SysX: System for Training under Memory Constraints

Junyeol Ryu*
Seoul National University

Yujin Jeong*
Seoul National University

Daeyoung Park
Seoul National University

Jinpyo Kim
Seoul National University

Heehoon Kim
Seoul National University

Jaejin Lee
Seoul National University

Abstract

Training large language models (LLMs) with limited computing resources is challenging because of their immense memory space requirements. In this paper, we specifically focus on the scenarios where we have insufficient aggregate GPU memory to store all model states but explore pipeline parallelism and offloading across all system resources to train the model. In this context, SysX presents a hybrid GPU and CPU pipelining mechanism that consists of two pipelines: a GPU pipeline to reduce the bubbles in conventional pipeline parallelism and a GPU-CPU pipeline to alleviate data transfer overhead and CPU bottlenecks in offloading data and computing. We evaluate SysX for training LLMs of various sizes with diverse configurations in practice. The result indicates that SysX outperforms the state-of-the-art by 1.26×. We plan to make SysX publicly available to broaden the accessibility of large-scale model training.

1 Introduction

Large language models (LLMs) [3, 15, 40, 46, 56] have scaled dramatically to trillion parameters and are very successful for various downstream tasks. However, such an overwhelming number of parameters requires large memory space during training. State-of-the-art models, such as LLaMA [27, 48, 49] and OPT [54], require a memory footprint on a terabyte-scale. They are typically trained with a supercomputer-scale cluster in a data center [12, 41, 47].

The cost and resources for training an LLM are highly challenging for many academic institutions and startups because they typically rely on small GPU clusters or small-scale cloud services. For example, a 0.1 trillion parameter model requires 1.83 terabytes to store its states during training [42], which far exceeds the aggregate GPU memory of a small GPU cluster with a few nodes. Thus, developing a technique that efficiently trains large models with limited resources can significantly broaden the accessibility of LLMs.

*Equal contribution

A practical approach to mitigating the memory requirement is scale-out techniques, such as model parallelism [12, 41, 47], to distribute model training across multiple GPUs. Among others, pipeline parallelism [7, 12] partitions the model into different stages and assigns the stages to GPUs. A mini-batch is divided into smaller micro-batches and executed across the pipeline stages. It requires only peer-to-peer (P2P) communication to transfer activations between GPUs, thereby minimizing communication overhead. However, it also introduces inefficiencies due to GPU idle times, referred to as pipeline bubbles. It may lead to significant system under-utilization and necessitate sophisticated pipeline scheduling to reduce them [20, 23, 30–32].

Another widely-used approach to alleviating the memory requirement is offloading [1, 10, 11, 14, 19, 22, 34, 44, 45]. The memory capacity is extended to non-GPU memory (e.g., the CPU main memory) to allow larger model training. Only the minimum amount of data required for the current operation is fetched and placed in the GPU memory (e.g., layer parameters are fetched on demand just before the computation for the layer). After performing the operation, the fetched data are freed, and newly generated data by the operation are offloaded to the non-GPU memory (e.g., gradients of the layer are stored in the CPU memory after the layer's backward pass). Recent approaches even offload some computational tasks (e.g., optimization steps) to the CPU to further exploit heterogeneous resources [8, 26, 44]. However, these approaches introduce data transfer overhead between the GPU and CPU. In addition, the low computational capacity of the CPU may become a performance bottleneck.

To this end, this paper proposes SysX, a hybrid GPU-CPU pipeline for training LLMs under memory pressure. We specifically focus on the scenarios where we have insufficient aggregate GPU memory to store all model states but explore the use of model parallelism and offloading together across all system resources to train the model *by any means*. In this context, SysX offers an efficient solution through two pipelines: *GPU pipeline* and *GPU-CPU pipeline*. The GPU pipeline reduces the bubbles introduced in conventional pipeline parallelism.

The GPU-CPU pipeline hides the data transfer overhead between the CPU and GPUs and alleviates the performance bottleneck caused by the slower CPU when offloading data and computing.

SysX's GPU pipeline presents a *decoupled pass assignment*, which assigns the forward and backward passes of the same stage to different GPUs for better pipeline scheduling. Such mechanism is facilitated by storing the model parameters on the CPU's shared memory (shmem) and exploiting activation recomputation [5, 13, 17, 18]. Moreover, SysX introduces fine-grained stage partitioning to further eliminate the bubbles due to the gap in execution time between the forward and backward passes and optimizes the communication schedule for activation checkpoints to hide the additional communication overhead between the GPUs.

SysX's GPU-CPU pipeline presents a *asynchronous CPU optimizer*, which executes the optimization steps in parallel with the GPU pipeline, thereby overlapping and hiding the CPU optimizer overhead. This mechanism is enabled by bypassing optimizer synchronization [9, 38, 39] and shifting the numerical validation as a post-step process while guaranteeing correctness through a roll-back mechanism.

The major contributions of SysX are summarized as follows:

- We propose SysX, a hybrid GPU-CPU pipelining mechanism that efficiently leverages offloading and achieves high utilization of both GPUs and the CPU when training LLMs under memory pressure.
- We compare SysX against the state-of-the-art LLM pipelining mechanisms with offloading, Mobius [8] and Megatron [32], on an eight node cluster with 32 GPUs by training LLaMA-2 models [49] with various model sizes and batch sizes. SysX outperforms these methods 1.26× and 1.51× on average, respectively.
- We will make SysX publicly available after publication to foster research and expand the accessibility of LLMs.

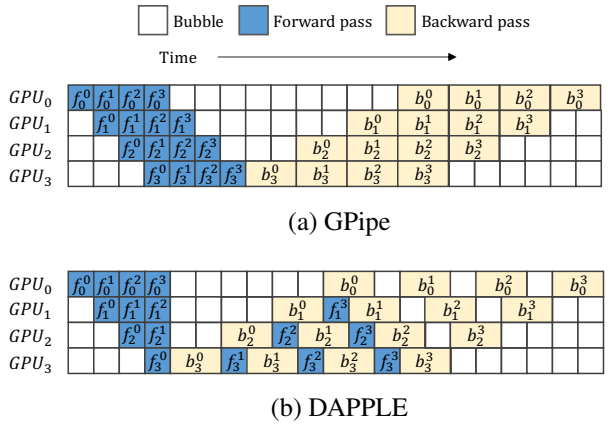
2 Background and Related Work

This section introduces pipeline parallelism and its techniques to train language models under GPU memory pressure.

2.1 Pipeline Parallelism

Pipeline parallelism is a type of model parallelism [12, 41, 47] that trains large models on multiple GPUs. It partitions a model into sequential groups of layers called *stages* and assigns the stages to GPUs. It divides a mini-batch into smaller micro-batches and executes them in a pipelined manner across these stages.

Suppose a model is partitioned into N_s stages, and a mini-batch is divided into N_m micro-batches. We denote the ith



(a) GPipe

(b) DAPPLE

Figure 1: Different pipeline schedules with four GPUs. GPU_i denotes the ith GPU. f_i^j and b_i^j denote the ith stage's forward/backward pass on the jth micro-batch, respectively. Note that a backward pass has twice the workload than a forward pass.

$(0 \le i < N_s)$ stage as S_i and jth $(0 \le j < N_m)$ micro-batch as m_j. We denote the forward and backward passes of S_i for m_j as f_i^j and b_i^j, respectively. For convenience, we denote the set of forward passes f_i^j for all m_j as f_i. Similarly, b_i denotes the set of backward passes b_i^j for all m_j.

Figure 1(a) illustrates an AFAB (all forward, all backward) schedule of GPipe [12] with $N_g = 4$ GPUs, $N_s = 4$ stages, and $N_m = 4$ micro-batches. Thus, it has f_i^j and b_i^j for $i \in \{0,1,2,3\}$ and $j \in \{0,1,2,3\}$. GPipe first pipelines the forward passes of all micro-batches, followed by the backward passes of all micro-batches.

