ReFlow: Exposing Parallelism in CPU-Offloaded LLM Training via Register-Resident SIMD Fusion and Decoupled Update Scheduling

Seongtae Bang
Daegu Gyeongbuk Institute of Science and Technology (DGIST)
Daegu, Republic of Korea
st.bang@dgist.ac.kr

Gyeongseo Park
Electronics and Telecommunications Research Institute (ETRI)
Daejeon, Republic of Korea
gspark@etri.re.kr

Ki-Dong Kang
Electronics and Telecommunications Research Institute (ETRI)
Daejeon, Republic of Korea
kd_kang@etri.re.kr

Hyunkyun Shin
Yonsei University
Seoul, Republic of Korea
hk.shin@yonsei.ac.kr

Sungju Kim
Yonsei University
Seoul, Republic of Korea
sungju.kim@yonsei.ac.kr

Daehoon Kim*
Yonsei University
Seoul, Republic of Korea
daehoonkim@yonsei.ac.kr

Abstract

CPU offloading is a practical approach to large language model training when FP32 optimizer state exceeds GPU HBM capacity. Prior work treats the CPU path as a slow fallback and hides its latency rather than reducing it. We observe that the exposed host-side bottleneck is fundamentally a host-memory-traffic problem, not a CPU-arithmetic problem, rooted in unnecessary materialization, gradient inflation, and serialized update scheduling. We present *ReFlow*, a CPU-offloaded training framework that redesigns the host update path around this diagnosis. To this end, *ReFlow* introduces register-resident optimizer fusion, host-side gradient processing, and deadline-aware update scheduling. Across the evaluated workloads, *ReFlow* improves throughput by up to 4× over the conventional baseline and 3× over the state-of-the-art in CPU-offloaded training, while preserving exact training semantics.

CCS Concepts: • **Computer systems organization** → **Heterogeneous (hybrid) systems**; • **Software and its engineering** → *Runtime environments*; *Software performance*.

Keywords: Large Language Model; CPU Offloading; Heterogeneous Memory; Efficient Deep Learning Training; CPU SIMD

ACM Reference Format:
Seongtae Bang, Gyeongseo Park, Ki-Dong Kang, Hyunkyun Shin, Sungju Kim, and Daehoon Kim. 2027. ReFlow: Exposing Parallelism in CPU-Offloaded LLM Training via Register-Resident SIMD Fusion and Decoupled Update Scheduling. In *Proceedings of the 32nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '27), April 11–15, 2027, Heraklion, Greece.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3845814.3850051

*Daehoon Kim is the corresponding author.



This work is licensed under a Creative Commons Attribution 4.0 International License.

ASPLOS '27, Heraklion, Greece
© 2027 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-3012-2/2027/04
https://doi.org/10.1145/3845814.3850051

1 Introduction

As model sizes and sequence lengths grow, Large Language Model (LLM) training is increasingly constrained by memory capacity [6, 8, 12, 13, 24, 32]. In particular, the optimizer states can be several times larger than the model parameters under mixed precision, making it difficult to keep all states in GPU memory [11, 22, 29, 47, 50]. As a result, offloading optimizer state to host DRAM has become a practical requirement when optimizer state exceeds the GPU memory budget [3, 14, 16–18, 21, 23, 30, 31, 38, 44, 46, 54].

While offloading enables training at scale, it fundamentally changes the execution model by introducing a host-side critical path between CPU and GPU. Once optimizer state resides on the host, the next iteration cannot begin until the CPU completes the update and returns updated BF16 parameters to the GPU. This dependency directly translates into GPU idle time; on our 8-GPU B200 system, it accounts for up to 47.9% of the iteration time after offloading. The host update can only be finalized after step-wide validation (e.g., overflow detection and norm clipping) resolves, which requires the complete gradient set after the backward pass [25, 29, 34, 37, 39–41]. Consequently, BF16 parameter generation is delayed until the entire host update pipeline completes, exposing it as the critical bottleneck in CPU-offloaded training. Importantly, this bottleneck does not arise from a lack of compute resources, but from the lack of parallelism in how the host update is executed. Although CPU and GPU resources are both available, the update path is structured as a serialized execution, preventing overlap between computation, communication, and update generation.

Table 1. Comparison of *ReFlow* with prior training systems (OPT-30B, 4 B200 GPUs, GA=1, batch size 32). All reductions are relative to ZeRO-Infinity. W denotes model size, and α denotes additional GPU-side optimizer memory.

System	Model-state memory across GPUs	Gradient promotion	Backward host memory traffic	Backward overhead reduction	Critical-path host memory traffic	Host update time reduction	Preserves loss	Per-GPU TFLOPS
ZeRO-Infinity [44]	$\leq 4W$	GPU	120GB	Baseline	1260GB	Baseline	Yes	366.40
Deep Optimizer States [28]	$\leq 4W + \alpha$	GPU	120GB	1.9%	~990GB	17.8%	Yes	393.50
SuperOffload [25]	$\leq 4W + \alpha$	GPU	~120GB	1.1%	~720GB	44.1%	No	438.78
ZeRO-3 [43]	$16W$	GPU	-	28.7%	-	-	Yes	681.64
ReFlow (this work)	$\leq 4W$	CPU	60GB	26.7%	~0GB	96.8%	Yes	681.18

To mitigate this exposed GPU stall, prior CPU-offloaded systems focus on reducing or hiding the portion of host delay that blocks the next iteration, implicitly treating the host path as fundamentally inefficient. Conventional offloading systems like ZeRO-Offload [46] and ZeRO-Infinity [44] execute the full optimizer update on the CPU, leaving the GPU exposed to the entire latency. Subsequent approaches attempt to hide this delay by shifting work back to the GPU or overlapping execution across devices. For example, Deep Optimizer States [28] moves part of the update to the GPU at the cost of additional computation, while SuperOffload [25] overlaps optimizer execution with the backward pass using speculation, introducing rollback overheads when validation fails. These approaches are effective at reducing visible stall, but they do not address its root cause. In particular, they assume that the bottleneck lies in CPU computation and therefore attempt to bypass the host path, while leaving intact the dominant cost of host-side memory traffic and data movement. As a result, the fundamental inefficiency of the offloaded update path remains.

Contribution-1: Identifying that the host update bottleneck arises from serialized, memory-traffic-heavy execution that limits parallelism, rather than insufficient CPU computation (§3). We analyze the host-side update path and find that its delay is not fundamentally limited by optimizer arithmetic, but by excessive memory traffic and data movement on the host. Across 2–8 B200 GPUs, these overheads are roughly 5× the cost of optimizer computation, exposing memory handling as the true bottleneck. We identify three structural sources of this overhead. First, the optimizer is decomposed into separate tensor operations, each of which materializes its output to DRAM even when the value is consumed only once by the next operation. Second, gradient handling inflates communication and memory traffic. Gradients are promoted to FP32 before transfer, which doubles PCIe payload. Under gradient accumulation, previously offloaded partial gradients must also be repeatedly retrieved from the host, further increasing traffic. Third, latency-critical BF16 parameter generation is serialized with FP32 optimizer-state writeback, even though only the BF16 parameters are required to start the next iteration.

These sources share a common root cause. The host update path is executed as a materialize-everything pipeline because conventional tensor operators offer no fine-grained control over what is materialized and when. Each operator must fully write its output before the next can begin, serializing gradient processing, optimizer computation, and parameter generation across the CPU–GPU boundary, even when their true dependencies would permit overlap. The key challenge is therefore not only reducing host-memory traffic under limited host DRAM bandwidth [1, 19], but restructuring the host update path to recover this blocked parallelism while preserving exact training semantics.

Contribution-2: Enabling parallel execution of the host update path via register-resident SIMD fusion and store-controlled decoupled update scheduling (§4). The key insight is that the host update path contains substantial parallelism that is obscured by conventional serialized execution. SIMD execution enables this restructuring by keeping intermediate values in registers until explicitly stored, eliminating unnecessary materialization, and by exposing each store as an explicit control point. This explicit-store property lets the system prioritize latency-critical BF16 parameter generation and defer FP32 optimizer-state persistence, breaking the forced coupling between the two outputs. Based on this property, we propose *ReFlow*, which restructures the host update path around register-resident execution and store-controlled scheduling.

First, we fuse the host optimizer computation into a single register-resident SIMD body, eliminating intermediate DRAM round-trips and enabling continuous execution (FlashOpt, §4.2). Second, we restructure gradient handling to eliminate repeated cross-device dependencies, allowing gradient processing to proceed independently on the CPU (CPU-Centric Gradient, §4.3). Third, we decouple outputs by their consumption deadline and schedule FP32 state updates during CPU idle time that would otherwise be wasted, allowing latency-critical BF16 parameter generation to complete first and thereby removing FP32 persistence from the critical path (Split-Phase Schedule, §4.4). Together, these mechanisms transform the host update path from a serialized execution into a parallel CPU–GPU pipeline.

Evaluation Summary: We evaluate *ReFlow* on an 8-GPU NVIDIA B200 HGX system and a 4-GPU NVIDIA A100 PCIe server using decoder-only Transformer models across a wide range of scales. *ReFlow* consistently outperforms prior CPU-offloaded systems, achieving up to 4× higher throughput than ZeRO-Infinity and 3× higher throughput than SuperOffload. On 8 B200 GPUs, *ReFlow* reaches 98% of ZeRO-3 throughput at 50B under the same hardware budget and training options, despite offloading the optimizer to the host. Under tighter memory budgets, where ZeRO-3 is forced into deeper gradient accumulation and suboptimal micro-batches, *ReFlow* surpasses ZeRO-3 by 1.2× at 80B. Importantly, *ReFlow* preserves exact training semantics even when validation requires norm clipping, matching the loss curve of ZeRO-Infinity, unlike speculative approaches that introduce accuracy deviations.

Table 1 summarizes the key differences between *ReFlow* and prior systems. *ReFlow* maintains the minimal GPU memory footprint of CPU-offloaded training while eliminating the host-path bottleneck. Unlike prior approaches that increase GPU memory usage to reduce host latency, *ReFlow* keeps the optimizer state fully offloaded while reducing both backward-phase overhead and critical-path host traffic. In particular, because our SIMD kernel performs gradient promotion efficiently on the CPU, *ReFlow* reduces both GPU-side backward overhead and host communication, bringing performance close to that of fully GPU-resident training. As a result, *ReFlow* matches the throughput of fully GPU-resident training despite requiring substantially less GPU memory. Crucially, *ReFlow* achieves these gains while preserving baseline training accuracy, unlike SuperOffload.

