. Such *recomputation of KV cache* generally involves tracking only the token IDs of the accumulated context (prompt + generated tokens) and using that context to go through the prefill phase, as explained in Section 2.2. This approach not only enables larger batch sizes by freeing GPU memory from the KV cache, but also increases GPU utilization by providing more computation.

However, since completely removing the KV cache incurs excessively repetitive computations across token generations, maintaining only a portion of the context as KV cache
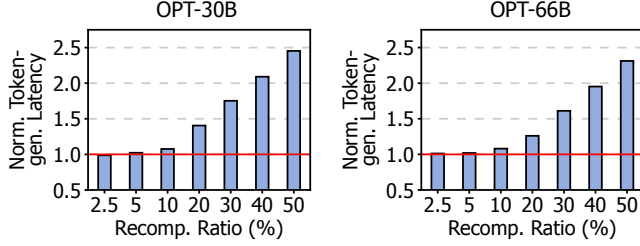
Figure 4. Token generation latency normalized to the latency without recomputation, with varying recomputation ratios for OPT-30B (left) and OPT-66B (right). The red line indicates the basis for normalization

and the rest as token IDs is more promising. As balancing computation and communication is a key challenge for improving throughput, the ratio of KVs to token IDs must be adjusted carefully.

Marginal performance gains of token recomputation. Figure 4 demonstrates the impact of increasing recomputation ratios on token generation latency for OPT-30B and OPT-66B. As the ratio of recomputation rises, the reduction in host-GPU data transfer is offset by a significant increase in token generation latency. With a batch size of 64 and context lengths of 1024 for OPT-30B and 512 for OPT-66B, latency increases by 1.45× for OPT-30B and 1.31× for OPT-66B at a 50% recomputation ratio. The reason of performance degradation is excessive temporal overhead of KV cache recomputation: Recomputation time exceeds the savings from reduced data transfer. This indicates that token recomputation-based KV compression sacrifices performance for memory efficiency.

3.3 Potentials of Activation Cache

An alternative to the conventional KV cache is to maintain activations. This section explores the potential of the **Activation cache**, which accelerates KV recomputation through **activation checkpointing**. By checkpointing all decoder input tensors into the Activation cache, the stored activations can serve as primers for KV recomputation in each decoder layer. This approach improves the efficiency of KV recomputation by skipping operations after QKV generation without a loss of accuracy.

Comparison with KV cache. Compared to the KV cache, this *activation recomputation* requires an additional linear transformation of Equation 7 to convert activation checkpoints ($\mathbf{A_c}$) into key and value tensors ($\mathbf{K_c}$, $\mathbf{V_c}$).

$$[\mathbf{K_c} \quad \mathbf{V_c}] = \mathbf{A_c} \times [\mathbf{W_K} \quad \mathbf{W_V}] \tag{7}$$

By storing only one tensor ($\mathbf{A_c}$) through activation checkpointing rather than saving both key and value tensors, this approach reduces memory usage and host-GPU traffic by 50%. Moreover, while conventional KV caching may lead to GPU underutilization, activation checkpointing efficiently
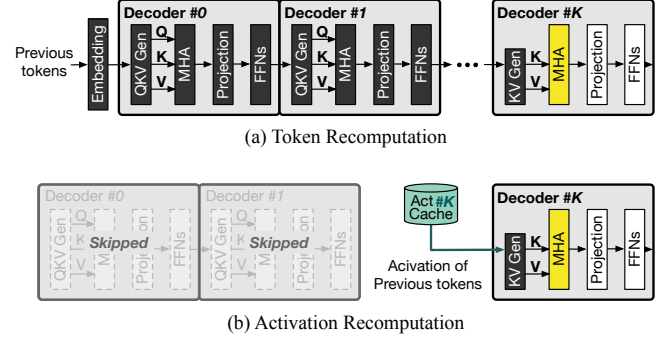

(a) Token Recomputation

(b) Activation Recomputation

Figure 5. Computation difference for retrieving context of previous tokens for multi-head attention, in (a) Token recomputation and (b) Activation recomputation.
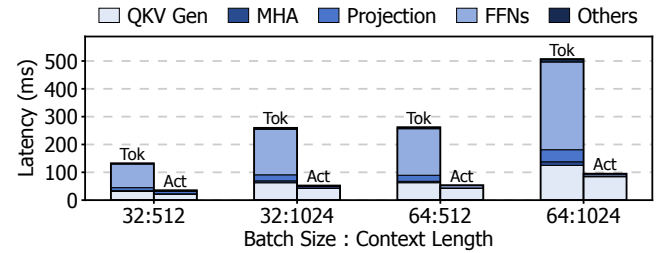

Figure 6. Breakdown of a single layer execution time of OPT-30B, with token recomputation from token IDs (Tok) and activation recomputation from activations (Act).

leverages underutilized GPU computation units through recomputation. As a result, this extra computation step provides an effective balance between reducing host-GPU transfer and minimizing computational overhead.

Comparison with token recomputation. Activation checkpointing substantially reduces computational overhead compared to the conventional KV recomputation (i.e. token recomputation). Figure 5 illustrates the difference between token recomputation and activation recomputation. In Figure 5 (a), to generate the KV required for the multi-head attention (MHA) of the k-th layer, all preceding layers must be recomputed with dependency. In contrast, Figure 5 (b) shows that with activation checkpointing, only the activation from the k-th layer is needed to generate the KV, bypassing the need to recompute earlier layers.