A pipeline schedule typically introduces idle times on GPUs. We call them to as *bubbles*. Suppose the forward and backward passes take T_f and T_b time, respectively. Note that there are two kinds of bubbles in GPipe. The forward pass causes one (the time taken by the bubble is T_f), and the backward pass causes the other (the time taken by the bubble is T_b). Then, the time consumed by bubbles, T_{bbs}, in GPipe's pipeline on a GPU is formulated as follows:

$$T_{\text{bbs}} = T_f \cdot (N_g - 1) + T_b \cdot (N_g - 1). \tag{1}$$

We define *bubble ratio* as the idle time consumed by bubbles divided by the overall execution time T_{overall} of the pipeline on a GPU in Figure 1(a):

$$T_{\text{overall}} = T_f \cdot (N_m + N_g - 1) + T_b \cdot (N_m + N_g - 1). \tag{2}$$

Thus, GPipe has a bubble ratio:

$$\frac{T_{\text{bbs}}}{T_{\text{overall}}} = \frac{T_f \cdot (N_g - 1) + T_b \cdot (N_g - 1)}{T_f \cdot (N_m + N_g - 1) + T_b \cdot (N_m + N_g - 1)}. \tag{3}$$

DAPPLE [7] in Figure 1(b) presents a 1F1B (one forward, one backward) schedule, where each GPU alternates between forward and backward passes of different micro-batches. It

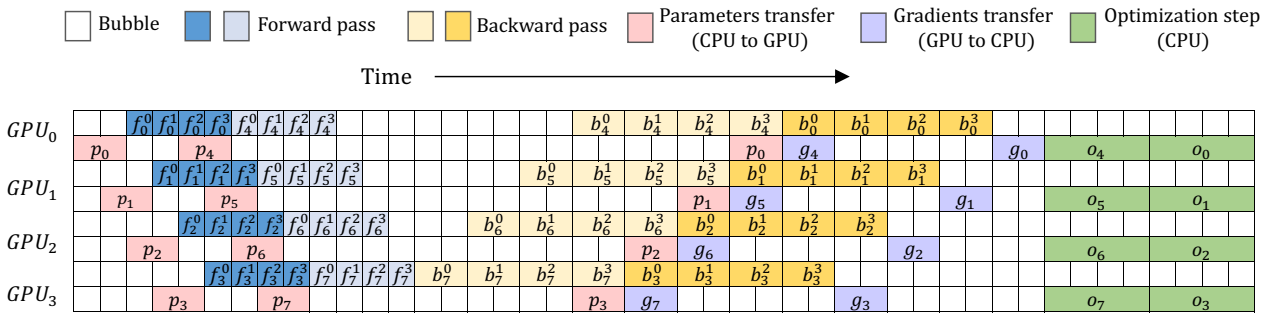


Figure 2: Mobius pipeline schedule with four GPUs. GPU_i denotes the ith GPU. f_i^j and b_i^j denote the ith stage's forward/backward pass on the jth micro-batch. Note that a backward pass has twice the workload than a forward pass. Two colors distinguish each forward/backward pass to indicate that it belongs to a different stage assigned to the same GPU. For example, GPU_0's forward passes on stage S_0 and S_4 are colored blue and sky blue. p_i and g_i denote CPU to GPU parameters transfer and GPU to CPU gradients transfer of stage S_i, respectively. o_i denotes the optimization step of stage S_i executed on the CPU.

reduces the peak activation memory usage of a stage from $\leq N_m \cdot M_a$ to $N_g \cdot M_a$ by releasing activations as early as possible, where M_a is the memory required for all activations of a single micro-batch for a stage. The reduced activation memory usage enabled by a different pipeline schedule allows a larger N_m with the same GPU memory space. However, DAPPLE has an identical bubble ratio to GPipe.

A high bubble ratio reduces pipeline utilization. To reduce the bubble ratio, one can simply increase the number of micro-batches N_m, thereby increasing T_{overall}. However, achieving a sufficiently large N_m (e.g., $N_m \geq 4N_g$ as suggested in [12]) often introduces inefficiencies due to two key factors. One is that models typically have a practical upper limit on the mini-batch size, beyond which convergence is negatively affected [2, 6, 51–53]. The other is that increasing N_m reduces the micro-batch size for a given mini-batch size, compromising GPU computational efficiency [20].

Coupled pass assignment. On the other hand, reducing bubbles to decrease T_{bbs} is challenging. In both GPipe and DAPPLE, the same GPU is responsible for both the forward and backward passes f_i^j and b_i^j of the same stage S_i for micro-batch m_j. This creates bubbles at the beginning of the backward pass, as the backward pass proceeds in the reverse order of stages of the forward pass. We define the assignment of the same GPU to both f_i^j and b_i^j as *coupled pass assignment*. Almost all existing pipelines adhere to the coupled pass assignment for three key reasons. First, both f_i^j and b_i^j use S_i's parameters Ψ_i. Second, b_i^j reuses the activations a_i^j generated by f_i^j. Finally and more critically, Ψ_i and a_i^j are stored in the GPU memory.

This paper reexamines the coupled pass assignment in offloading scenarios when pipeline parallelism uses non-GPU memory to alleviate GPU memory pressure. It focuses on opportunities to reduce bubbles, improving the bubble ratio to increase performance.

2.2 Pipeline Parallelism with Offloading

As model sizes continue to grow, the increased memory space requirement results in GPU memory pressure. As a remedy, pipeline parallelism can leverage memory-efficient techniques, such as offloading and activation recomputation.

Offloading [1, 10, 11, 14, 19, 22, 34, 44, 45] is a technique to use non-GPU memory (e.g., the CPU main memory) to store model states (e.g., parameters, gradients, and optimizer states) and residual states (e.g., activations) during training.

Activation recomputation [5, 13, 17, 18] reduces activation memory usage by recomputing the activations in the backward pass instead of storing them in the forward pass and keeping them until the backward pass. Only a subset of activations, or *checkpoint*, is stored in the forward pass and used to recompute all activations before calculating the gradients in the backward pass. Large models such as Turing-NLG 17.2B and GPT-3 175B were trained using activation recomputation [42].

Mobius [8] is a state-of-the-art pipelining mechanism that uses offloading. It introduces an *interleaved* AFAB schedule, in which the stages of GPipe are further subdivided into smaller stages to reduce the memory space requirements of each stage. Along with assigning multiple stages per GPU, all data are stored in the CPU memory, with only the minimum amount of data required for the current stage fetched and placed in the GPU memory. Their key idea for minimizing the overhead of accessing non-GPU memory is to prefetch the data required for the next stage in an overlapped manner with the computation of the current stage. In addition, it exploits activation recomputation when training large models to reduce the data transfer overhead.

Consider Figure 2, which illustrates the pipeline schedule of Mobius with $N_g = 4$ GPUs, $N_s = 8$ stages, and $N_m = 4$ micro-batches. Thus, it has f_i^j and b_i^j for $i \in \{0, \cdots, 7\}$ and $j \in \{0, \cdots, 3\}$. Stage $S_{0,4}$, $S_{1,5}$, $S_{2,6}$, and $S_{3,7}$ are mapped to GPU_0, GPU_1, GPU_2, and GPU_3, respectively. Mobius first

pipelines the forward passes of all micro-batches (f_{0-3}) for the first stage in each GPU (S_{0-3}), followed by that (f_{4-7}) for the second stage in each GPU (S_{4-7}). Then, it pipelines the backward passes of all micro-batches (b_{4-7}) for the second stage in each GPU (S_{4-7}), followed by that (b_{0-3}) for the first stage in each GPU (S_{0-3}).