2 Background

2.1 CPU-offloaded mixed-precision training

Mixed-precision training accelerates execution by performing forward and backward passes in BF16 on the GPU, while maintaining the optimizer state in FP32 to preserve numerical stability [22, 29, 52]. For Adam, the persistent optimizer state consists of one FP32 master parameter and two FP32 moment tensors, requiring 12 B per parameter in total, or 6× the storage of a 2 B BF16 parameter. When this state exceeds device memory capacity, training systems adopt a CPU-offloaded configuration where the optimizer state resides in host DRAM. Consequently, the subsequent GPU forward pass is blocked until the host completes the parameter update and returns the new BF16 parameters over PCIe.

In conventional CPU-offloaded training, gradients are promoted to FP32 on the GPU and transferred to the host. After the backward pass, step-wide validation resolves loss scaling and gradient clipping yielding an effective update factor $\gamma = s^{-1}\lambda$, where s^{-1} corrects the loss scaling and λ enforces the gradient clipping [29, 44, 46]. The host update then consumes the FP32 gradient buffer and applies the finalized γ to the host update. Algorithm 1 details the conventional host update. It consists of three discrete operations: TensorScale applies γ, AdamSIMD updates the FP32 optimizer state, and TensorCastBf16 converts the updated parameter for the next forward pass. Among them, only AdamSIMD modifies iteration-persistent FP32 state, while the others produce only transient intermediates. However, because these operations execute as separate tensor operations, their boundaries force even these single-use values to be materialized in DRAM, inflating host-memory traffic.

Algorithm 1 Conventional host update.

Require: FP32 gradients g, FP32 optimizer state $(p,\ m_1,\ m_2)$, factor γ

1: $g_{\text{scaled}} \leftarrow \textsc{TensorScale}(g,\ \gamma)$
2: $(p',\ m_1',\ m_2') \leftarrow \textsc{AdamSIMD}(g_{\text{scaled}},\ p,\ m_1,\ m_2)$
3: $p_{\text{bf16}} \leftarrow \textsc{TensorCastBf16}(p')$
4: $\textsc{SendToGPU}(p_{\text{bf16}})$

Algorithm 2 Conventional SIMD Adam kernel.

1: **for** $i = 0$ **to** N **step** VL **do**
    Vector loads
2: $(g,\ p,\ m_1,\ m_2) \leftarrow \text{VLOAD}(grad,\ param,\ state1,\ state2)$
    Register-resident arithmetic
3: $(p',\ m_1',\ m_2') \leftarrow \textsc{update}(g,\ p,\ m_1,\ m_2)$
    Stores only iteration-persistent state
4: $\text{VSTORE}(param,\ state1,\ state2 \leftarrow p',\ m_1',\ m_2')$
5: **end for**

2.2 SIMD execution and host-memory traffic

The distinction between discrete tensor operations and SIMD kernels determines how intermediate values interact with host memory. A tensor operation executes as an independent pass over memory-resident tensors. It fetches inputs from DRAM, computes the result, and writes the output back to host memory. Consequently, every boundary between tensor operations explicitly materializes a tensor in host memory, converting even single-use values into heavy memory traffic. Conversely, a SIMD kernel processes data within vector registers. Intermediate values computed within a single SIMD body are forwarded directly between instructions without interacting with the memory hierarchy. As a result, whether a value becomes host-memory traffic is determined by explicit stores, not by operation boundaries. A computed value becomes host-memory traffic only when the kernel issues an explicit store instruction (`VSTORE`) [2, 20]. Algorithm 2 illustrates this register-resident execution. AdamSIMD loads the gradient and state vectors, executes the arithmetic logic within the vector registers, and issues `VSTORE` only for the updated persistent tensors (p', m_1', and m_2'). While modern CPU-offloaded systems utilize this SIMD property for the core optimizer arithmetic [25, 46], gradient scaling and BF16 conversion remain outside this SIMD boundary as separate,

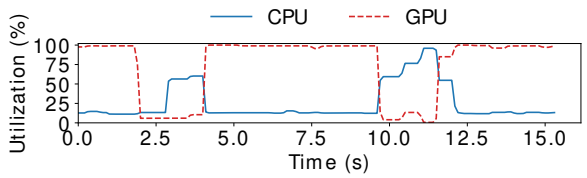


Figure 1. Utilization of ZeRO-Infinity on OPT-30B with 8 GPUs and batch size 64.

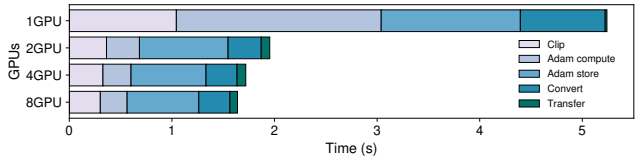


Figure 2. Host-side update time breakdown on B200s with asynchronous transfer for OPT-30B at per-GPU batch size 8.

memory-bound tensor passes (Algorithm 1). Because intermediate values across these operational boundaries must be written to and read from DRAM, these boundaries form the basis of the host-memory traffic analyzed in Section 3.1.

3 Opportunities

Existing CPU-offloaded architectures place the host-side update directly on the iteration critical path. As Figure 1 shows for ZeRO-Infinity, the CPU remains largely underutilized during the forward and backward passes, yet its update still gates the next iteration. At the iteration boundary, the GPU therefore idles while waiting for updated BF16 parameters from the host. The resulting iteration time is $T_{\text{iter}}^{(i)} \approx T_{\text{gpu}}^{(i)} + T_{\text{host}}^{(i)}$, where $T_{\text{host}}^{(i)}$ denotes the host-side work that must complete before iteration i+1 begins.

To isolate the intrinsic overhead of the host update path, we profile the baseline with asynchronous transfers enabled. Figure 2 shows that core Adam arithmetic is not the dominant component of $T_{\text{host}}^{(i)}$. Across 2 to 8 GPUs, computation accounts for only 15.9% to 16.4% of the host delay, while store instructions alone take 2.7× longer than the arithmetic. When combined with surrounding tensor operations such as gradient clipping, precision conversion, and transfer scheduling, the non-arithmetic overhead constitutes over 80% of the total host delay, which is approximately 5× the actual Adam compute time. This result indicates that the exposed host-side bottleneck is fundamentally a data-movement problem, motivating the three architectural opportunities outlined below.

Table 2. End-to-end throughput (GParams/s) for processing 1B parameters. For BF16→FP32, the arrow includes transfer: $_G$ denotes GPU-side promotion before transfer, and $_C$ denotes CPU-side promotion after BF16 transfer.

Operation	Torch	AVX (Store)	AVX (No Store)
BF16→FP32$_G$	13.6	–	–
BF16→FP32$_C$	1.6	14.3	22.8
Optimizer$_{32}$	–	9.6	12.1
Optimizer$_{16}$	–	12.2	16.2

3.1 Unnecessary materialization at operator boundaries

A significant portion of host-memory traffic originates at tensor-operation boundaries. Algorithm 1 decomposes the conventional host update into three steps: TensorScale, AdamSIMD, and TensorCastBf16. Only AdamSIMD updates the iteration-persistent FP32 optimizer state. TensorScale and TensorCastBf16 handle only transient values. This split creates two forms of critical-path host-memory traffic: traffic required to produce the updated state and parameter, and boundary-induced overhead. The former is unavoidable. Per parameter, the host reads the FP32 optimizer state (12 B) and gradient (4 B), then writes the updated FP32 optimizer state (12 B) and BF16 parameter (2 B). Thus, $B_{\text{req}}^{\text{base}} = 30$ B. The latter arises only because intermediate values are forced to cross DRAM at operation boundaries. TensorScale writes the scaled gradient (4 B) to DRAM, and AdamSIMD rereads it, adding 8 B. TensorCastBf16 then rereads the updated FP32 parameter (4 B) to produce the BF16 parameter, adding another 4 B. Thus, $B_{\text{int}}^{\text{base}} = 12$ B, and the total per-parameter critical-path host-memory traffic is $B_{\text{host}}^{\text{base}} = B_{\text{req}}^{\text{base}} + B_{\text{int}}^{\text{base}} = 30\,\text{B} + 12\,\text{B} = 42\,\text{B}$. This is a materialize-everything execution pattern in which operation boundaries force even single-use values to cross DRAM.

Summary: Discrete tensor operations force single-use values to be materialized in DRAM, introducing unnecessary traffic on the critical path.

3.2 Gradient traffic amplification

The host update is further bottlenecked by amplified gradient traffic. This amplification is driven by two factors: the precision of transferred gradients and their repeated movement across the PCIe boundary. Conventional systems inflate this traffic by prematurely promoting gradients to FP32 and by anchoring accumulation logic on the GPU. As a result, early FP32 promotion doubles both the PCIe payload and the gradient load traffic, while GPU-resident accumulation forces gradients to repeatedly cross the PCIe bus across micro-steps.

Table 3. GPU stall from gradient accumulation on OPT-30B with 8 GPUs and batch size 64.

GA depth (A)	# Aggregations	Avg. stall (ms)	Per Agg. (ms)
1	0	0	0
2	1	1063	1063
4	3	3188	1063
8	7	7439	1063
16	15	15942	1063

3.2.1 Premature gradient promotion.

Conventional CPU-offloaded systems [25, 28, 44, 46] prematurely promote gradients to FP32 before the device-to-host transfer, inflating both PCIe traffic and host-side memory load. This relies on the assumption that GPU-side promotion is necessary to compensate for the slow arithmetic of host CPUs. However, our profiling (Table 2) shows that this assumption no longer holds for optimized CPU implementations. While a naive Torch-based CPU promotion achieves only 1.6 GParams/s, our SIMD implementations (e.g., AVX) reach 14.3 GParams/s, outperforming the conventional GPU-side promotion-and-transfer path (13.6 GParams/s). Furthermore, directly feeding BF16 gradients to the optimizer (Optimizer_{16}) achieves 12.2 GParams/s, compared to 9.6 GParams/s for FP32 inputs (Optimizer_{32}), indicating that early promotion both doubles host-side gradient load traffic and reduces optimizer efficiency.

Summary: Premature FP32 promotion unnecessarily doubles PCIe traffic and host-side gradient load, slowing the host update.

3.2.2 Redundant PCIe crossing during gradient accumulation.

Gradient accumulation introduces repeated cross-device traffic in CPU-offloaded training. Although intermediate gradients are stored on the host, accumulation remains GPU-resident, forcing gradients to repeatedly cross the PCIe bus. Specifically, each new micro-step requires retrieving previously offloaded gradients from the host to perform aggregation on the GPU. For example, at an accumulation depth of $A = 4$, micro-steps 2, 3, and 4 each trigger a blocking host-to-device transfer. As Table 3 shows, each retrieval incurs a fixed delay of 1,063 ms, which accumulates linearly with the number of micro-steps. As a result, the total communication stall scales as $T_{\text{GA-stall}}(A) \approx (A-1) \cdot T_{\text{agg}}$, exposing gradient accumulation as a major source of PCIe overhead.