Figure 6 provides a breakdown of the single-layer execution latency for the OPT-30B model [33], with varying batch sizes and context lengths. This latency includes the time needed to recompute the KV for a single token. Tok represents the execution latency with token recomputation, while Act shows the latency when applying activation checkpointing for activation recomputation. We observe that activation recomputation reduces latency by 78% compared to token recomputation in geometric mean, saving considerable computational time with activation checkpointing.
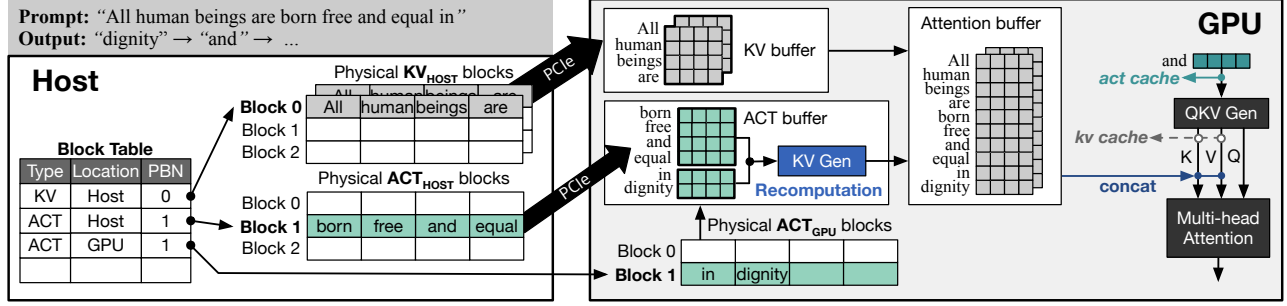
Figure 7. Overview of the proposed system with KV-Activation hybrid caching mechanism.

The need of KV-Activation hybrid caching. While replacing KV cache to half-sized Activation cache through activation checkpointing has enough potential of throughput improvement by allowing a larger batch size, it may not be practical to replace the entire KV cache to Activation cache. Since activation recomputation can only performed after the checkpoint is loaded, fully replacing the KV cache with an Activation cache limits the available time for recomputation to the duration of host-to-GPU weight loading. However, as recomputation still requires some GPU cycles, retaining a portion of the KV cache is necessary to allow sufficient time for activation-to-KV conversion. To optimize both communication and computation resources, we propose a novel **KV-Activation hybrid caching** strategy in the following section: Using KV cache and Activation cache together.

4 HybridServe System Design

4.1 Overview

We introduce HybridServe, an offloading-based LLM inference framework designed to optimize both communication and computation resources through a hybrid KV-Activation caching strategy. HybridServe efficiently manages activation recomputation during data transfers, significantly reducing the memory footprint of the KV cache while preserving LLM accuracy. Furthermore, HybridServe maximizes the overlap between computation and communication in a user-transparent manner, mitigating data transfer bottlenecks and automatically balancing activation recomputation with KV transfers.

In the following sections, we detail two key components of HybridServe's design: First, we present the execution engine (Section 4.2), which flexibly utilizes both KV and Activation caches to optimize performance. Second, we describe the hybrid cache management policy (Section 4.3), which dynamically adjusts the ratio of activation checkpointing. This adaptive approach enables the system to balance recomputation and data transfer in response to workload demands, facilitating efficient concurrent execution.

Hybrid cache blocks. The primary consideration for the hybrid caching is the flexible adjustment of the ratio between KV cache and activation checkpoints to maximize overlap

between computation and communication. To achieve this, our hybrid caching strategy is built on a system leveraging PagedAttention [13], which supports a flexible management of KV cache through logical-physical block mapping, minimizing the memory fragmentation as well. In addition to KV cache block (*KV block*), HybridServe also manages activation checkpoints in block granularity - denoted as Activation cache block (*ACT blocks*). Each block is associated to the same number of tokens, saving either part of the KV cache (KV block) or the activation checkpoints (ACT block).

4.2 Block Mapping and Execution Engine

4.2.1 Mapping Hybrid Blocks to Memory. HybridServe automatically determines the ratio between KV and ACT blocks to optimize the overlap between computation and data transfer. This ratio guides both the activation checkpointing and KV caching processes during the prefill and generation phases. Prompts and generated tokens for each request are allocated within a unified logical block space that spans both GPU and host memory, adhering to the KV and ACT block ratio defined in Section 4.3. These logical blocks are then mapped to physical memory blocks. Since the size of an ACT block is half that of a KV block, HybridServe prioritizes storing activation checkpoints in GPU memory. As a result, ACT blocks can be stored in both GPU and host memory, while KV blocks are typically stored in host memory. The recomputation of ACT blocks is overlapped with the retrieval of KV blocks via PCIe, improving both GPU utilization and bandwidth efficiency. For smaller batch sizes, HybridServe can allocate GPU memory for the KV cache to minimize stall times during recomputation.

Translation and transfer of cache blocks. Figure 7 illustrates the process of advancing to the next generation phase.Each request has its own block table that stores the information of the logical blocks mapped to the context in sequence. The block table entry saves information including the type, location, and physical block number (PBN) for each logical block, indicating whether it belongs to the KV cache or activation cache, and whether it resides in host memory or GPU memory. During the generation phase, physical blocks are transferred from host memory to two types of reserved
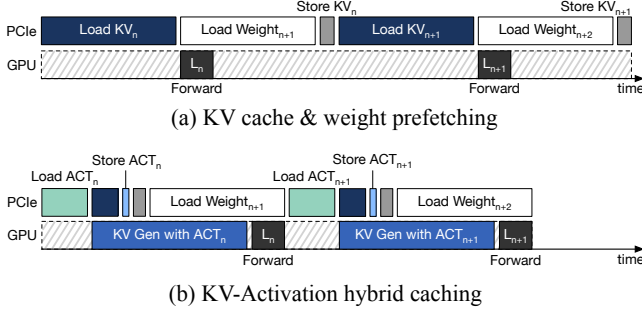
(a) KV cache & weight prefetching

(b) KV-Activation hybrid caching

Figure 8. Timeline comparison. (a) Existing system with KV cache and weight prefetching (b) Asynchronous activation recomputation (KV Gen) with KV-Activation hybrid cache.

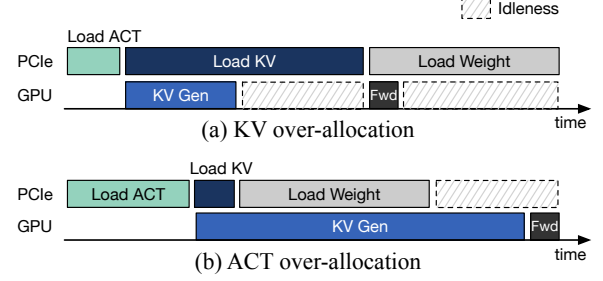(a) KV over-allocation

(b) ACT over-allocation

Figure 9. Pipeline imbalance. (a) GPU underutilization due to over-allocation for KV cache, (b) PCIe underutilization due to over-allocation for Activation cache.

buffers within the GPU: the *KV buffer* for KV cache and the *ACT buffer* for recomputation. Both buffers operate concurrently, facilitating maximum overlap between PCIe data transfers and GPU computation through a double-buffering mechanism.