Mobius stores all stages in the CPU memory. Hence, it transfers a copy of stage's parameters from the CPU memory to GPU memory before executing it, and frees this copy after finishing the stages' execution on all micro-batches. We denote the CPU to GPU transfer of stage S_i's parameters copy as p_i. Similarly, it transfers a stage's gradients, accumulated across all micro-batches, from the GPU memory to CPU memory after finishing the stage's backward passes on all micro-batches. We denote the GPU to CPU transfer of S_i's gradients as g_i. We assume training with activation recomputation, so activations are not transferred between GPU and CPU in Figure 2. Optimization steps, with o_i denoting the CPU's optimization step of S_i, are processed by the CPU after the pipeline flush, as explained in detail in Section 2.3.

Decoupled pass assignment. Mobius also adheres to the coupled pass assignment (Section 2.1). In Mobius, both f_i^j and b_i^j use S_i's parameters Ψ_i. However, with activation recomputation, b_i^j does not reuse the activations a_i^j generated by f_i^j but instead recomputes them during the backward pass. More critically, Ψ_i and a_i^j (if it exists) are not stored in the GPU memory but in the CPU memory. In a nutshell, pipeline parallelism with offloading indicates that the forward and backward passes for the same stage no longer need to be assigned to the same GPU but can be *decoupled*. Based on this observation, we investigate a new mechanism to improve the bubble ratio for pipeline parallelism with offloading.

2.3 Hybrid GPU-CPU Training

Pipeline parallelism with offloading stores optimizer states in non-GPU memory, along with parameters and gradients. Mobius leverages a CPU-based optimizer, similar to DeepSpeed CPU Adam [44], to update parameters directly on the CPU. Such a mechanism to exploit both GPUs and CPUs is called hybrid GPU-CPU training [25, 26]. Executing optimization steps on the CPU is crucial when training under GPU memory pressure, as optimizer states are often significantly larger than other model states [21, 26, 42]. For instance, in mixed-precision training with Adam, the memory space required for optimizer states is $\times 8$ that of the parameters.

Consider the green squares of Figure 2, which illustrates Mobius's optimization steps on the CPU. It introduces inefficiencies because they begin synchronously across all GPUs and do not overlap with the forward and backward pass execution on the GPU. Such inefficiencies arise because conventional mixed-precision training requires synchronization of overflow in the gradients before the optimization step. Figure 3 describes its detailed mechanism. The FP16 gradients transferred from the GPU to the CPU are first converted into FP32 (Line 1), unscaled, and checked for overflow (Line 2). The results of each stage's gradients are synchronized across all stages (Line 3). If overflow is detected at any stage, an invalid loss scale was used during that training iteration. Thus, the optimizer skips the parameter update. Otherwise, all gradients are used to update the parameters, ensuring numerical stability (Lines 4-7). Thus, the explicit synchronization of overflow prevents Mobius from processing the optimization steps of different stages asynchronously.

```
1 g^32 ← copy_fp16_grad_to_fp32(g^16)
2 overflow ← unscale_and_check_overflow(g^32)
3 all_reduce(overflow)
4 if overflow = False then
5     p^32 ← param_update(p^32, g^32)
6     p^16 ← copy_fp32_param_to_fp16(p^32)
7 end
```

Figure 3: Optimization step of mixed-precision training.

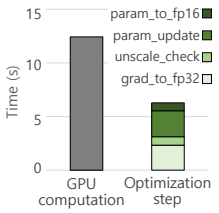


Figure 4: Optimization step breakdown.

Unfortunately, the optimization steps can consume a substantial amount of time on the CPU. Figure 4 compares the time taken by the GPU's computation and the CPU's optimization steps during Mobius's training, with a breakdown into functions in Figure 3. Although these functions primarily rely on element-wise operations with low computational intensity, they can add significant idle times to GPUs when not overlapped with the GPU's computation. As a result, GPUs and CPUs cannot achieve high utilization simultaneously, limiting the benefits of hybrid GPU-CPU training.

Building on this insight, we further explore pipelining the optimization steps on the CPU with the execution on the GPU, thereby improving both their utilization and mitigating the overhead of the CPU optimization steps.

2.4 Related Work

Different pipeline schedules. A key objective of pipeline schedules is to reduce the bubble ratio. PipeDream [30] skips periodic pipeline flushes and injects more micro-batches into the pipeline to achieve an almost zero bubble ratio. However, this requires updating the parameters after each micro-batch's backward pass and storing additional versions to ensure parameter consistency between the forward and backward passes of the same micro-batch. Bidirectional pipelines, such as Chimera [20] and MixPipe [55], operate two pipelines in opposite directions to reduce bubbles, but this requires duplicating parameters across each two GPUs. Interleaved-stage approaches, like Megatron [32] and Hanayo [23], partition a model to assign multiple stages per GPU, reducing the bubble

time while increasing the amount of communication.

Leveraging heterogeneous devices. Existing proposals support offloading model states [37, 44] and residual states [1, 10, 19, 22, 43, 45] to non-GPU memory only for training with a single GPU. Among these, ZeRO-Offload [44] offloads optimizer states to the CPU memory and executes optimization steps on the CPU. ZeRO-Infinity [42] and Mobius [8] extend this to support fully sharded data parallelism [41] and pipeline parallelism [12], respectively. ZeRO-Offload++ [50] and Deep Optimizer States [26] perform optimization steps on both the GPU and CPU.

3 The Design of SysX

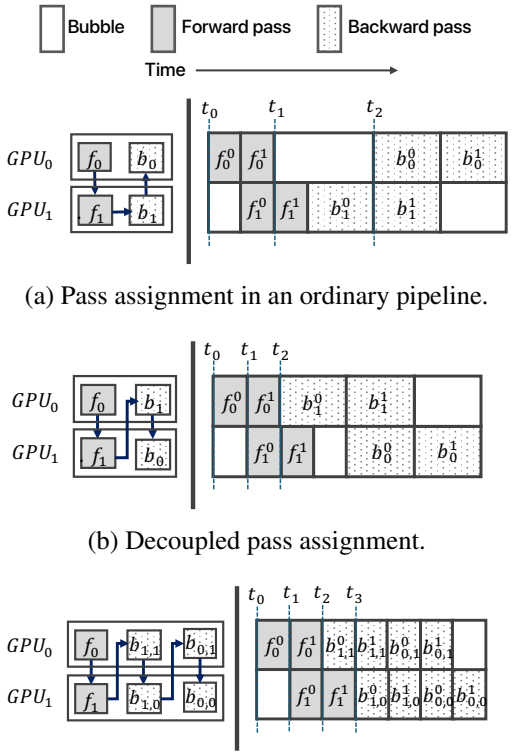
This section describes the pipelining mechanism of SysX. It consists of two pipelines: a *GPU pipeline* and a *GPU-CPU pipeline*. The GPU pipeline's key idea is to assign the forward and backward passes of the same stage to different GPUs (*decoupled pass assignment*) for better pipeline scheduling. The GPU-CPU pipeline assigns the optimization steps to the CPU and executes them in parallel with the GPU pipeline for better utilization of heterogeneous resources.

3.1 GPU pipeline

In ordinary pipelining mechanisms, a micro-batch's forward and backward passes are assigned together to the same GPU. Figure 5(a) gives the pipeline diagram of a typical training pipeline with two model stages, S_0 and S_1, where the forward passes (f_0 and f_1) and backward passes (b_0 and b_1) of S_0 and S_1 are mapped to two different GPUs, GPU_0 and GPU_1, respectively. Suppose that we have two micro-batches in a mini-batch. At time t_1 for a micro-batch m_0, GPU_0 waits for GPU_1 to finish b_1^0 because the gradients computed by b_1^0 are necessary to proceed b_0^0. Unfortunately, GPU_1 executes f_1 for all micro-batches first, executes b_1^0, and finally transfers the gradients GPU_0 waits for at time t_2. As a result, GPU_0 remains idle from t_1 to t_2.