Summary: GPU-resident accumulation forces unnecessary PCIe crossings, introducing a communication stall that grows linearly with accumulation depth.

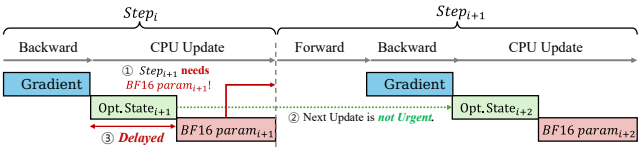


Figure 3. Different deadlines across outputs.

3.3 Coupled update outputs with mismatched deadlines

The host update generates two outputs with inherently different deadlines: *Readiness*, the BF16 parameter needed immediately to start the next forward pass, and *Persistence*, the FP32 optimizer state required only for the next host update. Despite this asymmetry, conventional systems couple both outputs into a single serialized update phase. As a result, latency-critical *Readiness* is delayed by the heavier *Persistence*, unnecessarily extending the iteration critical path. This rigid coupling constrains the execution schedule and prevents the system from exploiting the natural slack between these two outputs.

3.3.1 Output-side deadline mismatch: *Persistence* delays *Readiness*.

Although the host update produces two outputs, they have fundamentally different deadlines. *Readiness* (BF16 parameters) is required immediately to start the next iteration, whereas *Persistence* (FP32 optimizer state) is needed only for the subsequent host update. As shown in Figure 3, iteration i+1 requires the BF16 parameters from step i, while the corresponding optimizer state can be deferred. Despite this asymmetry, conventional systems couple both outputs into the same update path, forcing *Readiness* to wait for *Persistence*. This coupling is problematic because *Persistence* dominates the update cost and determines the overall latency. While *Readiness* requires only a lightweight 2 B write per parameter, *Persistence* incurs a heavy 12 B FP32 state commit to host DRAM. Given the limited host memory bandwidth, this memory-intensive *Persistence* significantly delays the entire update and extends the iteration critical path. Table 2's Optimizer_{16} further confirms that memory traffic, rather than computation, dominates this cost: removing FP32 stores from the SIMD path increases throughput from 12.2 to 16.2 GParams/s, showing that *Persistence* drives the critical-path latency.

Summary: Coupling outputs with different deadlines forces latency-critical BF16 parameters to wait for heavyweight FP32 state writes, extending the critical path.

3.3.2 Input-side dependency: validation delays *Readiness*.

Conventional execution unnecessarily delays BF16 parameter generation by treating validation as a strict barrier. As discussed in Section 2.1, step-wide validation determines the global clipping factor λ, and existing systems wait for

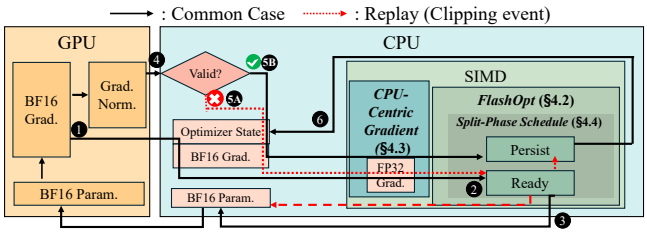
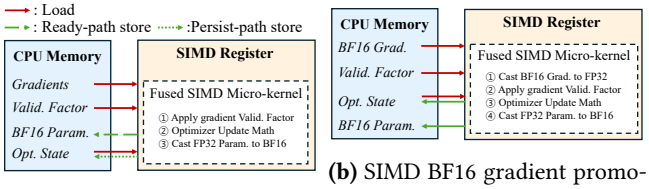


Figure 4. *ReFlow* overview.



(a) Register-resident update path (FlashOpt).

(b) SIMD BF16 gradient promotion (CPU-Centric Gradient).

Figure 5. SIMD-fused body.

this step to complete before generating BF16 parameters. However, when clipping does not occur ($\lambda = 1.0$), the BF16 parameter is already fully determined by the arriving gradients and the stored FP32 optimizer state, both of which are available before validation completes. In this case, BF16 parameters could be generated immediately, but conventional systems still delay them due to the enforced validation barrier. This shows that the dependency between validation and *Readiness* is overly conservative and not required by the underlying data flow. In contrast, only the final FP32 optimizer-state commitment truly depends on validation, as its value cannot be fixed until λ is determined.

Summary: Validation is unnecessarily treated as a barrier, delaying BF16 parameter generation beyond its true data dependency.

Taken together, these opportunities reveal a core challenge: how to minimize critical-path host-memory traffic, both in volume and when it is incurred, under limited host DRAM bandwidth while preserving training semantics.

4 Design

4.1 Overview

We propose *ReFlow*, a CPU-offloaded training framework that accelerates the host optimizer step by reducing host-memory traffic and shortening the host-visible critical path. The design is organized around three mechanisms (FlashOpt, CPU-Centric Gradient, and Split-Phase Schedule), each targeting a different structural bottleneck identified in Section 3. FlashOpt (Section 4.2) fuses the optimizer update into a single register-resident SIMD body, eliminating unnecessary intermediate materialization on the host. CPU-Centric Gradient (Section 4.3) keeps gradients compact across the GPU–CPU boundary and promotes them to FP32 only at the point of use. Split-Phase Schedule (Section 4.4) separates latency-critical BF16 parameter generation from deferrable FP32-state persistence, keeping only deadline-critical work on the iteration critical path. Together, these mechanisms turn the conventional monolithic host update into a traffic-efficient, deadline-aware pipeline.

Algorithm 3 FlashOpt

Require: FP32 gradient partition g_{in}, FP32 optimizer state (p, m_1, m_2), effective factor $\gamma = s^{-1}\lambda$, pass mode $\rho \in \{\text{Ready}, \text{Persist}\}$

1: $(\hat{g}, \hat{p}, \hat{m}_1, \hat{m}_2) \leftarrow \text{VLOAD}(g_{\text{in}}, p, m_1, m_2)$
2: $\hat{g} \leftarrow \hat{g} \cdot \gamma$
3: $(\hat{p}', \hat{m}_1', \hat{m}_2') \leftarrow \text{update}(\hat{g}, \hat{p}, \hat{m}_1, \hat{m}_2)$
4: **if** $\rho = \text{Ready}$ **then**
5: $\quad \hat{p}_{\text{bf16}} \leftarrow \text{cast_bf16}(\hat{p}')$
6: $\quad \text{VSTORE}(p_{\text{bf16}} \leftarrow \hat{p}_{\text{bf16}})$
7: **else**
8: $\quad \text{VSTORE}(p, m_1, m_2 \leftarrow \hat{p}', \hat{m}_1', \hat{m}_2')$
9: **end if**

Figure 4 illustrates the overall execution flow of *ReFlow*, with steps ❶–❻ labeling the major data movements and control decisions. BF16 gradients are first transferred from the GPU to the CPU ❶. The CPU then performs the host-side update ❷. It generates a BF16 parameter for the next forward pass ❸. In parallel, the GPU runs the backward pass, computes the gradient norm, and performs validation once the full norm becomes available ❹. If the step is accepted 5B, the generated BF16 parameters are used directly for the next forward pass, and the FP32 optimizer state is written back in the background ❻. If clipping is required 5A, *ReFlow* discards the provisional outputs (i.e., BF16 parameters) and re-executes only the lightweight Ready path, as denoted by the red dotted arrows.

4.2 FlashOpt

FlashOpt eliminates the operator-boundary memory traffic identified in Section 3.1. The key idea of FlashOpt is to fuse the optimizer update into a single SIMD computation. As a result, intermediate values remain in vector registers throughout the update instead of being materialized in memory at each step. Thus, intermediate states used only within the update no longer generate avoidable host-memory traffic.

Algorithm 3 and Figure 5a illustrate this execution model. FlashOpt issues explicit SIMD instructions to load the gradients and the FP32 optimizer state directly into vector registers. It then scales the gradients, and applies the Adam arithmetic to the resident FP32 state. As a result, the host

path no longer incurs redundant memory traffic to materialize scaled gradients before the optimizer step, nor does it unnecessarily reread stored FP32 parameters merely to cast them to BF16. Any intermediate value whose lifecycle ends within the fused body remains register-resident.

Crucially, FlashOpt leverages SIMD's explicit-store semantics to split the conventional monolithic update output into separate Ready and Persist outputs. As depicted by the branch condition (ρ) in Algorithm 3, the underlying update arithmetic remains identical across both modes, but each mode stores only its targeted output. In Ready mode, FlashOpt stores only the lightweight BF16 parameters, removing parameter generation from the post-validation critical path. In Persist mode, it stores only the memory-intensive FP32 optimizer state to host DRAM. This decoupling of update arithmetic from final stores isolates the latency-critical path and provides the foundation for the scheduling techniques in Section 4.4.

4.3 CPU-Centric Gradient

CPU-Centric Gradient minimizes communication overhead and host-memory traffic by maintaining gradients in BF16 format across the PCIe boundary. As illustrated in the transition from Figure 5a to Figure 5b, this mechanism extends the FlashOpt datapath to directly ingest BF16 inputs, deferring FP32 promotion until the exact SIMD instruction that computes the update. By eliminating premature gradient expansion, this approach keeps the host input stream compact and avoids unnecessary data materialization. It therefore reduces PCIe payload inflation and eliminates GPU cycles previously wasted on pre-transfer casting. CPU-Centric Gradient realizes this optimized datapath through two key mechanisms: point-of-use precision promotion and CPU-resident gradient accumulation.

4.3.1 BF16 transfer and CPU promotion. Conventional systems promote gradients to FP32 on the device before transfer, doubling both the PCIe payload and the host-side gradient load traffic. This also forces the host to materialize transient gradients in FP32, only to reread them into the CPU SIMD pipeline. *ReFlow* eliminates this inefficiency by transferring each partition in compact BF16 form and deferring FP32 promotion until the point of consumption within the FlashOpt vector registers. Because FlashOpt already fuses the host update into a single register-resident SIMD body, this compact gradient path avoids unnecessary memory materialization. Furthermore, Table 2 shows that ingesting native BF16 gradients and promoting them on-the-fly within SIMD registers yields higher optimizer throughput than processing pre-materialized FP32 inputs.

This point-of-use promotion also avoids unnecessary GPU computation. By leaving gradients in BF16, *ReFlow* eliminates GPU-side pre-transfer promotion, allowing the GPU to focus on backward computation. Combined with the 12 B per-parameter reduction achieved by FlashOpt, retaining gradients in BF16 until point-of-use removes an additional 2 B from the exposed host datapath, reducing cumulative critical-path traffic to 28 B per parameter. The resulting device-to-host transfers proceed asynchronously and overlap with the concurrent GPU backward pass.