4.2.2 Execution engine. The right side of Figure 7 illustrates the token generation workflow of HybridServe execution engine. For each iteration of generation phase, HybridServe recomputes the key and value tensors using pre-loaded activation checkpoints stored in the *ACT buffer* (*KV Gen* in Figure 7). Once this KV recomputation completes, recomputed KVs are concatenated with pre-loaded KV cache. In addition to KV recomputation, HybridServe also have to perform QKV generation for the input token of current iteration ("and" in Figure 7). The newly generated KV tensors are concatenated with the KV cache to perform the multi-head attention operation. During this process, HybridServe determines whether to store the newly generated token as an activation checkpoint or a KV cache. This decision is made by following the policy of Section 4.3.

Mini-batch. If the total size of hybrid cache blocks for a single layer exceeds the available GPU buffer capacity, it becomes necessary to expand the GPU buffer. However, this expansion reduces the available space for storing ACT blocks on the GPU (ACT_{GPU} in Figure 7), negatively impacting storage efficiency and overall system performance. Furthermore, the batch size is constrained by the GPU memory capacity, limiting the reuse of weight parameters. To alleviate these limitations, HybridServe divides the requests in the generation phase into multiple *mini-batches* for inference. While conventional batches are scheduled in iteration-level, mini-batches are scheduled and switched in layer-level. All decoder operations for each mini-batch in the current layer must be completed before proceeding to the next layer, following a scheduling strategy similar to FlexGen [24]. This maximizes weight reuse and improves resource utilization.

Activation checkpointing and recomputation. HybridServe integrates activation checkpointing and recomputation

into an asynchronous pipeline [8], enabling simultaneous PCIe data loading with GPU computation. Figure 8 presents the execution timeline of generation phase with (a) the conventional KV cache and (b) the KV-Activation hybrid cache employed by HybridServe, respectively. The "PCIe" pipeline represents data transfers, while the "GPU" pipeline shows operations related to recomputation and the forward pass. In both configurations, the weights for the $(n + 1)$th layer are prefetched during the forward pass of the nth layer to minimize latency. In Figure 8 (a), the GPU computation of KV cache-only approach is dedicated solely to the forward pass for token generation within the batch. As a result, the GPU computation unit remains largely underutilized while loading KV data, fetching weights, and storing the generated KV cache through PCIe. In contrast, as shown in Figure 8 (b), KV-Activation hybrid caching reduces the amount of KV data to load by utilizing activation checkpoints. Activation recomputation (denoted as KV Gen) begins immediately after the required activations are loaded and runs concurrently with tasks such as activation checkpointing, KV storage, and weight loading. This asynchronous task overlapping minimizes communication overhead, reducing generation latency and enhancing performance.

4.3 Cache Management Policy

As mentioned earlier, achieving efficient overlap between data transfer and computation is important for reducing latency in offloading-based LLM inference. However, imbalances in the hybrid cache can create bottlenecks. Figure 9 shows two possible cases of imbalance incurred by (a) over-allocation of KV blocks and (b) excess of ACT blocks. While excessive KV blocks strains PCIe bandwidth therefore resulting in GPU underutilization, over-reliance on activation checkpointing also suffer from recomputation delays.

To mitigate these issues, we propose a cache allocation and scheduling policy for KV-Activation hybrid cache. As illustrated in Figure 10, our cache management policy consists of three key steps: ❶ **Host memory block allocation** determines cache allocation either as KV blocks or as ACT
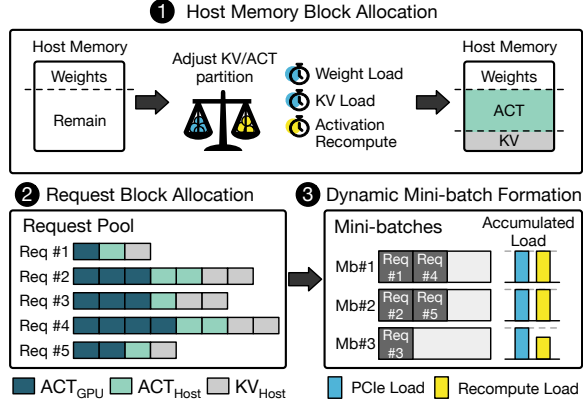
Figure 10. Overview of cache management policy for hybrid cache block allocation and request scheduling.

blocks in host memory space. ❷ **Request block allocation** assigns KV and activation blocks to each request based on the allocation ratio determined for host memory blocks. ❸ **Dynamic mini-batch formation** organizes each mini-batch to balance PCIe communication and recomputation.

Problem definition. There are two major terms regarding the execution time of HybridServe pipeline: The total latency for PCIe communication (T_{PCIe}) and the computation latency ($T_{Computation}$). The primary objective of the optimization problem is to find the appropriate number of KV and ACT blocks that balances T_{PCIe} and $T_{Computation}$, thereby minimizing pipeline idleness. This balance is crucial to ensuring efficient overlap between PCIe communication and GPU computation in the inference pipeline of HybridServe. To formalize this optimization problem, we aim to minimize the absolute difference between T_{PCIe} and $T_{Computation}$ to keep both pipelines balanced and prevent either from becoming a bottleneck. Equation 8 shows the basic objective of the cache management policy.

$$Minimize \; |T_{PCIe} - T_{Computation}| \qquad (8)$$

T_{PCIe} is determined by the time required to load weights and KV blocks from host memory, while the computation latency, while $T_{Computation}$ accounts for generating KV tensors from activation blocks residing in both GPU and host memory. Table 3 defines the symbols used for describing the allocation and scheduling problem, and T_{PCIe} and $T_{Computation}$ can be represented with the symbols of the Table as follows:

$$T_{PCIe} = T_{load\_w} + T_{load\_kv}(\#KV_{Host}) \qquad (9)$$

$$T_{Computation} = T_{kv\_gen}(\#ACT_{Host} + \#ACT_{GPU}) \qquad (10)$$

Considering Equation 9 and Equation 10, the objective of this optimization problem can be translated into adjusting the number of activation blocks in GPU ($\#ACT$) and KV blocks ($\#KV$) in both GPU and host memory. This is for making T_{PCIe} and $T_{Computation}$ aligned closely to each other,