Decoupled pass assignment. SysX GPU pipeline is based on the observation that the pipeline bubbles can be reduced if a single micro-batch's forward and backward passes for the same stage are assigned to different GPUs. Figure 5(b) is an example of a decoupled pass assignment. While f_0 and f_1 are mapped to GPU_0 and GPU_1, respectively, b_0 and b_1 are mapped to GPU_1 and GPU_0, respectively. As GPU_0 is assigned b_1 instead of b_0, b_1^0 can start immediately at GPU_0 at t_2. We see less bubbles in Figure 5(b) than the ordinary pipeline in Figure 5(a).

Fetching parameters from non-GPU memory. With the decoupled approach, each GPU has to store all parameters for the stages assigned to it. For example, each GPU_0 and GPU_1 has to store the parameters Ψ_0 and Ψ_1 of all stages



(a) Pass assignment in an ordinary pipeline.

(b) Decoupled pass assignment.

(c) Fine-grained backward stage partitioning.

Figure 5: SysX GPU pipeline optimizations. For simplicity, we omit the data transfer time between GPUs. Arrows depict dependence between the passes.

S_0 and S_1 in its memory. However, when parameters are offloaded to non-GPU memory (e.g., the CPU memory), which is a common setting when training large models under GPU memory pressure, the same parameters can be fetched to different GPUs without permanently storing them redundantly on different GPUs' memory. Moreover, SysX mitigates the overhead of fetching parameters from non-GPU memory by prefetching in an overlapped manner with GPU computation. SysX makes a GPU only keep the GPU memory spaces for the current computation and the parameters being prefetched instead of storing all parameters for the stages assigned to the GPU. For example, consider Figure 5(b). At t_0, GPU_0 starts to fetch Ψ_1 required for b_1. Similarly, GPU_1 starts to fetch Ψ_0 at t_1. GPU_0 also frees Ψ_0 as soon as f_0 finishes at t_2.

Activation recomputation. A problem with the decoupled pass assignment in Figure 5(b) is that for all micro-batches, all activations of f_1 generated at GPU_1 have to be transferred to GPU_0 to perform b_1 and *vice versa* for f_0. To solve this problem, we adopt activation recomputation [5, 13, 17, 18]. Activation recomputation is a common setting when training large models under GPU memory pressure. For example, at time t_2 in Figure 5(b), activation recomputation allows only the activation checkpoint of S_1 to be transferred from GPU_1

to GPU_0 to perform b_1^0 instead of all activation tensors of S_1 including intermediate activation tensors generated from performing f_1^0.

Fine-grained backward stage partitioning. However, as shown in Figure 5(b), when the execution times of the forward and backward passes differ, it incurs pipeline bubbles. Based on this observation, we decompose backward stages into finer granularity to minimize the bubbles by balancing the execution times. Specifically, the backward pass b_i for model stage S_i can be decomposed into d backward passes, as shown in Equation 4 so that each $b_{i,k}$ computes the backward pass of $|S_i|/d$ Transformer blocks:

$$b_i = b_{i,0} \circ b_{i,1} \circ \cdots \circ b_{i,d-1}. \quad (4)$$

As $b_{i,k}$ only requires a portion of S_i's parameters for its computation (i.e., $1/d$ of Ψ_i), we denote such portion as $\Psi_{i,k}$. Figure 5(c) is the result of decomposing the backward stage in Figure 5(b) into two fine-grained backward stages. Similar to Figure 5(b), f_0 and f_1 are mapped to GPU_0 and GPU_1, respectively. However, $b_{1,1}$ and $b_{0,1}$ are mapped to GPU_0, and $b_{1,0}$ and $b_{0,0}$ are mapped to GPU_1. At t_3, GPU_1 can start $b_{1,0}^0$ immediately, reducing bubbles. Note that GPU_0 starts to fetch $\Psi_{1,1}$ and $\Psi_{0,1}$ at t_0 and t_2, respectively. GPU_1 starts to fetch $\Psi_{1,0}$ and $\Psi_{0,0}$ at t_1 and t_3, respectively.

Another effect of finer-grained backward stages is that it reduces the GPU memory usage by $1/d$ at the cost of increasing the number of activation checkpoint transfers by up to $\times d$. However, the number of activation checkpoint transfers does not strictly scale by factors of d. This is because, while the forward pass f_i on S_i generates activation checkpoints for $S_{i,0}, \cdots, S_{i,d-1}$ required for the backward passes $b_{i,0}, \cdots, b_{i,d-1}$, those activation checkpoints for the backward passes mapped to the same GPU as f_i do not need to be transferred. For example, Figure 5(c) requires the same number of activation checkpoint transfers as in Figure 5(b) because while $b_{1,1}$ and $b_{0,0}$ require activation checkpoint transfers from another GPU, $b_{1,0}$ and $b_{0,1}$ do not. Compared to the reduced bubbles and GPU memory savings, this results in marginal communication cost, which will be further optimized next.

Asynchronous checkpoint communication. We denote the activation checkpoint required for the recomputation during $b_{i,k}^j$ as $c_{i,k}^j$. Checkpoints $c_{i,0}^j, \cdots, c_{i,d-1}^j$ are generated during f_i^j. Each backward stage requires a checkpoint, while not all of them should be sent from other GPUs. For example, Figure 6 focuses on the relationship between f_0^0 and $b_{0,0}^0$, $b_{0,1}^0$ of Figure 5(c). GPU_0 performs f_0^0, which is micro-batch m_0's forward pass on S_0. f_0^0 generates checkpoints $c_{0,0}^0$ and $c_{0,1}^0$, which are used during the recomputation of $b_{0,0}^0$ and $b_{0,1}^0$, respectively. $c_{0,0}^0$ and $c_{0,1}^0$ are depicted as red and orange circles, respectively. As $b_{0,1}^0$ is also assigned to the same GPU_0, $c_{0,1}^0$ needs to be saved only on the memory of GPU_0

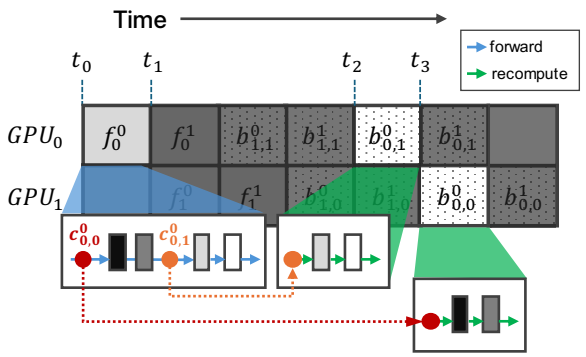


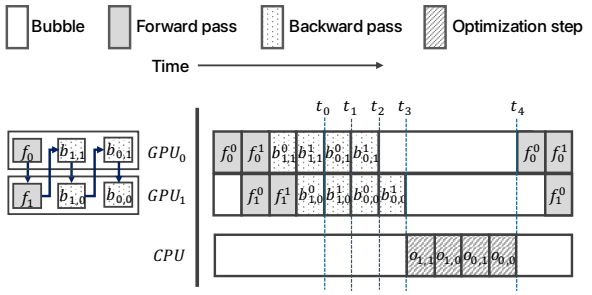
Figure 6: Asynchronous communication of checkpoints generated by the forward pass of stage S_0 for micro-batch m_0 from Figure 5(c). Red and orange circles are the checkpoints generated by the forward pass of stage S_0 for micro-batch m_0, required by the backward pass of fine-grained stages $S_{0,0}$ and $S_{0,1}$ for m_0, respectively. Only the red circle is sent asynchronously from GPU_0 to GPU_1 at t_0 and used at t_3.

until it is used at time t_2. However, $c_{0,0}^0$ should be sent from GPU_0 to GPU_1 before it is used at time t_3.