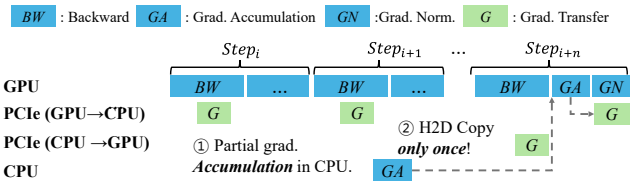
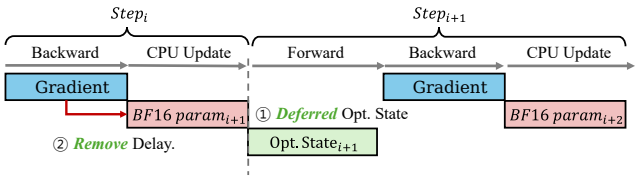


Figure 6. CPU-resident accumulation.

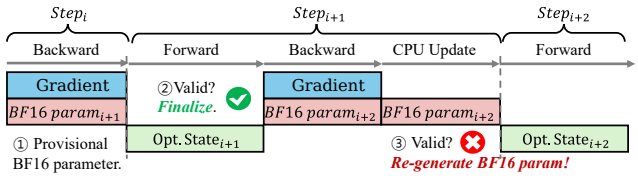
4.3.2 CPU-resident accumulation. CPU-resident accumulation eliminates redundant PCIe traversals by accumulating gradients on the host. *ReFlow* retains the first offloaded partition in host memory and accumulates subsequent intermediate partial gradients using a lightweight SIMD kernel. By processing compact BF16 operands through a register-resident SIMD pipeline similar to FlashOpt, this CPU-side accumulation runs efficiently without incurring host-memory bottlenecks. Since this operation is scheduled within the available CPU idle cycles during the backward pass (Figure 1), the accumulation process removes memory traffic from the critical path. As illustrated in Figure 6, intermediate micro-steps therefore no longer trigger blocking host-to-device retrievals. The resulting traffic becomes a one-way stream from the GPU to the CPU during intermediate micro-steps. At the terminal micro-step (step_{i+n}), the accumulated host-side partition is returned to the GPU exactly once, where the GPU performs the final addition and computes the gradient norm in the same phase. This placement preserves data locality by co-locating the final accumulation with the norm computation, after which the finalized gradient partition is transferred back to the host to sustain the FlashOpt pipeline. Combined with BF16 transfers and deferred precision promotion, this approach eliminates the linear communication stall identified in Section 3.2.2. With boundary communication optimized, the remaining bottleneck is the serialized post-validation schedule, which we address via Split-Phase Schedule.

4.4 Split-Phase Schedule

After FlashOpt and CPU-Centric Gradient minimize the intermediate and input traffic, the residual bottleneck is the conventional post-validation output phase. It serializes both the BF16 parameters and the FP32 optimizer state on the critical path, even though only the BF16 parameters are immediately needed by the GPU. Split-Phase Schedule removes



(a) Deferred FP32-state persistence.



(b) Provisional BF16 parameter.

Figure 7. Split-Phase Schedule mechanism.

this coupling by scheduling each output according to its respective deadline. FP32-state persistence is deferred to the CPU background window that opens once the GPU resumes, and provisional BF16 parameter generation is overlapped with the backward pass.

4.4.1 Deferred FP32-state persistence. Once the CPU delivers the BF16 parameters to the GPU, the GPU immediately begins its next forward pass while the CPU is otherwise idle. *ReFlow* exploits this idle interval, $S^{(i)}_{\text{cpu}}$ (from BF16 parameter delivery to the next optimizer step), to run the memory-intensive Persist path, hiding its 12 B-per-parameter FP32-state write cost from the critical path. As Figure 7a illustrates, the GPU-visible host stall shrinks to $T^{(i)}_{\text{ready}}$ alone. FP32-state persistence runs in the background and is completely hidden from the iteration critical path whenever it fits within $S^{(i)}_{\text{cpu}}$, with any excess Persist time spilling back onto the critical path. Formally, for step i, let $T^{(i)}_{\text{ready}}$ and $T^{(i)}_{\text{persist}}$ denote the execution times of the Ready and Persist paths:

$$T^{(i)}_{\text{ready}} = T^{(i)}_{\text{FlashOpt,ready}} + 2N/\beta_{\text{hw}} + 2N/\beta_{\text{H2D}},$$
$$T^{(i)}_{\text{persist}} = T^{(i)}_{\text{FlashOpt,persist}} + 12N/\beta_{\text{hw}},$$

where N is the number of parameters, β_{hw} is the effective host-memory bandwidth, and β_{H2D} is the effective host-to-device PCIe bandwidth. $S^{(i)}_{\text{cpu}}$ denotes the CPU background window from BF16 parameter delivery to the next optimizer step. The exposed host delay and the corresponding latency reduction (ΔT) are then given by:

$$T^{\text{split},(i)}_{\text{host}} \approx T^{(i)}_{\text{ready}} + \max\left(T^{(i)}_{\text{persist}} - S^{(i)}_{\text{cpu}},\ 0\right),$$
$$\Delta T^{(i)}_{\text{split}} \approx \min\left(T^{(i)}_{\text{persist}},\ S^{(i)}_{\text{cpu}}\right). \quad (1)$$

Deferred persistence requires the step-i gradients to remain in memory after the Ready path finishes. The Persist path re-runs the FlashOpt kernel with the same gradients to write the updated FP32 optimizer state to DRAM. At the same time, the step-$(i{+}1)$ gradients are already arriving, requiring an available slot to receive them. While a naive approach would allocate an extra BF16-sized buffer, *ReFlow* avoids this overhead entirely.

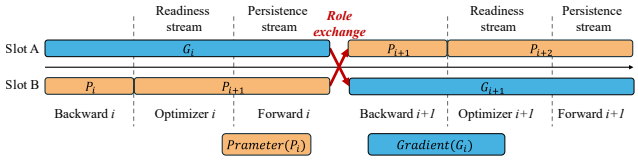


Figure 8. Double-buffer slot assignment over time.

The training pipeline already contains exactly two BF16-sized buffers, one for the incoming gradient and one for the outgoing BF16 parameter. Their live ranges do not overlap. The parameter buffer is freed as soon as the BF16 outputs are transferred to the GPU, whereas the gradient buffer is held only until the deferred Persist path completes. Since these releases occur at different times, the two buffers alternate roles each step without requiring additional memory allocation. One retains the step-i gradients for deferred persistence. The other receives the step-$(i{+}1)$ gradients, and once persistence frees the first slot, that slot accepts the step-$(i{+}1)$ BF16 outputs. Figure 8 illustrates this alternation.

4.4.2 Provisional BF16 Parameter Generation. Even with deferred persistence, $T^{(i)}_{\text{ready}}$ remains exposed as long as BF16 parameter generation follows validation. To eliminate this delay, *ReFlow* must decouple BF16 parameter generation from FP32-state commitment. The challenge is that BF16 parameters must be generated before validation resolves to hide their latency. The FP32 optimizer state, however, must not be modified until the final update factor is known. In the conventional update path, these two actions are coupled. Generating the BF16 parameters also writes the FP32 optimizer state to DRAM, so early execution would require rollback when validation changes or rejects the update.

ReFlow resolves this through the Ready/Persist output split of FlashOpt (Figure 7b). During the backward pass, *ReFlow* runs the Ready path. It computes the Adam update in registers and stores only the BF16 parameter, leaving the FP32 optimizer state in DRAM unmodified. Because the FP32 state has not been committed, these provisional BF16 parameters can be forwarded to the GPU before validation determines the final γ. Their use by the next forward pass, however, remains gated by validation. The Persist path, which commits the FP32 optimizer state, runs only after validation provides the finalized γ.

This separation allows the Ready path to overlap with the backward pass. When the backward pass is long enough to cover $T^{(i)}_{\text{ready}}$, BF16 parameters are ready before the pass ends. Accepted steps therefore incur no exposed Ready delay.

More generally, if $T^{(i)}_{\text{bwd}}$ denotes the available overlap window, the exposed host time is:

$$T^{\text{accept},(i)}_{\text{host}} \approx \left[T^{(i)}_{\text{ready}} - T^{(i)}_{\text{bwd}}\right]_+ + \left[T^{(i)}_{\text{persist}} - S^{(i)}_{\text{cpu}}\right]_+,$$
$$T^{\text{clip},(i)}_{\text{host}} \approx T^{(i)}_{\text{ready}} + \left[T^{(i)}_{\text{persist}} - S^{(i)}_{\text{cpu}}\right]_+.$$

If the step is accepted ($\lambda = 1.0$), the provisional BF16 parameters are used directly and FP32-state persistence proceeds in the background. If clipping is required ($\lambda < 1.0$), *ReFlow* discards the provisional output and replays the Ready path with the finalized $\gamma = s^{-1}\lambda$ (Figure 7(b)). Since the FP32 state remains unmodified, the replay regenerates only the BF16 parameters without requiring state rollback. The Persist pass then commits the FP32 state using the same finalized γ. A barrier is required only if this pass remains pending at the next optimizer step. If validation detects overflow, *ReFlow* discards the provisional output and skips the optimizer update. The BF16 parameters are restored from the unchanged FP32 master weights, and the loss scale is updated as in the baseline.

Even if every valid step requires clipping, the replay cost is bounded by $T^{(i)}_{\text{ready}}$. Because FlashOpt and CPU-Centric Gradient have already shortened this path below the conventional serialized update, even this worst case completes faster than the baseline:

$$T^{\text{clip},(i)}_{\text{host}} < T^{\text{base},(i)}_{\text{host}}.$$

Together, deferred persistence and provisional BF16 generation restructure the conventional serialized update into a deadline-aware pipeline. FP32-state persistence is absorbed into the CPU background window, while provisional BF16 parameter generation overlaps the backward pass. Because committed FP32 state is modified only after validation resolves, *ReFlow* preserves the baseline mixed-precision update semantics.

5 Evaluation

5.1 Experimental Methodology

Hardware platforms. We evaluate *ReFlow* on two platforms. The first is an 8-GPU NVIDIA B200 HGX [35, 36] system with two AMD EPYC 9365 CPUs, 1.6 TB of application memory, and a 400 GB shared-memory budget. The second is a 4-GPU NVIDIA A100 PCIe [33] server, equipped with two AMD EPYC 9354 CPUs and 512 GB of host memory. The GPUs are connected by NVLink within pairs, while inter-pair communication traverses PCIe. We report results across 1, 4, and 8 GPUs. This scales up to the maximum capacity supported by the target platform and model size.