Terms & Functions for Time	
T_{load_w}	Latency for one decoder block weight load
$T_{kv\_gen}()$	Function for KV generation (recompute) latency
$T_{load\_kv}()$	Function for KV cache load latency
Terms for # of Blocks	
$\#KV_{Host}$	Number of KV blocks in host memory
$\#ACT_{Host}$	Number of ACT blocks in host memory
$\#ACT_{GPU}$	Number of ACT blocks in GPU memory

Table 3. Symbols for allocation and scheduling decision.
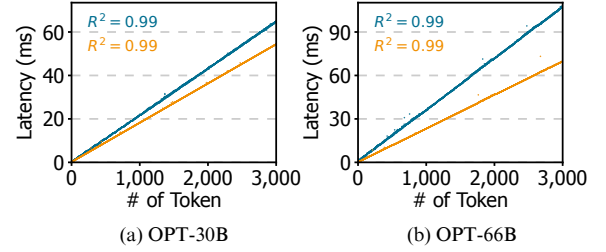

(a) OPT-30B   (b) OPT-66B

Figure 11. Sampling points of T_{kv_gen} (blue) and T_{load_kv} (orange) for linear regression. R^2 is the correlation coefficient.

so that PCIe communication and GPU computation can be overlapped efficiently.

Sampling-based linear regression. Considering the Equation 9 and Equation 10, the correlation between time functions (i.e. T_{load_kv} and T_{kv_gen}) and their inputs (i.e. # of tokens) should be found to solve the optimization problem for cache management policy. Figure 11 shows the sampling result for T_{kv_gen} and T_{load_kv} with various number of tokens, in an NVIDIA RTX 4090 GPU with PCIe 4.0 x16 environment. As shown in the figure, both T_{kv_gen} and T_{load_kv} are fairly linear to the number of tokens: While the correlation coefficient (denoted as R^2) closer to 1 indicates the perfect linear correlation, the R^2 value of both functions with respect to the input reports 0.99. Based on this observation, we perform linear regression for these two functions, and use approximated linear functions in following subsections.

4.3.1 Host Memory Block Allocation. Although the host memory allocation is performed statically only once at the start of HybridServe, it plays a crucial role in reducing the variance in recomputation time and data transfer time for each request during whole inference progress. Figure 10 ❶ and Algorithm 1 shows a two-step algorithm that determines the appropriate number of ACT and KV blocks to be allocated in host memory. In the first step, the algorithm calculates the initial number of ACT or KV blocks needed to eliminate idle time by comparing the recomputation time of GPU-resident ACT blocks with the weight loading time. In the second step, it calculates the additional ACT and KV blocks required to fully utilize the remaining host memory.

Initial allocation. Initially, the number of host blocks required to eliminate idle time is calculated by comparing the

recomputation time of ACT blocks allocated to the GPU with the time taken to load weights (lines 10-18 of Algorithm 1). If weight loading takes longer, the algorithm calculates the number of additional host ACT blocks (ACT_{init}) to be processed, to avoid GPU computation unit idleness. Conversely, if the PCIe bus is expected to idle due to long computation time, the number of KV blocks (KV_{init}) to be fetched from the host is calculated.

Remaining allocation. In the next step, the number of KV and ACT blocks to reside in the remaining host memory is determined, to ensure balance between the two pipelines. After accounting for the memory used by weight parameters and the blocks allocated in the first step, the remaining capacity of host memory can be calculated (lines 22-23 of Algorithm 1). To fill this space, the number of additional ACT and KV blocks (#ACT and #KV) required in order to balance PCIe communication and GPU recomputation is computed using a linear system (lines 25-26). Given that S_{ACT} is equal to (= $\frac{1}{2}S_{KV}$) and both T_{kv_gen} and T_{load_kv} are linear, the cost of this calculation is low.

4.3.2 Request Block Allocation.
Based on the values obtained through the two-step allocation policy, the number of ACT and KV physical blocks to allocate in host memory is set to $ACT_{init} + ACT_{remain}$ and $KV_{init} + KV_{remain}$, respectively. HybridServe ensures that each request maintains the same ratio of $\#ACT_{req}$ to $\#KV_{req}$ as the ratio of $\#ACT_{Host}$ to $\#KV_{Host}$, as shown in Figure 10 ❷ and Equation 11.

$$\#ACT_{req} : \#KV_{req} = \#ACT_{Host} : \#KV_{Host} \quad (11)$$

After the prefill phase, context blocks are stored through either activation checkpointing or KV caching according to the ratio specified by Equation 11. During the generation phase, new blocks are allocated to maintain this ratio. For instance, if the ratio of $\#ACT_{Host}$ and $\#KV_{Host}$ is 3:1 when five ACT blocks and two KV blocks are present, the next block allocated would be an ACT block.

4.3.3 Dynamic Mini-batch Formation.
While Algorithm 1 ensures a balance between KV and activation within a single request, it's also important to consider this balance at the mini-batch level for better performance. Since a mini-batch handles both computation and communication, balancing KV and activation dynamically is essential. This approach not only maximizes PCIe and computational resource utilization but also improves the efficiency of double buffering.