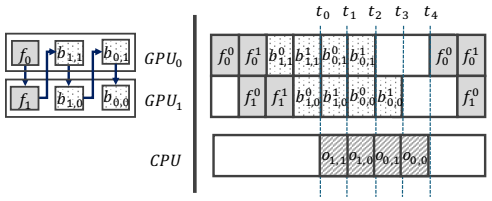
A naïve way to transfer $c_{0,0}^0$ from GPU_0 to GPU_1 would be to send and receive at time t_3 immediately before it is used for recomputation. In such a case, checkpoint communication lies on the critical path of the GPU pipeline along with the activation and gradient communication, adding a significant communication overhead. Instead, SysX transfers $c_{0,0}^0$ as soon as possible (at t_0) after $c_{0,0}^0$ is ready, overlapping its transfer with independent computations. This mechanism is enabled through asynchronous communication, which allows data transfer between GPUs to be initiated without waiting for completion. Hence, GPU_1 can use $c_{0,0}^0$ during $b_{0,0}^0$ without waiting, as $t_3 - t_0$ provides sufficient time for the checkpoint to arrive.

3.2 GPU-CPU pipeline

As explained in Section 2.3, in the hybrid GPU-CPU training [25,26], CPU optimization steps do not overlap with GPU computations, limiting its benefits. Figure 7(a) illustrates such a case on top of SysX's optimized pipeline with decoupled pass assignment and fine-grained backward stage partitioning, using the same setting of Section 3.1. At time t_0, GPU_0 finishes $b_{1,1}$ for all micro-batches and offloads the accumulated gradients to the CPU memory. The same process occurs for $b_{0,1}$ on GPU_0 at t_2, and $b_{1,0}$ and $b_{0,0}$ on GPU_1 at t_1 and t_3, respectively. Then, all offloaded gradients are validated for numerical stability, and the results are synchronized at t_3. If no overflows are found, the CPU proceeds by updating the parameters of all stages. The GPUs remain idle until all optimization steps are complete at t_4 to use the updated parameters for the next training iteration.



(a) CPU optimizer with gradients overflow synchronization.



(b) Asynchronous CPU optimizer.

Figure 7: SysX GPU-CPU pipeline optimization. $o_{i,k}$ denotes the optimization step of stage $S_{i,k}$. For simplicity, we omit the data transfer time between GPUs and GPU and CPU. Arrows depict dependence between the passes.

Asynchronous CPU optimizer. SysX's optimizer shifts numerical validations to a post-step process. Hence, a stage's CPU optimization step can proceed as soon as its GPU backward passes are complete and the gradients are offloaded. SysX overlaps the optimization step of a stage on the CPU with the subsequent stage's backward passes on the GPU to reduce GPU idle times. At the same time, correctness is ensured through a rollback mechanism following the post-step synchronization.

Consider Figure 7(b) that illustrates the CPU optimization steps of SysX. While GPU_0 offloads the gradients of $b_{1,1}$ at t_0 similar to Figure 7(a), the CPU immediately executes stage $S_{1,1}$'s optimization step $o_{1,1}$ at t_0 as it bypasses synchronization. The similar process is repeated for stage $S_{1,0}$, $S_{0,1}$, and $S_{0,0}$ at t_1, t_2, and t_3, respectively, as if the stages' backward passes on the GPUs and the optimization step on the CPU were pipelined. At t_4, when all parameter updates are complete, the gradients are finally checked for overflows, and the results are synchronized. If any overflow is detected, SysX performs a rollback of the updated parameters of all stages.

Bypassing and rollback mechanism. SysX pipelines the CPU optimization steps altogether with the GPU's computation by shifting the numerical validations after the parameter updates. Each stage performs its own local validation (i.e., checking gradient overflows) without waiting for the synchronization of results across all stages. Each stage executes its optimization step based on its own validation results. Synchronization finally occurs when all stages have completed their optimization steps. If any stage fails its local validation, parameter updates of all stages are rolled back. Optimizers, such as Adam [16] and AdamW [24], facilitate rollback without additional memory overhead because their parameter update steps are arithmetically reversible. While these rollbacks introduce some overhead compared to conventional pre-step validation, invalidations are rare during training and, therefore, have minimal impact on the overall training time [38, 39].

The CPU optimizer pipelining in Figure 7(b) shows an ideal scenario where the optimizer of each stage starts after the preceding stage has been completed. In practice, the optimizer for a subsequent stage may start before the previous stage has finished. Thus, each optimizer stage is processed in parallel on the CPU using multi-threading. Furthermore, for efficient pipelining, maximizing the overlap between the CPU optimization step and the backward passes is crucial. This can be done by selecting an appropriate number of micro-batches for each stage, allowing the backward passes to hide the CPU optimization step.

3.3 SysX Overall

Figure 8 illustrates the overall pipeline of SysX with $N_g = 4$ GPUs, $N_m = 4$ micro-batches, $N_{sf} = 8$ forward stages, and $N_{sb} = 16$ backward stages. Comparing SysX and Mobius in Figure 8 and Figure 2 with identical training settings, their bubble ratios are 25% and 47%, respectively, showing the benefit of SysX. There are two major reasons for this. One is that pipeline bubbles at the beginning of the backward passes are eliminated in SysX. The other is that CPU optimization steps are overlapped with GPU computation in SysX.

SysX further optimizes Mobius based on two key observations. One is that the coupled pass assignment mentioned in Section 2 is unnecessary when pipelining with offloading and activation recomputation. The other is that CPU optimization steps can operate in parallel by bypassing numerical validation.

4 Implementation

SysX is implemented on top of the Megatron-LM [33]. We modify its pipeline schedule, implement offloading, and integrate CPU optimization steps.

Pipeline schedule. SysX implements its pipeline schedule using separate CUDA streams for CPU-to-GPU data transfer, GPU computation, and GPU-to-CPU data transfer. Prefetching the next stage, performing forward/backward computations of the current stage, and offloading the gradients of the previous stage are thus processed in parallel. They are synchronized using CUDA events. While GPU-GPU communication of output activations and input gradients are sent and received synchronously, communication of

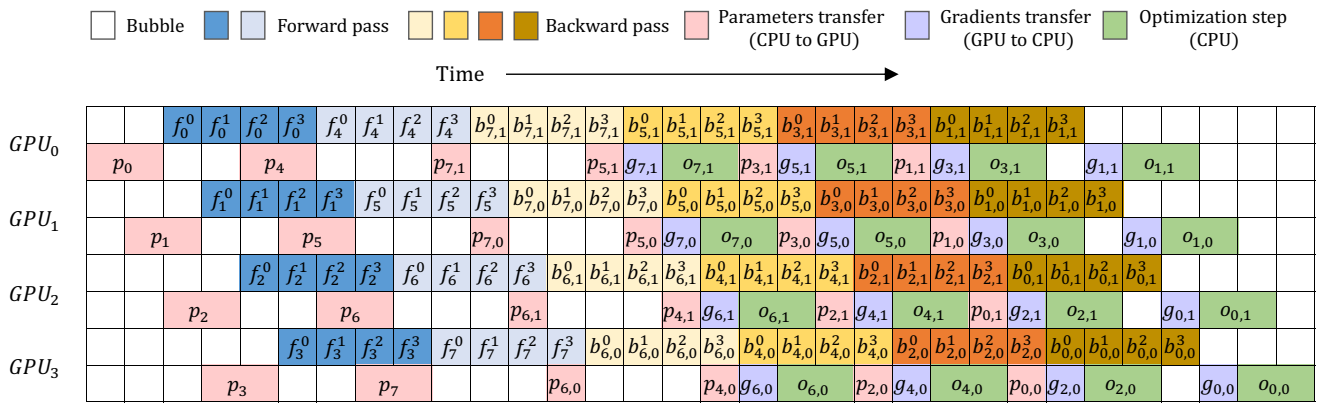


Figure 8: SysX pipeline schedule with four GPUs. GPU_i denotes the ith GPU. f_i^j denotes the ith forward stage's forward pass and $b_{i,k}^j$ denotes the finer-grained $(di+k)$th backward stage's backward pass on the jth micro-batch. Note that a backward pass has identical workload with a forward pass. Each forward/backward pass is distinguished by 2 and 4 different colors, respectively, to indicate that it belongs to a different forward/backward stage assigned to the same GPU. For example, GPU_0's forward passes on forward stage S_0 and S_4 are colored blue and sky blue, and backward passes on backward stage $S_{7,1}$, $S_{5,1}$, $S_{3,1}$, and $S_{1,1}$ are colored in light yellow, yellow, orange, and brown. p_i (or $p_{i,k}$) and $g_{i,k}$ denote CPU to GPU parameters transfer and GPU to CPU gradients transfer of forward stage S_i (or backward stage $S_{i,k}$) and backward stage $S_{i,k}$, respectively. $o_{i,k}$ denotes the optimization step of backward stage $S_{i,k}$ executed on the CPU.