Workloads. Our throughput evaluation covers GPT and LLaMA-style [42, 49, 50] decoder-only Transformers [51].

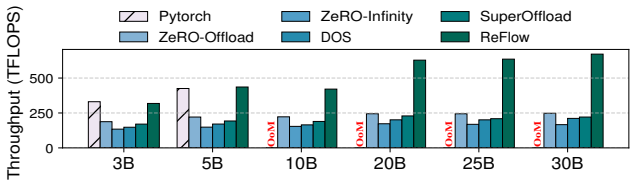
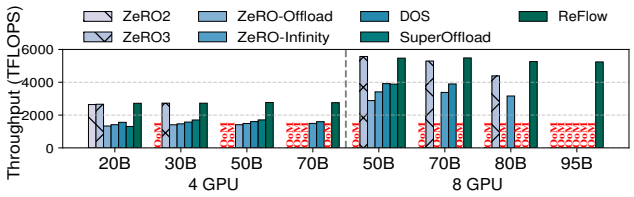


Figure 9. Throughput on a single B200 GPU.

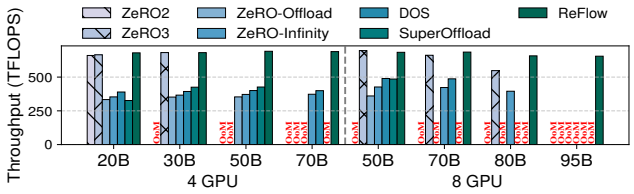
Following prior work [25], we use native OPT-30B and LLaMA-3-70B [26] at 30B and 70B and scaled variants for the remaining sizes. To isolate architectural effects, our mechanism-focused studies use OPT-30B [53] as a fixed representative workload. Unless otherwise stated, we use BF16 mixed precision with AdamW [22, 27] and a global batch size of $8G$ on G GPUs.

Baselines. We compare *ReFlow* against representative GPU-only, sharded, and CPU-offloaded baselines. These include PyTorch with DeepSpeed [45], ZeRO-2 [43], ZeRO-3 [43], ZeRO-Offload [46], ZeRO-Infinity [44], Deep Optimizer States (DOS) [28], and SuperOffload [25]. ZeRO-2 and ZeRO-3 are excluded from single-GPU experiments because they yield no distributed benefits without parallelism. PyTorch is omitted from multi-GPU experiments because it lacks the necessary distributed sharding mechanisms. All DeepSpeed-based systems run on DeepSpeed v0.18.1 with NVMe offloading disabled. This ensures that measured differences reflect system architecture rather than storage bandwidth [9, 45]. ZeRO-Offload and ZeRO-Infinity use the implementations from the public DeepSpeed release to represent highly optimized state-of-the-art baselines. For SuperOffload, we adapt the released implementation [9] to our x86 platforms. We replace its architecture-specific routines with the x86 ZeRO CPUAdam implementation [9, 25, 44, 46]. This guarantees a fair comparison using the most optimized vectorized Adam available on the target platform.

Fairness and tuning. All systems use the same model architectures, optimizers, precision formats, and input pipelines. Each baseline is evaluated under the highest-throughput configuration that fits within the hardware budget. When a target batch setting exceeds memory capacity, we consider two standard adjustments. The first is gradient accumulation with a reduced per-GPU micro-batch. The second is activation checkpointing [4] with the largest feasible micro-batch. We then report the best configuration. All CPU-offloaded baselines share the same host-side settings, including NUMA pinning and pinned-memory policies. We tune machine-dependent parameters alongside other baseline-specific configurations. Unless otherwise noted, we report the highest stable aggregate effective TFLOPS achieved for each system. Table 5 reports per-GPU TFLOPS. When activation checkpointing is enabled, we exclude recomputation volume. This ensures TFLOPS reflects effective training throughput.



(a) Aggregate throughput.



(b) Per-GPU throughput.

Figure 10. Throughput on multiple B200 GPUs.

5.2 Training Throughput

Figures 9 and 10 report throughput on the B200 GPUs. Figure 11 reports the corresponding results on A100 GPUs.

Single-B200 GPU. A single-GPU setup isolates the offloaded update path from distributed communication overheads. This directly reveals the architectural efficiency of the host-side update. Across all model sizes, *ReFlow* delivers the highest throughput among CPU-offloaded baselines. At 30B, it improves throughput by 4× over ZeRO-Infinity, 3.2× over DOS, and 3× over SuperOffload. ZeRO-Infinity retains the full host-side update cost. While DOS and SuperOffload attempt to mitigate the exposed delay by shifting computation back to the GPU, this strategy is severely limited on a single GPU. The limitation arises from PCIe bandwidth constraints and the ensuing computational burden placed on the device. *ReFlow*, in contrast, resolves the bottleneck within the offloaded path itself. This enables efficient concurrent CPU and GPU execution without reintroducing GPU-side overheads.

Multi-B200 GPUs. Figure 10 reports aggregate and per-GPU throughput on 4 and 8 B200 GPUs. *ReFlow* increases aggregate throughput while maintaining high per-GPU efficiency across GPU counts. As model size grows, prior CPU-offloaded systems accumulate three penalties: inflated gradient traffic, redundant device-side promotion, and serialized BF16 parameter generation. By eliminating these overheads, *ReFlow* reaches 98% of ZeRO-3 throughput at 50B on 8 GPUs. At 80B, where memory pressure forces ZeRO-3 to use deeper gradient accumulation and smaller micro-batches, *ReFlow* surpasses ZeRO-3 by 1.2×. Constrained by memory, ZeRO-3 must trade throughput for capacity by falling back to deeper accumulation as models grow. Prior CPU-offloaded baselines also encounter capacity limits. SuperOffload is already OOM at 70B, while DOS becomes OOM at 80B. In contrast, *ReFlow* scales to 95B by eliminating the standalone FP32 gradient stage. Section 5.5 examines this footprint reduction and the capacity limits of SuperOffload and DOS.

Table 4. Per-step time breakdown across B200 GPU counts for OPT-30B at GA=1. D2D denotes inter-GPU collective communication, including the cross-GPU reduction for global gradient-norm computation. Comm denotes CPU–GPU gradient and parameter transfers (D2H+H2D); Idle denotes GPU time spent waiting for the host.

	ReFlow (%)				ZeRO-Infinity (%)			
GPUs	Comp.	D2D	Comm	Idle	Comp.	D2D	Comm	Idle
1	96.1	0.0	1.3	2.7	20.9	0.0	31.2	47.9
4	96.4	1.3	0.5	1.9	49.6	2.8	20.0	27.6
8	94.5	2.7	0.4	2.3	56.1	4.2	11.0	28.7

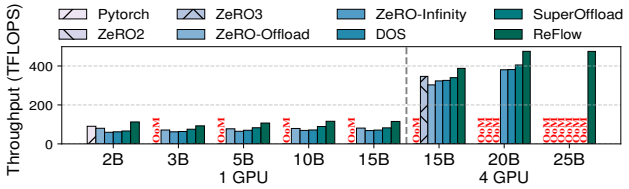


Figure 11. Throughput on A100 GPUs.

Per-GPU scaling behavior. *ReFlow* maintains nearly constant per-GPU throughput across GPU counts, with only a small drop from 4 to 8 GPUs, whereas ZeRO-Infinity improves with scale. Table 4 explains this difference through their distinct bottlenecks. For *ReFlow*, the host path is already largely hidden at one GPU, whereas ZeRO-Infinity benefits from reduced per-GPU optimizer and gradient traffic as ZeRO sharding increases. *ReFlow* is compute-bound from the start: computation accounts for 96.1% of step time at one GPU, while Comm and Idle account for only 1.3% and 2.7%, respectively. The only component that increases with scale is D2D communication, which increases from 0.0% to 1.3% and 2.7% at 1, 4, and 8 GPUs, respectively. This increase, dominated by the cross-GPU gradient-norm reduction, explains the small reduction in per-GPU throughput from 4 to 8 GPUs. ZeRO-Infinity exhibits the opposite trend. Because ZeRO shards optimizer and gradient state across GPUs, its per-GPU CPU–GPU communication decreases with scale, with Comm falling from 31.2% at one GPU to 11.0% at eight GPUs. However, the serialized host update remains on the critical path. Accordingly, Idle falls from 47.9% to 27.6% between one and four GPUs but remains at 28.7% on eight GPUs even as Comm continues to decrease.

A100 GPUs. This performance trend persists on the A100 platform. On a single GPU at 15B, *ReFlow* achieves 1.7×, 1.6×, and 1.4× higher throughput than ZeRO-Infinity, DOS, and SuperOffload, respectively. On 4 GPUs, it maintains a 1.3× and 1.2× throughput advantage at 20B over ZeRO-Infinity

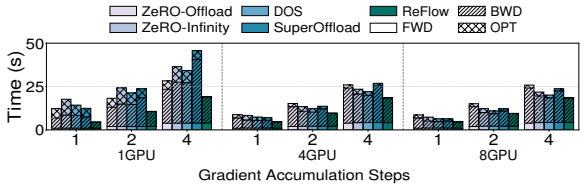


Figure 12. Step-time breakdown under gradient accumulation.

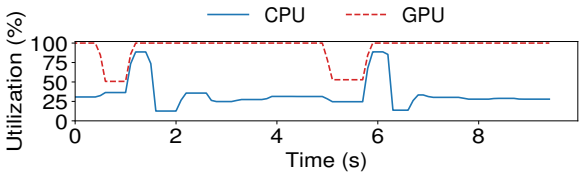


Figure 13. Average utilization for OPT-30B on 8 GPUs.

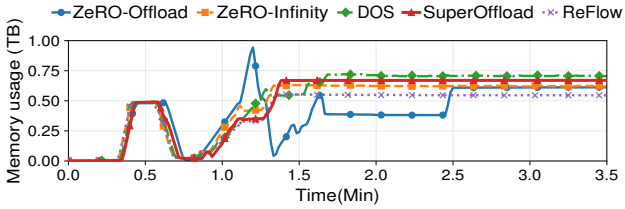


Figure 14. CPU-memory usage over time for OPT-30B on 8 GPUs.

and DOS, respectively, and extends the model-size limit from 20B to 25B. These results indicate that the exposed update path remains the dominant bottleneck, rather than the nominal bandwidth of a particular platform.

5.3 Gradient Accumulation and Phase Breakdown

We use OPT-30B with a per-GPU micro-batch size of 8 and vary the accumulation depth (GA=1, 2, and 4) across 1, 4, and 8 GPUs. Figure 12 breaks step time into forward (FWD), backward (BWD), and update (OPT) phases. Because FWD remains nearly unchanged across configurations, the impact of gradient accumulation appears primarily in the BWD phase and the exposed update tail.