Ensuring pipeline balance. To achieve the best performance, mini-batch should always keep the balance between $T_{kv\_gen}()$ and $T_{load\_kv}()$. We use *balance* as a metric for measuring how balanced two pipelines are in a mini-batch. Basically, *balance* is a ratio of T_{kv_gen} and T_{load_kv} in current mini-batch, as defined in Equation 12.

$$balance = T_{kv\_gen}(\#ACT_{mb})/T_{load\_kv}(\#KV_{mb}) \quad (12)$$

Algorithm 1 Hybrid Cache Allocation Policy

Parameter: T_{load_w}: Weight load latency for one decoder block
$\quad\quad\quad\quad \#ACT_{GPU}$: # of ACT blocks in GPU memory
$\quad\quad\quad\quad M_{Host}$: Total capacity of host memory
$\quad\quad\quad\quad S_{weight}$: Total size of weight parameter
$\quad\quad\quad\quad S_{KV}$: Size of a KV block
$\quad\quad\quad\quad S_{ACT}$: Size of an ACT block (= $\frac{1}{2}S_{KV}$)
Output: $\#ACT\_Host, \#KV\_Host$

```
1:  # Step 1: Initial allocation
2:  ACT_init, KV_init ← initial_cache_allocation()
3:
4:  # Step 2: Allocate remaining blocks after initial alloc
5:  ACT_remain, KV_remain ← alloc_remaining(ACT_init, KV_init)
6:
7:  # Step 3: Return #ACT_Host, #KV_Host
8:  return (ACT_init + ACT_remain), (KV_init + KV_remain)
9:
10: def initial_cache_allocation():
11:     # Calculate time balance
12:     ACT_init, KV_init ← 0, 0
13:     T_budget = T_load_w − T_kv_gen(#ACT_GPU)
14:     if T_budget ≥ 0 then
15:         ACT_init ← find #ACT s.t. T_kv_gen(#ACT) = T_budget
16:     else
17:         KV_init ← find #KV s.t. T_load_kv(#KV) = −T_budget
18:     return ACT_init, KV_init
19:
20: def alloc_remaining(ACT_init, KV_init):
21:     # Calculate used memory
22:     M_occupied = S_ACT × ACT_init + S_KV × KV_init
23:     M_remaining = M_Host − S_weight − M_occupied
24:     ACT_remain, KV_remain ← find #ACT, #KV s.t.
25:         M_remaining = S_ACT × #ACT + S_KV × #KV
26:         T_kv_gen(#ACT) = T_load_kv(#KV)
27:     return ACT_remain, KV_remain
```

Note that $\#KV_{mb}$ and $\#ACT_{mb}$ are number of KV blocks and ACT blocks for current mini-batch. Since an ideal value of *balance* is 1, the purpose of the request scheduling algorithm is making *balance* converge to 1 in each mini-batch. To transform this to the minimization problem, we define a cost function $F_b()$ as Equation 13.

$$F_b(\#ACT_{mb}, \#KV_{mb}) = max(balance, 1/balance) \quad (13)$$

The goal of our mini-batch formation algorithm is minimizing $F_b()$ for each mini-batch.

Solving bin packing problem for mini-batches. Our dynamic mini-batch formation algorithm employs a greedy approach that seeks to minimize both the number of mini-batches and the imbalance metric *balance*. The algorithm begins by defining $\#ACT_{max}$ and $\#KV_{max}$ based on the size of the available GPU buffer, which serve as upper bounds for $\#KV_{mb}$ and $\#ACT_{mb}$ (i.e., bin capacities in a bin packing problem). It then evaluates whether a request can be added to the current mini-batch by checking if it fits within the size

constraints (in terms of #*KV* and #*ACT*) and if its inclusion reduces $F_b()$—the imbalance metric—relative to the current mini-batch state. If both conditions are met, the request is incorporated into the mini-batch. Once no further requests can be added, the algorithm proceeds to form a new mini-batch. This process is repeated until all requests are allocated to mini-batches, ensuring an efficient balance of resources.

5 Evaluation

5.1 Methodology

Implementation. HybridServe is a single-GPU LLM inference system that extends vLLM 0.4.3 [13], incorporating host memory offloading capabilities for both weights and KV caches. KV-Activation Hybrid caching along with cache block allocation and mini-batching scheme is implemented by modifying 12K lines of code using Python and C++/CUDA with PyTorch framework [3]. To ensure seamless integration of the hybrid cache with the PagedAttention mechanism, HybridServe expands PagedAttention kernel of vLLM to enable multi-head attention functionality across diverse KV buffer types, extending beyond the conventional KV cache.

Models. Throughout the evaluation, we employ the Open Pretrained Transformer (OPT) [33], a widely-used LLM with publicly available checkpoints. We evaluate four model sizes: 6.7B, 13B, 30B, and 66B parameters, all pre-trained within float16 precision. Note that OPT-6.7B is capable of running entirely within the GPU memory (i.e. without host memory offloading), and we use this model to evaluate the efficiency of host memory offloading, as well as analyzing the extent of data transfer volume reduction.

Baselines. We compare our system against two modern LLM inference frameworks that supports host memory offloading: DeepSpeed-Inference [2] and FlexGen [24], introduced in Section 2.4. Frameworks employing lossy sparsification techniques, such as ALISA [35] and InfiniGen [14], are not included in our experiments, as we aimed to avoid potential impact on language modeling performance. However, these sparsification methods could still be applied alongside our hybrid cache approach to help reduce data transfer in a complementary way.

For FlexGen, we use the configuration that shows the best performance across various setups. This optimal configuration keeps as many weight parameters as possible on the GPU, while offloading the remaining weights and the KV cache to host memory. Additionally, to prevent Out-of-Memory (OOM) errors caused by intermediate tensors, we adjust the block size and batch size in zig-zag scheduling, enabling the largest feasible batch size for inference. On the other hand, for DeepSpeed-Inference, which does not support zig-zag scheduling, we select the largest possible batch size that could be processed without causing OOM errors during the prefill phase. As a result, the batch size

of DeepSpeed-Inference gets smaller than FlexGen. In addition to prior work, we also compare our system with an Activation-cache only system, denoted as HybridServe-Act-Cache.

Environment Setup. We perform evaluation in a single GPU system featuring an NVIDIA RTX 4090 GPU with 24GB of GDDR6X memory and PCIe 4.0 x16 interface. The host system is powered by a dual-socket 16-core Intel Xeon Gold 6326 processor, with 882GB of DDR4 memory.

Evaluation metrics. The performance evaluation of HybridServe focuses on token generation throughput, a key metric for LLM tasks that leverage host memory offloading. Token generation throughput is defined as a number of tokens generated per unit of time. Additionally, to better understand the performance gains of our system, we analyze the volume of KV transfers between the host and GPU. To assess the efficiency of resource utilization, we calculate the temporal utilization of the GPU using NVIDIA Nsight Systems. Temporal utilization is defined as the percentage of cycles during which the GPU's computation units were active.