activation checkpoints uses asynchronous P2P operations in `torch.distributed`. A checkpoint communication schedule is constructed so that both sender and receiver GPUs call `isend` and `irecv` at the same timestep, respectively, with the receiver GPU later synchronizing on the completion status of the returned handle. This checkpoint communication schedule is constructed once during the initialization phase and cached, as all training iterations use the same schedule.

Offloading. A common practice for multi-GPU training is launching one process for each GPU. When using CPU offloading, each GPU process can independently allocate CPU memory buffers to store the offloaded data. However, in SPipe, a single stage's forward and backward passes are executed on different GPUs, necessitating that multiple processes share the same model stage.

To address this, we allocate POSIX shared memory [36] for each node and adjust the tensor pointers to reference this shared memory. The offloaded parameters are physically shared and virtually mapped to the GPU processes assigned. Specifically, a GPU process responsible for a backward stage b_i allocates space for Ψ_i in the shared memory while another GPU process requiring Ψ_i for its forward stage f_i sets its pointers to the corresponding region in the shared memory.

For multi-node training, Remote Direct Memory Access (RDMA) is used to fetch parameters stored in the shared memory of a remote node. The shared memory size required per node is the sum of memory space required for all Ψ_i of b_i stages assigned to GPUs within that node. We observe that the memory space required for the model's parameters (e.g., 200GB for a model with 100 billion parameters, assuming mixed-precision training) divided by the number of nodes closely approximates the size of the shared memory per node.

CPU optimizer. We assign a separate CPU optimizer for each stage using C++ threading [4]. We pass a CUDA event that records the corresponding gradient offloading operation from a PyTorch main thread to the CPU optimizer thread, and the optimizer thread waits for the event completion before beginning the asynchronous optimization steps at the CPU to ensure correctness. Each CPU optimizer thread uses a CPU-based Adam [16] implementation of DeepSpeed [29] that leverages thread-level and instruction-level parallelism. We modify it to include the event synchronization mechanism, floating-point precision conversion, unscaling, and local numerical validations. After all CPU optimizers are complete, a post-step synchronization collects the local validation results, and rollback is triggered in case of invalidation. The rollback leverages Adam's arithmetic reversibility, and we implement it by modifying the optimization step implementation of DeepSpeed [29].

5 Evaluation

In this section, we evaluate SysX against existing approaches to train LLMs under memory pressure. We further examine the effectiveness of our optimizations and analyze the overheads of SysX, providing insights into its trade-offs and performance benefits.

Table 1: Node configuration of the eight-node cluster.

Motherboard	ASRock ROMED8-2T
CPU	1 x AMD 32-core EPYC 7452
Main Memory	8 x DDR4-2666 64GB
GPU	4 x NVIDIA Tesla V100 32GB PCIe
NIC	1 x Mellnox ConnectX-6 Infiniband HDR
PCIe	16 x Gen3 lanes per GPU
OS	Ubuntu 20.04.4 LTS (kernel 5.4.0-100)
GPU Driver	550.54.15
CUDA Version	12.4

5.1 Evaluation Environment

System configurations. Table 1 describes the system configuration for experiments. The cluster has eight nodes, targeting LLM training in a small GPU cluster setup. Each node has four NVIDIA Tesla V100 32GB GPUs connected through 16x PCIe Gen3, an AMD 32-core CPU, 512GB main memory, and an InfiniBand HDR NIC (200Gb/s).

Workloads. We use LLaMA2-based language models [49] of eight different sizes: 10B, 19B, 30B, 40B, 52B, 69B, 88B, and 110B. Table 2 summarizes their configurations. Models are trained using a varying number of nodes to reflect differences in their size: the (10B, 19B), (30B, 40B), (52B, 69B), and (88B, 110B) models are evaluated using 1, 2, 4, and 8 nodes, respectively. We use mixed-precision training [28] and OpenWebText [35] as the training dataset. We run five warmup iterations and average the measurements from the subsequent five iterations. Note that all experiments are conducted with activation recomputation [5, 13, 17, 18] because preserving all activations results in GPU out-of-memory (OOM) errors even for the smallest model size.

Variables. We vary model sizes, sequence lengths, and batch sizes to capture diverse configurations used in practice. Model size reflects the diverse scales of modern LLMs. Sequence length (SEQ) influences the model's ability to comprehend context. Batch size, a primary determinant of training efficiency, is also explored. In pipeline parallelism, the input mini-batch is divided into micro-batches. Increasing the size of micro-batches enhances GPU computational efficiency but is constrained by memory capacity. To address this, we scale the global mini-batch size by increasing the number of micro-batches. Both micro-batch size (MBS) and global mini-batch size (GBS) are investigated as variables in our experiments.

Baselines. Our baselines for comparison are Mobius [8] and Megatron [32]. Megatron is a widely used framework for training transformer models using an interleaved 1F1B (one forward, one backward) schedule. However, since it does not support offloading, even our smallest workload, the 10B model, encounters out-of-memory (OOM) on the GPUs. Thus, we extend its implementation to support CPU offloading. Mobius is a state-of-the-art pipeline framework with an interleaved AFAB (all forward, all backward) schedule optimized for offloading all training states. All the techniques, Mobius, Megatron, and SysX, offload parameters, gradients, and optimizer states to the CPU memory and exploits activation recomputation to minimize GPU memory usage. Due to the lack of public availability, we evaluate using our implementations of Mobius and offloading-extended Megatron. However, we verify the completeness of our implementations in the appendix. All experiments are conducted without incorporating any other parallelism strategies to contrast performance differences solely in pipeline parallelism.

Table 2: Configurations of the LLaMA-2 models used in the evaluation. Model sizes are on a scale of billion (B) parameters. Columns l, d, d_{FFN}, # KV heads, and # Nodes represent the number of Transformer layers, hidden dimension size, FFN layer's hidden dimension size, number of KV heads, and number of nodes used, respectively.

Model Size	l	d	d_{FFN}	# KV heads	# Nodes
10B	48	4,096	10,880	2	1
19B	48	5,632	14,976	4	1
30B	96	5,120	13,632	4	2
40B	96	5,888	15,680	4	2
52B	96	6,656	17,728	8	4
69B	96	7,680	20,480	8	4
88B	192	6,144	16,384	16	8
110B	192	6,912	18,432	16	8

5.2 Comparison

Figure 9 shows the speedups of Megatron and SysX over Mobius. In this experiment, we train all models in Table 2, each using the corresponding `# Nodes` nodes, global mini-batch size (GBS) of 16× `# Nodes`, fixed micro-batch size (MBS) of two, and sequence lengths (SEQ) of 1024 and 2048.

Overall, SysX achieves, on average, the speedups of **1.26** and **1.51** over Mobius and Megatron, respectively. SysX outperforms Mobius and Megatron across all cases. The performance gain from the GPU pipeline and GPU-CPU pipeline varies significantly with model sizes, sequence lengths, and batch sizes. In addition, the AFAB schedule of Mobius and SysX outperforms the 1F1B schedule of Megatron by efficiently hiding the offloading overhead. While Megatron closely matches Mobius's performance for large sequence lengths, SysX continues outperforming Megatron.