Prior systems do not reduce both the BWD phase and the exposed update tail simultaneously. ZeRO-Offload lowers single-GPU step time but inflates the multi-GPU BWD phase by up to 46.9%. DOS overlaps some communication, yet reduces BWD by at most 8.3%. SuperOffload shortens OPT, but because its gradient-transfer path differs from the baseline, accumulation-handling overhead grows with GA and prevents it from reducing the accumulation-driven BWD overhead; at GA=4 on 8 GPUs, its BWD phase is 19.1% slower than ZeRO-Infinity. As GA increases from 1 to 4, redundant PCIe traversals increase the BWD phase of ZeRO-Infinity by up to 16 s. *ReFlow*, in contrast, reduces this accumulation-driven increase by up to 26.6% relative to ZeRO-Infinity.

This reduction comes from CPU-Centric Gradient, which keeps gradients in BF16 across PCIe, performs FP32 promotion at the point of use inside the fused SIMD body, and retains logical gradient partitions on the CPU. Together, these changes remove GPU-side pre-transfer promotion and repeated cross-boundary retrievals, reducing the BWD phase by 7.0–55.3%. At the same time, FlashOpt and Split-Phase Schedule advance the Ready path into the backward window while deferring the persist path off the critical path. As a result, *ReFlow* reduces the exposed update tail by over 96.8%, lowers total step time by up to 74.3%, and keeps the update tail close to zero across accumulation depths.

5.4 CPU–GPU Utilization

Figure 13 shows sustained CPU–GPU co-execution rather than alternation between GPU compute and a serialized host update. GPU utilization remains high, with only brief dips during global-norm computation and step finalization, and no extended idle interval. On the CPU side, the trace reveals distinct Ready and Persist phases. The Ready path sustains approximately 35% utilization over extended intervals (e.g., $t = 2$–5 s and 7–10 s), indicating that execution is paced by gradient arrival rather than limited by CPU arithmetic throughput. The sharp drops to minimum GPU utilization near $t = 1$ s and 5 s mark synchronization points where provisional BF16 parameter generation has completed and the CPU waits for global gradient-norm validation. Once validation completes, the persist path begins immediately. The distinct CPU spikes around $t = 1$ s and 6 s correspond to FP32-state persistence draining outside the GPU-gating path before the next update. Together, these traces show that the Ready path progresses continuously with incoming gradients, while FP32-state persistence is displaced from the path that blocks the next forward pass.

5.5 CPU Memory Footprint

Figure 14 plots CPU-memory usage over time for ZeRO-Offload, ZeRO-Infinity, DOS, SuperOffload, and *ReFlow* on OPT-30B with 8 GPUs. ZeRO-Offload exhibits the largest footprint during initialization because, as a ZeRO-2-based design, it cannot exploit the initialization-time memory reduction available to ZeRO-3-based systems. After this transient phase, ZeRO-Offload and ZeRO-Infinity converge to similar steady-state memory usage. DOS remains above ZeRO-Infinity because its asynchronous execution requires additional buffering for update-path data movement. SuperOffload also exceeds ZeRO-Infinity, but for a different structural reason: its process-based asynchronous optimizer replicates state across shared-memory worker processes, causing shared-memory demand to scale with model size and exceed the 400 GB budget of our 8-GPU B200 system at 70B

Table 5. Breakdown of *ReFlow* optimizations and their impact on per-GPU throughput. AB1: FlashOpt. AB2: AB1 + CPU-Centric Gradient. AB3: AB2 + deferred FP32-state persistence. *ReFlow*: AB3 + provisional BF16 generation.

GPU	ZeRO-Infinity	AB1	AB2	AB3	*ReFlow*
1 GPU	166.41	297.59	354.01	366.61	671.29
4 GPUs	366.40	458.34	494.46	530.67	681.18
8 GPUs	420.36	504.09	531.91	574.71	673.81

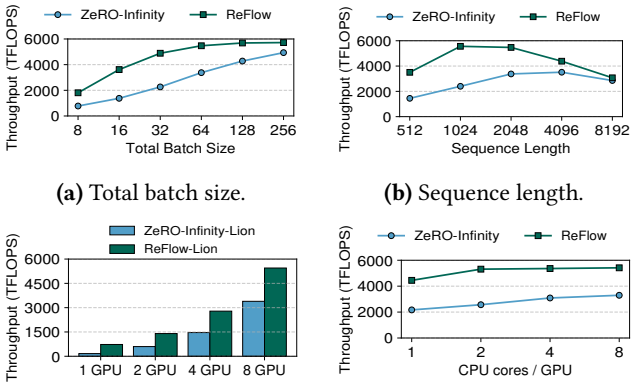
and above. *ReFlow* consistently uses the least CPU memory among the compared CPU-offloaded systems, with about 20% lower footprint than ZeRO-Infinity. This reduction comes directly from the design: CPU-Centric Gradient keeps gradients compact across the CPU–GPU boundary, no standalone FP32 gradient stage is introduced, and the deferred persist path executes in a single host address space rather than through per-worker shared-memory. As a result, *ReFlow* reduces host-memory footprint without introducing the additional memory overhead often associated with asynchronous execution. This lower footprint is consistent with the observed model-size limits. ZeRO-Offload is first constrained by initialization-time host-memory pressure, ZeRO-Infinity remains runnable through 80B, and *ReFlow* extends the limit to 95B. Together, the throughput and memory results indicate that *ReFlow* improves efficiency without increasing host-memory overhead.

5.6 Ablation Study

We use OPT-30B at GA=1 to isolate the contributions of FlashOpt, CPU-Centric Gradient, deferred FP32-state persistence, and provisional BF16 parameter generation. Table 5 shows that the gains compose across all GPU counts. AB1 improves per-GPU throughput over ZeRO-Infinity by 1.8×, 1.3×, and 1.2× on 1, 4, and 8 GPUs, respectively, establishing operator-boundary traffic in the host update path as a first-order bottleneck. AB2 delivers additional gains by compacting the gradient path, and AB3 further improves throughput by removing FP32-state persistence from the GPU-gating path. Adding provisional BF16 parameter generation on top of AB3 yields a further 1.8×, 1.3×, and 1.2×, respectively, showing that the remaining delay depends not only on how much host work remains on the critical path, but also on when that work can begin. Overall, *ReFlow* improves per-GPU throughput over ZeRO-Infinity by 4×, 1.9×, and 1.6× on 1, 4, and 8 GPUs, respectively.

5.7 Sensitivity Analysis

We evaluate the sensitivity of *ReFlow* to batch size, sequence length, optimizer, and CPU core count using OPT-30B on B200 GPUs. Unless otherwise varied, we use a sequence length of 2048 and a total batch size of 64.



(a) Total batch size. **(b)** Sequence length.

(c) Lion optimizer. **(d)** CPU cores per GPU.

Figure 15. OPT-30B throughput sensitivity to batch size, sequence length, optimizer, and CPU core count.

Batch size. *ReFlow* outperforms ZeRO-Infinity across all evaluated batch sizes (Figure 15a). At the smallest batch size, *ReFlow* achieves 2.3× higher throughput and reaches near-peak throughput at a batch size four times smaller than ZeRO-Infinity. At the largest batch size, the gap narrows to 1.2× as longer GPU computation increasingly hides ZeRO-Infinity's host-side update cost.

Sequence length. *ReFlow* outperforms ZeRO-Infinity across all evaluated sequence lengths (Figure 15b). The throughput gap narrows from 2.4× at the shortest sequence to 1.1× at the longest. Because the host optimizer cost changes little with sequence length, longer GPU execution increasingly hides the exposed host penalty. At long sequences, throughput also decreases as GPU arithmetic efficiency declines.

Optimizer. The benefits of *ReFlow* extend beyond AdamW (Figure 15c). We evaluate Lion [5], which maintains a single momentum state rather than the two moments of AdamW. *ReFlow*-Lion achieves 1.6×–4.4× higher throughput than ZeRO-Infinity-Lion across 1–8 GPUs. These gains persist despite Lion's different optimizer-state structure.

CPU cores. *ReFlow* remains robust as CPU resources are reduced (Figure 15d). Reducing the allocation from 8 to 1 core per GPU lowers *ReFlow* throughput by 18%, compared with 34% for ZeRO-Infinity. As shown in Figure 13, *ReFlow* uses roughly 30% of the available CPU capacity under the default configuration. Reducing the allocation to 2 cores per GPU therefore incurs only a 2% throughput loss. At 1 core per GPU, however, the deferred Persist pass can extend into the next iteration and delay the subsequent backward pass. ZeRO-Infinity is more sensitive because its serialized CPU optimizer remains on the critical path.

5.8 Clipping Behavior and Training Stability

We evaluate whether *ReFlow* preserves baseline training semantics under frequent clipping and quantify the cost of a clipped step. We train OPT-30B from scratch on The Pile [15]

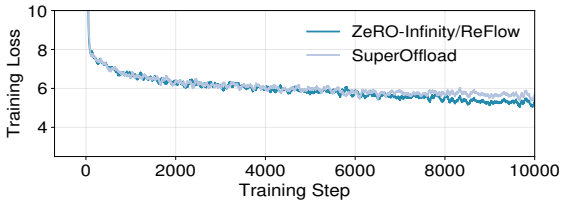


Figure 16. Training loss on OPT-30B with clipping.

Table 6. Critical-path host update time for a clipped step.

System	*ReFlow*	ZeRO-Infinity	SuperOffload
Update time (ms)	850	2440	4400

Table 7. Normalized LLC misses and DRAM traffic of *ReFlow* relative to ZeRO-Infinity CPUAdam (lower is better).

Path	Read miss (%)	Write miss (%)	DRAM (%)
Ready	31.3	0.9	31.3
Persist	43.5	1.6	50.0

using 8 GPUs with a global batch size of 32, and disable FlashAttention [7, 8] to remove variation from nondeterministic execution order. Under this setting, gradient clipping is triggered in more than 90% of training steps, allowing dense observation of clipped-step behavior. The clipped-step update time reported below is measured per clipped step. Figure 16 shows the training loss of ZeRO-Infinity, SuperOffload, and *ReFlow*. The loss curve of *ReFlow* matches that of ZeRO-Infinity throughout training, indicating that provisional BF16 parameter generation and deferred FP32-state persistence preserve the baseline update semantics even when clipping is frequent. In contrast, SuperOffload drifts from the baseline, consistent with the numerical error introduced by rollback. Table 6 reports the offloaded update time for a clipped step. Compared to ZeRO-Infinity, *ReFlow* reduces the update time by 65.2%. In contrast, SuperOffload increases the update time by 80.3%. Because clipping re-executes only the shortened Ready path, rather than the full post-validation update, its clipped-step overhead remains substantially lower than that of prior systems. ZeRO-Infinity remains slower because it retains a monolithic post-validation update path, while SuperOffload additionally pays the cost of speculative rollback.