5.2 Throughput Improvement

Figure 12 shows the throughput of HybridServe (HybridServe-Hybrid-Cache) in comparison with prior work and HybridServe-Act-Cache. The x-axis of each represents the input prompt length, while the y-axis indicates the token generation throughput. Throughput was obtained by dividing the total number of tokens by the end-to-end latency (= prefill latency + generation latency). The batch size is set to 128, and each request in the batch generates 128 output tokens.

Comparison with KV cache-only system. Compaerd to FlexGen, HybridServe-Hybrid-Cache shows an average throughput improvement of 2.19×, and HybridServe-Act-Cache achieves an average improvement of 1.61×. The throughput improvement of HybridServe above FlexGen is attributed to the utilization of GPU cycles to recompute the KV from the activation, while FlexGen has a significant amount of wasted GPU cycles. This alleviates the data transfer bottleneck by reducing the amount of KV cache transferred from host to GPU, thereby balancing data transfer and computation. The magnitude of the throughput improvement of HybridServe over FlexGen increases as the model size grows; the average throughput improvement of HybridServe-Hybrid-Cache over FlexGen is 2.05× for the OPT-6.7B, and 2.33× for the OPT-66B. This is because larger models take longer to transfer weights, allowing more activation recomputation. The throughput of DeepSpeed-Inference is only 29% of FlexGen and 13% of HybridServe-Hybrid-Cache on average since DeepSpeed-Inference does not support mini-batch scheduling. To process a batch all at once without dividing it into mini-batches requires a significant amount of GPU memory. In other words, the batch size must be reduced in
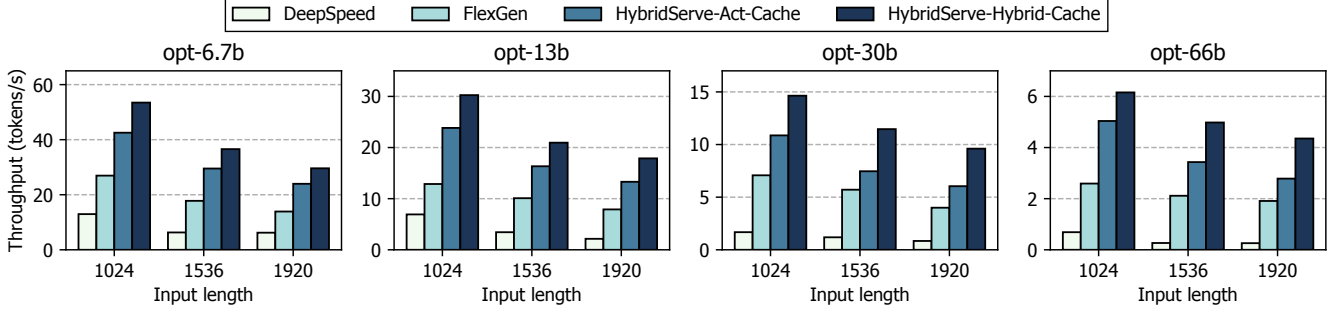
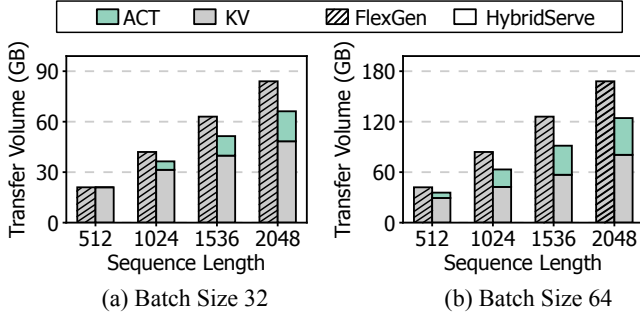**Figure 12.** Throughput of HybridServe with various OPT model sizes.

Figure 13. Breakdown of PCIe transfer volume for KV and Activation using OPT-30B with batch sizes of 32 and 64. Shaded bars represent FlexGen, while solid bars represent HybridServe.

Figure 14. GPU utilization of OPT-30B with varying batch sizes and input prompt lengths, comparing FlexGen and HybridServe.

memory-constrained situations, which leads to a decreased throughput. This gap has also been observed in other studies [24, 35].

Comparison with Activation cache-only system. HybridServe-Hybrid-Cache demonstrates an average throughput improvement of 1.35 × over HybridServe-Act-Cache. Using only Activation cache incurs computation bottleneck from recomputation of KV cache, as opposed to KV cache-only case. Therefore, a properly balanced KV cache and Activation cache lead to the optimal throughput, outperforming both KV cache-only and Activation cache-only systems. Using only the Activation cache still allows more time for KV recomputation with larger models, and it responds more sensitively compared to when using the hybrid cache. Consequently, throughput improvement tends to increase with larger models. This result demonstrates the validity and necessity of KV-Activation hybrid caching of HybridServe.

5.3 Host-GPU Traffic

Figure 13 shows the breakdown of traffic from the host to the GPU during generation with various input sequence lengths for the batch size of (a) 32 and (b) 64. The x-axis is an input sequence length, and the y-axis is the volume of data transfer (i.e. traffic) between host memory and GPU. For each sequence length, the comb-patterned bar on the left indicates
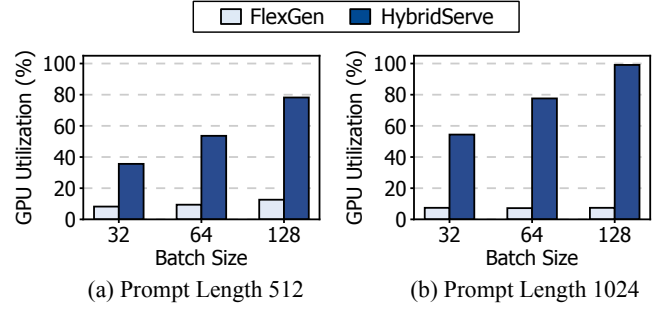
the traffic of FlexGen, and the bar on the right is the traffic of HybridServe. HybridServe achieves a traffic reduction up to 1.27× and 1.38× compared to FlexGen in batch size 32 and 64, respectively by replacing several KV blocks to smaller ACT blocks. Especially, when the batch size increases, the traffic reduction of KV-Activation hybrid caching becomes more emphasized as the improvement of data reuse allows more activation recomputation to be performed, increasing the portion of Activation cache.