Model sizes. Comparing two model sizes for each node configuration with the same sequence length, the overall improvements of SysX remain consistent regardless of the model size. This is because, as the model grows, the time saved by reducing GPU pipeline bubbles and overlapping CPU optimization steps increases proportionally with the total iteration time. This demonstrates that SysX delivers consistent performance improvements across varying model sizes.

Sequence lengths. In most cases, the speedup is larger for smaller sequence lengths. This is because the computation re-

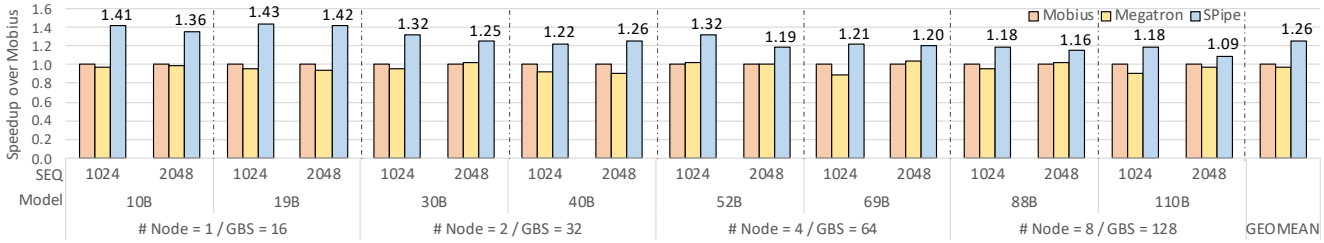


Figure 9: Speedups of SysX and Megatron over Mobius.

quired for the forward and backward passes increases quadratically with sequence length while the optimization step remains unaffected. As a result, the benefits of overlapping CPU optimization steps diminish as sequence length increases. In contrast, the GPU pipeline speedup remains constant because the size of pipeline bubbles also increases quadratically with sequence length.

Number of nodes and batch sizes. Due to the nature of pipelining, the minimum required number of micro-batches increases with the number of nodes. An increase in batch size leads to longer GPU computation time and causes it to dominate the total iteration time. Hence, the benefits of overlapping CPU optimization steps are less evident in the overall speedup. On the other hand, the speedup gained from reducing the pipeline bubble time remains constant because the pipeline depth also increases with the number of nodes along with the batch size, maintaining a steady bubble ratio.

5.3 Effect of the Batch Size

SysX's GPU pipeline effectively eliminates the bubbles in ordinary pipeline schedules. However, the performance benefit from bubble reduction is sensitive to the batch size. We conduct experiments in two scenarios: scaling the micro-batch size (MBS) and the global mini-batch size (GBS).

Scaling MBS. When the GBS is fixed, increasing MBS causes the computation required per micro-batch to grow linearly with MBS. Consequently, the size of pipeline bubbles also increases. As a result, Mobius experiences larger bubbles and a higher bubble ratio. This leads to an increase in the GPU pipeline speedup for SysX, as it effectively minimizes these bubbles. To analyze the individual effects of GPU pipeline bubble reduction and CPU optimization step overlapping, we break down the total speedup into the GPU pipeline speedup and CPU optimizer speedup.

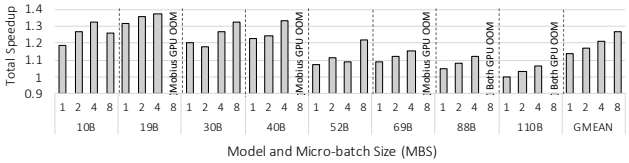
Figure 10 shows the results of MBS scaling. As the MBS varies with 1, 2, 4, and 8, the total speedup of SysX over Mobius also increases. They are, on average, 1.13, 1.17, 1.20, and 1.26, respectively. Similarly, when the MBS varies with 1, 2, 4, and 8, the GPU pipeline speedup over Mobius becomes at 1.00, 1.03, 1.07, and 1.12, respectively, validating larger gains from the efficient pipeline schedule of SysX.

On the other hand, scaling the MBS has no impact on CPU optimizer speedup because it does not affect the total GPU computation time per iteration, and the overlapping time for optimization steps remains unchanged. As the MBS scales from 1, 2, 4, and 8, the CPU optimizer speedup over Mobius remains constant with averages of 3.68, 3.66, 3.67, and 3.61, respectively.

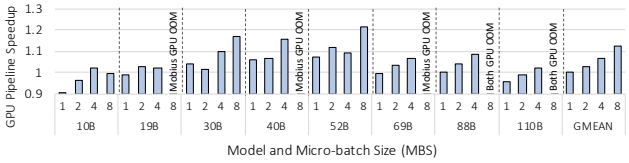
Additionally, SysX demonstrates its capability to train the models with a larger MBS than Mobius because only SysX successfully trained the 19B, 40B, and 69B models with an MBS of 8. It is the result of less GPU memory consumption during the backward pass caused by SysX's fine-grained backward-stage partitioning.

Scaling GBS. When the MBS is fixed, increasing the number of micro-batches translates to a larger GBS. This increases the overall execution time of the pipeline on a GPU while the idle time consumed by the bubbles remains constant. Hence, scaling the GBS results in a lower bubble ratio of Mobius and eventually reduces GPU pipeline speedup of SysX. Figure 11 shows the results of GBS scaling. As the GBS scales by factors of 1, 2, 3, and 4, the total speedup over Mobius decreases. The average speedups are 1.32, 1.26, 1.22, and 1.19, respectively. Similarly, the GPU pipeline speedup over Mobius also decreases. The average speedups are 1.15, 1.08, 1.04, and 1.05, respectively.

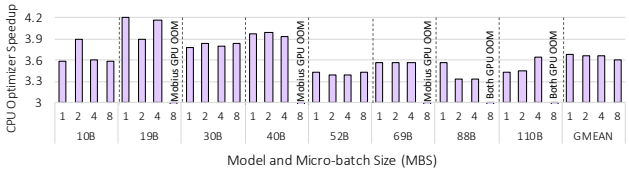
On the other hand, scaling GBS results in a better overlap of the optimization step because the optimization step may not overlap well with smaller GBS values. For example, in the case of the 10B model, increasing the GBS values with 8, 16, 24, and 32 increases the average CPU optimizer speedup over Mobius with 1.43, 2.49, 3.57, and 3.90, respectively. When GBS = 24, the speedup stabilizes. We observe a saturation point of CPU optimizer speedup exists in each model size, which is when the time required to process the optimization step of a stage on the CPU equivalents with the time required to process the backward pass of all micro-batches of the stage. For example, in 30B model, such a point is when the GBS is scaled to 32. When such a saturation point is reached, all optimization steps of the backward stages have already been fully overlapped, leaving only the latest processed backward stage to be run non-overlapped in SysX. As the GBS scales by factors of 1, 2, 3, and 4, the average speedup of the CPU



(a) Total speedup over Mobius.

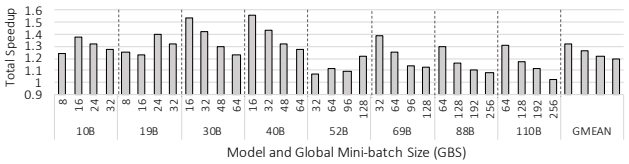


(b) GPU pipeline speedup over Mobius.

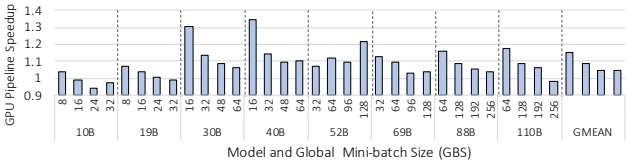


(c) CPU optimizer speedup over Mobius.