5.9 Cache and DRAM Traffic

To isolate host-update memory behavior, we run an optimizer-only microbenchmark at 30B parameters and normalize *ReFlow* against the CPUAdam path of ZeRO-Infinity. As Table 7 shows, the Ready path reduces write-side LLC misses by 99.1% and DRAM traffic by 68.7%. In ZeRO-Infinity, gradient scaling and BF16 conversion execute as separate full-tensor operations around CPUAdam. Because each tensor at the default subgroup size exceeds LLC capacity, the intermediate values are evicted before reuse. FlashOpt instead keeps each SIMD tile register-resident through scaling, promotion, update, casting, and selective stores. This locality directly benefits the bandwidth-bound CPU optimizer (Section 3): Table 2 shows 59% higher promotion throughput after eliminating intermediate stores and 27% higher optimizer throughput with BF16 rather than pre-materialized FP32 operands. Persist retains more traffic than Ready because it commits 12 B of FP32 state per parameter, compared with the 2 B BF16 output of Ready. Deferring Persist therefore keeps the GPU-gating path on the lighter Ready phase.

6 Conclusion

We presented *ReFlow*, a CPU-offloaded LLM training framework that redesigns the host update path around its memory-traffic bottleneck. Our analysis showed that unnecessary tensor materialization, inflated gradient movement, and serialized update scheduling amplify host-memory traffic and expose the CPU update on the training critical path. *ReFlow* addresses these inefficiencies by using register-resident SIMD execution to reduce materialization, compact the gradient path, and decouple BF16 readiness from FP32-state persistence. Unlike prior approaches that mitigate host delay by shifting work back to the GPU, *ReFlow* accelerates the offloaded path itself while preserving baseline training semantics. Across B200 and A100 platforms, *ReFlow* consistently improves CPU-offloaded training throughput, with up to a 4× gain over ZeRO-Infinity on OPT-30B with a single B200 GPU, while approaching GPU-resident performance and extending the trainable model size under fixed hardware budgets. These results show that CPU offloading can provide not only additional training capacity, but also an efficient execution path for hybrid CPU–GPU training.

Acknowledgments

This work was supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grants funded by the Korean government (MSIT) (Nos. RS-2024-00396013, RS-2025-02263167, RS-2025-02263869, and RS-2025-09942968); by a National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (No. RS-2026-25490694); and by the "Advanced GPU Utilization Support Program" funded by the Government of the Republic of Korea (Ministry of Science and ICT) (No. 02-26-01-0191).

References

[1] Advanced Micro Devices. 2023. AMD EPYC 9004 Series Processors Data Sheet. Retrieved March 29, 2026 from https://www.amd.com/content/dam/amd/en/documents/products/epyc/epyc-9004-series-processors-data-sheet.pdf

[2] Arm. 2020. Armv8-A Instruction Set Architecture. Retrieved April 6, 2026 from https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Learn%20the%20Architecture/Armv8-A%20Instruction%20Set%20Architecture.pdf

[3] Jianfei Chen, Lianmin Zheng, Zhewei Yao, Dequan Wang, Ion Stoica, Michael W. Mahoney, and Joseph E. Gonzalez. 2021. ActNN: Reducing Training Memory Footprint via 2-Bit Activation Compressed Training. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*. PMLR, 1803–1813. https://proceedings.mlr.press/v139/chen21z.html

[4] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. arXiv:1604.06174

[5] Xiangning Chen, Chen Liang, Da Huang, Esteban Real, Kaiyuan Wang, Hieu Pham, Xuanyi Dong, Thang Luong, Cho-Jui Hsieh, Yifeng Lu, and Quoc V. Le. 2023. Symbolic Discovery of Optimization Algorithms. In *Advances in Neural Information Processing Systems*, Vol. 36. 49205–49233. doi:10.52202/075280-2140

[6] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2023. PaLM: Scaling Language Modeling with Pathways. *Journal of Machine Learning Research* 24, 240 (2023), 1–113. https://jmlr.org/papers/v24/22-1144.html

[7] Tri Dao. 2024. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. In *International Conference on Learning Representations (ICLR)*. 26738–26751.

[8] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *Advances in Neural Information Processing Systems*, Vol. 35. 16344–16359. doi:10.52202/068431-1189

[9] DeepSpeed Team. 2025. DeepSpeed v0.18.1. Retrieved April 13, 2026 from https://github.com/deepspeedai/DeepSpeed/releases/tag/v0.18.1 Release v0.18.1, commit 3631712.

[10] DeepSpeed Team. 2026. DeepSpeed v0.19.3. Retrieved October 10, 2026 from https://github.com/deepspeedai/DeepSpeed/releases/tag/v0.19.3 Release v0.19.3, commit 0c36f6d.

[11] Tim Dettmers, Mike Lewis, Sam Shleifer, and Luke Zettlemoyer. 2022. 8-bit Optimizers via Block-wise Quantization. In *International Conference on Learning Representations (ICLR)*. 7891–7909.

[12] Nan Du, Yanping Huang, Andrew M. Dai, Simon Tong, Dmitry Lepikhin, Yuanzhong Xu, Maxim Krikun, Yanqi Zhou, Adams Wei Yu, Orhan Firat, Barret Zoph, Liam Fedus, Maarten P. Bosma, Zongwei Zhou, Tao Wang, Emma Wang, Kellie Webster, Marie Pellat, Kevin Robinson, Kathleen Meier-Hellstern, Toju Duke, Lucas Dixon, Kun Zhang, Quoc Le, Yonghui Wu, Zhifeng Chen, and Claire Cui. 2022. GLaM: Efficient Scaling of Language Models with Mixture-of-Experts. In *Proceedings of the 39th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 162)*. PMLR, 5547–5569. https://proceedings.mlr.press/v162/du22c.html

[13] William Fedus, Barret Zoph, and Noam Shazeer. 2022. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. *Journal of Machine Learning Research* 23, 120 (2022), 1–39.

[14] Trevor Gale, Deepak Narayanan, Cliff Young, and Matei Zaharia. 2023. MegaBlocks: Efficient Sparse Training with Mixture-of-Experts. In *Proceedings of Machine Learning and Systems*, Vol. 5. 288–304. https://proceedings.mlsys.org/paper_files/paper/2023/hash/5a54f79333768effe7e8927bcccffe40-Abstract-mlsys2023.html

[15] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. 2020. The Pile: An 800GB Dataset of Diverse Text for Language Modeling. arXiv:2101.00027 doi:10.48550/arXiv.2101.00027

[16] Chien-Chin Huang, Gu Jin, and Jinyang Li. 2020. SwapAdvisor: Pushing Deep Learning Beyond the GPU Memory Limit via Smart Swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 1341–1355. doi:10.1145/3373376.3378530

[17] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *Advances in Neural Information Processing Systems*, Vol. 32. 103–112.

[18] Changho Hwang, Wei Cui, Yifan Xiong, Ziyue Yang, Ze Liu, Han Hu, Zilong Wang, Rafael Salas, Jithin Jose, Prabhat Ram, HoYuen Chau, Peng Cheng, Fan Yang, Mao Yang, and Yongqiang Xiong. 2023. Tutel: Adaptive Mixture-of-Experts at Scale. In *Proceedings of Machine Learning and Systems*, Vol. 5. 269–287. https://proceedings.mlsys.org/paper_files/paper/2023/hash/5616d34cf8ff73942cfd5aa922842556-Abstract-mlsys2023.html

[19] Intel. 2023. Intel Xeon Platinum 8490H Processor. Retrieved March 29, 2026 from https://www.intel.com/content/www/us/en/products/sku/231747/intel-xeon-platinum-8490h-processor-112-5m-cache-1-90-ghz/specifications.html

[20] Intel. 2026. Intel 64 and IA-32 Architectures Software Developer's Manual. Retrieved October 9, 2026 from https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[21] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph E. Gonzalez, Kurt Keutzer, and Ion Stoica. 2020. Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization. In *Proceedings of Machine Learning and Systems*, Vol. 2. 497–511. https://proceedings.mlsys.org/paper_files/paper/2020/hash/0b816ae8f06f8dd3543dc3d9ef196cab-Abstract.html

[22] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *International Conference on Learning Representations (ICLR)*.

[23] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. 2023. Reducing Activation Recomputation in Large Transformer Models. In *Proceedings of Machine Learning and Systems*, Vol. 5. 341–353. https://proceedings.mlsys.org/paper_files/paper/2023/hash/80083951326cf5b35e5100260d64ed81-Abstract-mlsys2023.html

[24] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2021. GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding. In *International Conference on Learning Representations (ICLR)*. 14225–14247.

[25] Xinyu Lian, Masahiro Tanaka, Olatunji Ruwase, and Minjia Zhang. 2026. SuperOffload: Unleashing the Power of Large-Scale LLM Training on Superchips. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 249–264. doi:10.1145/3760250.3762217

[26] Llama Team. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 doi:10.48550/arXiv.2407.21783

[27] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. In *International Conference on Learning Representations*

(ICLR).

[28] Avinash Maurya, Jie Ye, M. Mustafa Rafique, Franck Cappello, and Bogdan Nicolae. 2024. Deep Optimizer States: Towards Scalable Training of Transformer Models Using Interleaved Offloading. In *Proceedings of the 25th International Middleware Conference*. 404–416. doi:10.1145/3652892.3700781

[29] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory F. Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. 2018. Mixed Precision Training. In *International Conference on Learning Representations (ICLR)*.