5.4 GPU Utilization

Figures 14 compares the GPU utilization of FlexGen and HybridServe under varying batch sizes and input prompt lengths. On average, HybridServe demonstrated a 7.39× higher GPU utilization than FlexGen, with the most significant increase observed at a batch size of 128, where HybridServe achieved 13.39× higher utilization. This substantial improvement in utilization can be attributed to HybridServe 's ability to generate activation checkpoints as key-value (KV) pairs during periods when the GPU would otherwise remain idle in FlexGen.

Notably, while FlexGen only saw a modest increase in GPU utilization, from 8.2% to 12.6% as the batch size increased from 32 to 128, HybridServe, leveraging activation recomputation, exhibited a more pronounced rise in utilization, from 35.6% to 78.2% for the same variation. Furthermore, because
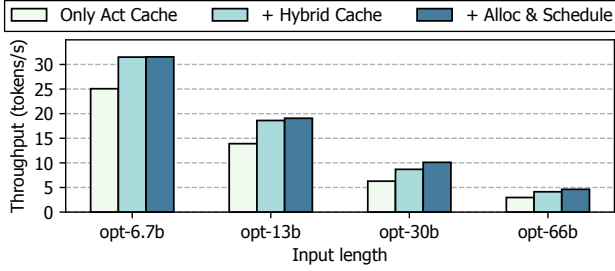
Figure 15. Throughput comparison across various OPT model sizes with a input prompt length of 1920, demonstrating the impact of hybrid caching and cache policies.

activation recomputation involves less computational overhead compared to token recomputation, HybridServe can generate more tokens at the same GPU utilization level.

However, it is important to note that higher GPU utilization does not directly result in proportional throughput gains. For example, doubling GPU utilization does not necessarily mean that the efficiency of converting activations to KV pairs will increase twofold. Nevertheless, HybridServe 's higher GPU utilization compared to FlexGen helps reduce the traffic from KV cache operations, improving overall system efficiency.

5.5 Effect of Block Allocation and Dynamic Packing

Figure 15 demonstrates the effect of progressively applying hybrid caching and cache management policies to the activation cache on throughput. Applying hybrid caching alone yields an average throughput improvement of 1.33×, with gains of 1.23× for OPT-6.7B and 1.39× for OPT-66B, indicating that larger models benefit more. This is because, while activation recomputation opportunities increase with model size, larger models are more prone to latency increases from recomputation-induced bottlenecks. When cache management policies, such as host memory allocation, request allocation, and dynamic bin-packing, are introduced, throughput improvements of 1.6× for OPT-30B and 1.56× for OPT-66B were observed over HybridServe-Act-Cache. For smaller models, the performance gains were minimal, likely due to the default 1:1 host memory split between ACT and KV caches, which closely matches their optimal ratio. In contrast, for OPT-30B and OPT-66B, the optimal KV-to-ACT ratios were 2:1 and 1.78:1, respectively. Adjusting the host memory allocation accordingly enabled greater throughput, particularly for the larger models, by improving overlap efficiency beyond what activation caching alone could achieve.e

6 Related Work

LLM inference with host memory offloading. To address the large GPU memory consumption of the KV cache due to long sequences and large batches, several studies have proposed strategies for offloading the KV cache to host

memory. FlexGen enables LLM inference on a single GPU by offloading the model parameters and KV cache to host memory [24]. ALISA reduces the memory footprint by leveraging the sparsity of the attention weights and dynamically employs host memory offloading and recomputation for the KV cache, depending on its size [35]. InfiniGen offloads the KV cache to host memory and identifies critical entries for each layer in advance to bring them to the GPU [14]. PowerInfer reduces data transfer overhead by selectively utilizing only the hot neurons that significantly contribute to the inference results [25]. Since ALISA, InfiniGen, and PowerInfer use only a portion of the KV cache or model parameters to reduce the memory footprint, they may experience some variations in language modeling output.

Optimizations for LLM inference serving. Various optimizations such as fine-grained scheduling or memory management have been applied to GPU-based LLM serving systems. Orca changes the granularity of batching from requests to iterations, reducing unnecessary computations on the GPU and improving throughput [32]. vLLM reduces GPU memory fragmentation and consequently enables larger batch sizes by applying paging to the KV cache [13]. Splitwise and DistServe allocate the prefill and generation phases to heterogeneous GPU nodes, considering the difference in computational load between the two phases [20, 37]. Sarathi-Serve introduced chunked prefill to address generation stalls and pipeline bubbles caused by scheduling prioritizing prefill phase [1]. These works mostly aim to reduce the latency of LLM inference in multi-GPU clusters.

ML frameworks with recomputation. Excessive memory capacity consumption in ML workloads can incur the underutilization of computation units and service failure due to the lack of memory. Several prior studies focused on reducing memory costs by generating intermediate results from recomputation instead of saving them in memory. These techniques have primarily been researched in DNN training and are commonly referred to as rematerialization. Checkmate reduced the GPU memory capacity required for DNN training by 5× through tensor rematerialization [9]. POET combines tensor rematerialization and paging to enable efficient DNN fine-tuning on edge devices [21].

7 Conclusion

This paper addresses the communication-computation imbalance in host memory offloading for LLM computation. It introduces a novel activation checkpointing with KV-Activation hybrid caching. It determines the best ratio of KV and Activation caches to maximize the utilization of three key resources: memory capacity, host-GPU communication bandwidth, and GPU computation capability. The source code will be publicly available after publication.

References

[1] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming {Throughput-Latency} Tradeoff in {LLM} Inference with {Sarathi-Serve}. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 117–134.

[2] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, et al. 2022. DeepSpeed Inference: Enabling Efficient Inference of Transformer Models at Unprecedented Scale. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[3] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. 2024. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 929–947.

[4] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Grechen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[5] Xinyun Chen, Petros Maniatis, Rishabh Singh, Charles Sutton, Hanjun Dai, Max Lin, and Denny Zhou. 2021. SpreadsheetCoder: Formula Prediction from Semi-structured Context. In *Proceedings of Machine Learning Research (PMLR)*.

[6] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *36th Conference on Nerual Information Processing Systems (NeurIPS)*.

[7] Hugging Face. 2022. Hugging Face Accelerate. https://huggingface.co/docs/accelerate/index

[8] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. 2024. Cost-Efficient Large Language Model Serving for Multi-turn Conversations with CachedAttention. In *Proceedings of the 2024 USENIX Annual Technical Conference (ATC)*.

[9] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Kurt Keutzer, Ion Stoica, and Joseph E. Gonzalez. 2020. Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization. In *Proceedings of 3rd Conference on Machine Learning and Systems (MLSys)*.

[10] Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hossein Nodehi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, Todd Mytkowicz, and Olli Saarikivi. 2022. Breaking the Computation and Communication Abstraction Barrier in Distributed Machine Learning Workloads. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS))*.

[11] Jinwoo Jeong, Seungsu Baek, and Jeongseob Ahn. 2023. Fast and Efficient Model Serving Using Multi-GPUs with Direct-Host-Access. In *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys)*.

[12] Jaehoon Jung, Jinpyo Kim, and Jaejin Lee. 2023. DeepUM: Tensor Migration and Prefetching in Unified Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Language and Operating Systems (ASPLOS)*.

[13] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *SOSP'23: Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*.

[14] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. 2024. InfiniGen: Efficient Generative Inference of Large Language Models with Dynamic KV Cache Management. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[15] Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, Benjamin Newman, Binhang Yuan, Bobby Yan, Ce Zhang, Christian Cosgrove, Christopher D. Manning, Christopher Ré, Diana Acosta-Navas, Drew A. Hudson, Eric Zelikman, Esin Durmus, Faisal Ladhak, Frieda Rong, Hongyu Ren, Huaxiu Yao, Jue Wang, Keshav Santhanam, Laurel Orr, Lucia Zheng, Mert Yuksekgonul, Mirac Suzgun, Nathan Kim, Neel Guha, Niladri Chatterji, Omar Khattab, Peter Henderson, Qian Huang, Ryan Chi, Sang Michel Xie, Shibani Santurkar, Surya Ganguli, Tatsunori Hashimoto, Thomas Icard, Tianyi Zhang, Vishrav Chaudhary, William Wang, Xuechen Li, Yifan Mai, Yuhihu Zhang, and Yuta Koreeda. 2022. Holistic Evaluation of Large Language Models. arXiv:2211.09110 [cs.CL]

[16] Chaofan Lin, Zhenhua Han, Chengruidong Zhang, Yuqing Yang, Fan Yang, Chen Chen, and Lili Qiu. 2024. Parrot: Efficient Serving of LLM-based Applications with Semantic Variable. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[17] Avanika Narayan, Ines Chami, Laurel Orr, Simran Arora, and Christopher Ré. 2022. Can Foundation Models Wrangle Your Data? arXiv:2205.09911 [cs.LG]

[18] Shanshi Narayan, Shay B. Cohen, and Mirella Lapata. 2018. Don't Give Me the Details, Just the Summary! Topic-Aware Convolutional Neural Networks for Extreme Summarization. arXiv:1808.08745 [cs.CL]

[19] NVIDIA. 2023. TensorRT-LLM. https://github.com/NVIDIA/TensorRT-LLM.

[20] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Inigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient Generative LLM Inference Using Phase Splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*.

[21] Shishir G. Patil, Paras Jain, Prabal Dutta, Ion Stoica, and Joseph E. Gonzalez. 2022. POET: Training Neural Networks on Tiny Devices with Integrated Rematerialization and Paging. In *Proceedings of the 39th International Conference on Machine Learning (ICML)*.

[22] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2022. Efficiently Scalaing Transformer Inference. arXiv:2211.05102 [cs.LG]

[23] 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 *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[24] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU. In *Proceedings otf the 40th International Conference on Machine Learning (ICML)*.

[25] Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. 2024. PowerInfer: Fast Large Language Model Serving with a Consumer-grade GPU. In *Proceedings of the 30th Symposium on Operating Systems Principles (SOSP)*.

[26] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard

Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL]

[27] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288 [cs.CL]

[28] Zhongwei Wan, Ziang Wu, Che Liu, Jinfa Huang, Zhihong Zhu, Peng Jin, Longyue Wang, and Li Yuan. 2024. LOOK-M: Look-Once Optimization in KV Cache for Efficient Multimodal Long-Context Inference. In *Empirical Methods in Natural Language Processing (EMNLP) Findings, 2024.*

[29] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. 2023. Efficient Streaming Language Models with Attention Sinks. arXiv:2309.17453 [cs.CL]

[30] Zhichao Xu, Ashim Gupta, Tao Li, Oliver Bentham, and Vivek Srikumar. 2024. Beyond Perplexity: Multi-dimensional Safety Evaluation of LLM Compression. In *Empirical Methods in Natural Language Processing (EMNLP) Findings, 2024.*

[31] June Yong Yang, Byeongwook Kim, Jeongin Bae, Beomseok Kwon, Gunho Park, Eunho Yang, Se Jung Kwon, and Dongsoo Lee. 2024. No Token Left Behind: Reliable KV Cache Compression via Importance-Aware Mixed Precision Quantization. arXiv:2402.18096 [cs.LG] https://arxiv.org/abs/2402.18096

[32] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. ORCA: A Distributed Serving System for Transformer-Based Generative Models. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI).*

[33] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. 2022. OPT: Open Pre-trained Transformer Language Models. arXiv:2205.01068 [cs.CL]

[34] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Rè, Clark Barret, Zhangyang Wang, and Beidi Chen. 2023. H_2O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models. arXiv:2306.14048 [cs.LG]

[35] Youpeng Zhao, Di Wu, and Jun Wang. 2024. ALISA: Accelerating Large Language Model Inference via Sparsity-Aware KV Caching. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA).*

[36] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. SGLang: Efficient Execution of Structured Language Model Programs. arXiv:2312.07104 [cs.AI] https://arxiv.org/abs/2312.07104

[37] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI).*