[30] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*. 1–15. doi:10.1145/3341301.3359646

[31] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. 2021. Memory-Efficient Pipeline-Parallel DNN Training. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*. PMLR, 7937–7947. https://proceedings.mlr.press/v139/narayanan21a.html

[32] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Anand Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 2021. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Article 58, 15 pages. doi:10.1145/3458817.3476209

[33] NVIDIA. 2020. NVIDIA A100 Tensor Core GPU Architecture. Retrieved April 16, 2026 from https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

[34] NVIDIA. 2023. Train With Mixed Precision. NVIDIA Documentation. Retrieved October 9, 2026 from https://docs.nvidia.com/deeplearning/performance/mixed-precision-training/index.html

[35] NVIDIA. 2025. Product Carbon Footprint (PCF) Summary for NVIDIA HGX B200. Retrieved April 16, 2026 from https://images.nvidia.com/aem-dam/Solutions/documents/HGX-B200-PCF-Summary.pdf

[36] NVIDIA. [n.d.]. NVIDIA HGX AI Factory (HGX H100, H200, B200): Components. NVIDIA Reference Architecture Documentation. Retrieved October 10, 2026 from https://docs.nvidia.com/enterprise-reference-architectures/hgx-ai-factory-h100-h200-b200/latest/components.html

[37] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. 2013. On the Difficulty of Training Recurrent Neural Networks. In *Proceedings of the 30th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 28)*. PMLR, 1310–1318. https://proceedings.mlr.press/v28/pascanu13.html

[38] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. 2020. Capuchin: Tensor-based GPU Memory Management for Deep Learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 891–905. doi:10.1145/3373376.3378505

[39] PyTorch Contributors. 2026. Automatic Mixed Precision Examples. PyTorch Documentation. Retrieved April 13, 2026 from https://docs.pytorch.org/docs/stable/notes/amp_examples.html

[40] PyTorch Contributors. 2026. FullyShardedDataParallel. PyTorch Documentation. Retrieved October 9, 2026 from https://docs.pytorch.org/docs/2.14/fsdp.html

[41] PyTorch Contributors. [n.d.]. torch.nn.utils.clip_grad_norm_. PyTorch Documentation. Retrieved October 9, 2026 from https://docs.pytorch.org/docs/2.14/generated/torch.nn.utils.clip_grad_norm_.html

[42] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. OpenAI Technical Report. Retrieved October 9, 2026 from https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf

[43] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. In *SC '20: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–16. doi:10.1109/SC41405.2020.00024

[44] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Article 59, 14 pages. doi:10.1145/3458817.3476205

[45] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. DeepSpeed: System Optimizations Enable Training Deep Learning Models with over 100 Billion Parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 3505–3506. doi:10.1145/3394486.3406703

[46] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. ZeRO-Offload: Democratizing Billion-Scale Model Training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 551–564. https://www.usenix.org/conference/atc21/presentation/ren-jie

[47] Noam Shazeer and Mitchell Stern. 2018. Adafactor: Adaptive Learning Rates with Sublinear Memory Cost. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 80)*. PMLR, 4596–4604. https://proceedings.mlr.press/v80/shazeer18a.html

[48] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford Alpaca: An Instruction-Following LLaMA Model. Retrieved April 16, 2026 from https://github.com/tatsu-lab/stanford_alpaca

[49] 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 doi:10.48550/arXiv.2302.13971

[50] 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 doi:10.48550/arXiv.2307.09288

[51] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. In *Advances in Neural Information Processing Systems*, Vol. 30. 5998–6008.

[52] Pedram Zamirai, Jian Zhang, Christopher R. Aberger, and Christopher De Sa. 2020. Revisiting BFloat16 Training. arXiv:2010.06192 doi:10.48550/arXiv.2010.06192

[53] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona T. 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 doi:10.48550/arXiv.2205.01068

[54] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. 2022. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, 559–578. https://www.usenix.org/conference/osdi22/presentation/zheng-lianmin

A Model Configurations

Table 8 summarizes the model configurations used in the paper evaluation.

Table 8. Model configurations used in the paper evaluation. Shape denotes layers / hidden size / FFN size / attention heads.

Size	Base	Type	Shape (L / H / FFN / heads)
3B	LLaMA-2-13B	Scaled	60 / 2048 / 5120 / 16
5B	LLaMA-2-13B	Scaled	44 / 3072 / 7936 / 24
10B	LLaMA-2-13B	Scaled	50 / 4096 / 10496 / 32
20B	LLaMA-2-13B	Scaled	25 / 8192 / 20992 / 64
25B	LLaMA-2-13B	Scaled	30 / 8192 / 22528 / 64
30B	OPT-30B	Native	48 / 7168 / 28672 / 56
50B	OPT-30B	Scaled	80 / 7168 / 29184 / 56
70B	LLaMA-3-70B	Native	80 / 8192 / 28672 / 64
80B	LLaMA-3-70B	Scaled	100 / 8192 / 25600 / 64
90B	LLaMA-3-70B	Scaled	113 / 8192 / 25600 / 64
95B	LLaMA-3-70B	Scaled	119 / 8192 / 25856 / 64

B Artifact Appendix

B.1 Abstract

ReFlow is an asynchronous CPU-offload optimizer for DeepSpeed [45] ZeRO Stage 3 [43]. The baseline is ZeRO-Infinity [44] (optimizer offload + Stage 3), which runs the CPU optimizer step serially after the backward pass. *ReFlow* instead overlaps that step with the backward pass and offloads gradients in half precision. The artifact demonstrates three properties. First, the backward and optimizer-step timers both shrink, so step time improves and TFLOPS increase. Second, host RAM and GPU memory usage are both reduced. Third, per-step losses remain byte-identical to the baseline.

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

- **Algorithm:** asynchronous CPU-offload optimizer overlapping the CPU optimizer step with the backward pass
- **Program:** `finetune_zero3.py`, Hugging Face Transformers on a DeepSpeed [10] v0.19.3 fork carrying the *ReFlow* runtime
- **Model:** OPT-350M/30B [53], Llama-7B [49], bundled in the image
- **Data set:** `tatsu-lab/alpaca`, bundled in the image
- **Run-time environment:** Linux x86_64 with the Docker image `aeasplos/reflow:AE1.1`. The host needs CUDA, an NVIDIA driver, Docker, and the NVIDIA Container Toolkit
- **Hardware:** one CUDA-enabled NVIDIA GPU and an x86_64 CPU with AVX, AVX-512 recommended
- **Metrics:** peak GPU and host memory, per-step time, TFLOPS, DeepSpeed's backward and optimizer-step timers, bit-exact hex-float loss
- **Output:** console tables printed by the two scripts
- **Experiments:** two scripts, `run_compare.sh` and `check_bitexact.sh`. Magnitudes vary with the host, but the direction of each metric does not.
- **How much disk space required (approximately)?:** over 100 GB, for the Docker image
- **How much time is needed to prepare workflow (approximately)?:** <1 hour
- **How much time is needed to complete experiments (approximately)?:** ~1 hour
- **Publicly available?:** yes
- **Code licenses (if publicly available)?:** Apache-2.0
- **Archived (provide DOI)?:** 10.5281/zenodo.23210298

B.3 Description

B.3.1 How to access.

- Docker image, everything preinstalled: https://hub.docker.com/r/aeasplos/reflow/tags
- Runtime, the DeepSpeed fork: https://github.com/aeasplos/DeepSpeed_reflow
- Experiments, the fine-tuning scripts: https://github.com/aeasplos/DeepSpeedExample_reflow

Detailed instructions, the full set of scripts, and customization options are documented in the README at https://github.com/aeasplos/DeepSpeedExample_reflow/tree/main/training/DeepSpeed-Reflow

The evaluation below is confined to a single GPU, which keeps it reproducible on the widest range of machines.

B.3.2 Hardware dependencies. One CUDA-enabled NVIDIA GPU is required. The binding constraint is the ZeRO-Infinity [44] baseline, the heavier of the two modes. On our machine its peaks were 88 GB of GPU memory and 530 GB of host RAM for OPT-30B, and 31 GB and 130 GB for Llama-7B. These are the figures `run_compare.sh` checks free memory against before starting a configuration. We used an NVIDIA RTX PRO 6000 Blackwell (96 GB), which covers both. The CPU must be x86_64 with AVX, and AVX-512 is recommended.

B.3.3 Software dependencies. The Docker image carries the *ReFlow* DeepSpeed fork and the Python stack. The host must provide Linux x86_64 with CUDA and an NVIDIA driver installed, plus Docker and the NVIDIA Container Toolkit. We used CUDA 12.8.

B.3.4 Data sets. `tatsu-lab/alpaca` [48], bundled in the Docker image.

B.3.5 Models. OPT-350M/30B [53], and Llama-7B [49], all bundled in the image, so the experiments below need no download. Any other model is fetched from the Hugging Face Hub on first use, and the gated Llama checkpoints then require a Hugging Face login with the licenses accepted.

B.4 Installation

```
docker pull aeasplos/reflow:AE1.1
docker run --gpus all -it aeasplos/reflow:AE1.1
cd /ReFlow/DeepSpeedExample_reflow/training/DeepSpeed-Reflow
```

The `docker` commands may need `sudo`, depending on how Docker is set up on the host. A from-source install is documented in the README.

Basic test. One command, a few minutes on any CUDA GPU:

```
bash finetune_opt-350m-1gpu.sh reflow
```

It should print the selected CPU kernel path, then one line per step with a decreasing loss.

B.5 Experiment workflow

Two commands, both run from the experiment directory inside the container. The two scripts are provided in the Docker image rather than in the repository, as they exist for this evaluation.

```
bash run_compare.sh
bash check_bitexact.sh 1   # arg: number of GPUs
```

`run_compare.sh` runs both modes on Llama-7B and OPT-30B and reports, in one table, the quantities behind three figures of the paper: the *ReFlow* versus ZeRO-Infinity throughput comparison of Fig. 9–11, the reduction in backward and optimizer time at GA1 shown in Fig. 12, and the reduction in memory usage shown in Fig. 14. `check_bitexact.sh` takes the number of GPUs as its argument and runs the same OPT-350M job in each mode under full determinism, diffing the per-step hex-float losses to show that they match at the bit level, as in Fig. 16.

B.6 Evaluation and expected results

Expected output of `run_compare.sh`, measured on one NVIDIA RTX PRO 6000 Blackwell (96 GB):

```
# finetune_opt-30b_1gpu
metric                     Reflow  ZeRO-Inf.
peak GPU memory (MiB)       87850      87980
peak host RAM (MiB)        469866     537467
step time (ms)              17426      47210
TFLOPS                      236.1       87.1
bwd_microstep (ms)        12842.8    16509.8
step_microstep (ms)         267.6    26548.9

  SPEEDUP: 2.71x
```

OPT-30B needs far more memory than Llama-7B, so it is the configuration most likely not to fit. `run_compare.sh` checks free host and GPU memory before starting each configuration and skips one that does not fit, reporting what was missing. In that case only the Llama-7B result is produced, which behaves the same way (2.74×). The optimizer-step timer collapses and the backward timer falls, so step time drops and TFLOPS rise. Peak host RAM and peak GPU memory both drop as well. Magnitudes depend on the host's GPU and CPU and will differ from the above, while the direction of each metric is expected to be the same.

`check_bitexact.sh` then prints `BIT-IDENTICAL`.

B.7 Experiment customization

The fine-tuning scripts in the repository cover further models, multi-GPU configurations, and Lion in place of Adam. See the README for the full set of scripts and options.

B.8 Notes

The evaluation above uses a single GPU for accessibility. The paper reports the multi-GPU results, and the README documents how to run those configurations.