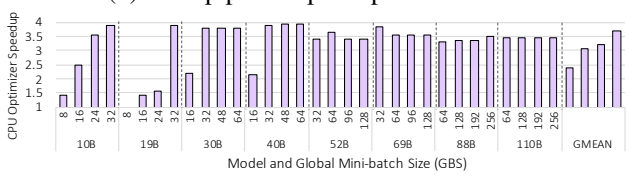
Figure 10: Effect of micro-batch size (MBS) scaling.



(a) Total speedup over Mobius.



(b) GPU pipeline speedup over Mobius.



(c) CPU optimizer speedup over Mobius.

Figure 11: Effect of global mini-batch size (GBS) scaling.

optimizer step increases with the values 2.38, 3.07, 3.22, and 3.68, respectively.

5.4 Effect of Various Optimizations

SysX proposes several optimization techniques to achieve an efficient pipelining scheme. We analyze the impact of these optimizations. As shown in Table 3, we decompose the proposed optimizations into distinct steps and evaluate various SysX configurations by incrementally incorporating the proposed techniques.

We first compare the different optimization configurations for a given model. Figure 12 illustrates the breakdown of the iteration time per configuration. To distinctly evaluate the effect of each technique, we partition the training iteration time into two components: the *GPU time* and the *non-overlapping CPU time*. The GPU time refers to the time spent on forward and backward computations, and the non-overlapping CPU time represents the remaining time spent on the CPU optimizer, excluding GPU computation. We see that 19B and 69B models have very different proportions of the GPU time and non-overlapping CPU time. For the 19B model, the GPU time constitutes 41.3%, while for the 69B model, GPU time constitutes 77.8% of the iteration time on average. Thus, the impact of each optimization configuration on the iteration time varies largely between the two models.

Figure 12 shows that the optimizations progressively applied in CFG0, CFG1, and CFG2 reduce GPU time. This is because the bubbles are reduced, and the extra communication overhead for activation checkpoints is effectively hidden. CFG3 reduces the non-overlapping CPU time by overlapping

Table 3: Evaluated SysX optimization configurations

Config.	SysX features included
CFG0	Decoupled pass assignment
CFG1	CFG0 with fine-grained backward stage partitioning
CFG2	CFG1 with optimized activation checkpoint communication
CFG3	CFG2 with asynchronous CPU optimizer
CFG4	CFG3 with ideal communication overhead

the CPU optimization step of the previous backward stage with the current stage's backward pass on GPUs. It also shows that CFG3 closely matches the ideal performance, which is CFG4, indicating that SysX implementation nearly reaches the optimal pipeline and CPU optimizer performance.

5.5 Recomputation Overhead

SysX currently requires activation recomputation because of the decoupled pass assignment, while Mobius can selectively apply activation recomputation. While this paper focuses on scenarios with insufficient aggregated GPU memory to store all model states, including activations, we experiment with the case for smaller models for which Mobius can train without activation recomputation.

Figure 13 compares SysX (with recomputation) and Mobius (without recomputation) to evaluate the overhead of activation recomputation in SysX. Experiments were conducted on a single node using smaller models up to 5B parameters, a sequence length of 2048, a micro-batch size of 1, and a global mini-batch size of 16. SysX shows the speedup of 0.7 for the 400M model and 0.85 for the 1.4B model. However, Mobius encounters GPU out-of-memory (OOM) for models larger

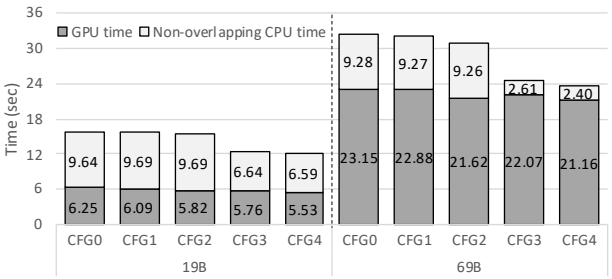


Figure 12: Impact of progressively adding system features to SysX system configurations.

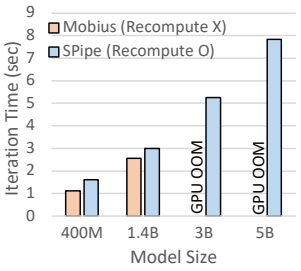


Figure 13: Recomputation

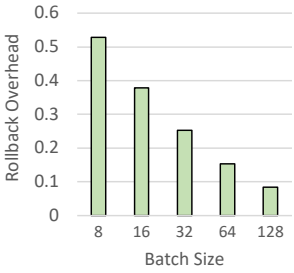


Figure 14: Rollback overhead

than 1.4B parameters, making direct comparisons for larger models infeasible.

While SysX shows lower performance compared to Mobius without recomputation for smaller models, these cases are not the focus of this study, as models of such size do not benefit significantly from offloading techniques. SysX is designed to address memory and scalability challenges in larger models, where recomputation becomes indispensable. This is particularly important for the AFAB (all forward, all backward) schedule, where the absence of activation recomputation would necessitate retaining activations for all micro-batches in the GPU memory the backward pass completes.

5.6 Rollback Overhead

SysX bypasses optimizer synchronization while ensuring numerical stability through post-validation. If validation fails, parameters are reverted to their pre-update state using the rollback algorithm. However, the rollback process does not overlap with GPU computation, leading to some overhead. The rollback overhead includes both the time spent on the rollback and the time of the non-overlapping optimization step that would have been skipped. Figure 14 shows the rollback overhead for a 10B model across different batch sizes, demonstrating that the overhead varies significantly from 8% to 53% depending on the batch size.

To assess the frequency of rollbacks during training, we set the initial scale factor to 2^{32} and trained a 10B model for 4,500 iterations, resulting in 20 rollbacks. Notably, rollbacks occurred during the first 11 iterations, which could have been avoided with a lower initial scale factor. Even in a conservative scenario where rollback occurs once every 100 iterations for a batch size of 8, the resulting overhead is only 0.53%, which is negligible compared to the speedup SysX achieves, making the rollback overhead an acceptable trade-off in the context of SysX's overall performance benefits.

6 Conclusion

This paper presents SysX, a hybrid GPU-CPU pipelining mechanism that enables efficient LLM training using pipeline parallelism with offloading to overcome insufficient aggregate GPU memory. SysX consists of two pipelines: a GPU pipeline and a GPU-CPU pipeline. SysX's GPU pipeline presents a novel pipeline scheduling scheme that decouples a stage's forward and backward passes for the same micro-batch to different GPUs by leveraging the CPU's shared memory and activation recomputation. It further optimizes its pipeline stages through fine-grained model partitioning that balances the passes' execution times and asynchronous checkpoint communication that hides the additional communication overhead. SysX's GPU-CPU pipeline presents an asynchronous CPU optimizer that executes the optimization steps on the CPU in parallel with the GPU pipeline stages. It efficiently utilizes the CPU to overlap the CPU optimizer overhead while guaranteeing the training correctness through a post-step validation and roll-back mechanism. Evaluation results of SysX show that it outperforms Mobius and the offloading-extended version of Megatron with average speedups of 1.26 and 1.51, respectively. We will make SysX publicly available to broaden the accessibility of large-scale model training.

References

[1] Jonghyun Bae, Jongsung Lee, Yunho Jin, Sam Son, Shine Kim, Hakbeom Jang, Tae Jun Ham, and Jae W Lee. FlashNeuron: SSD-enabled large-batch training of very deep neural networks. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 387–401, 2021.

[2] Tal Ben-Nun and Torsten Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys (CSUR)*, 52(4):1–43, 2019.

[3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda

Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[4] C++. std::thread. https://cplusplus.com/reference/thread/thread/, 2024.

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

[6] Daning Cheng, Shigang Li, Hanping Zhang, Fen Xia, and Yunquan Zhang. Why dataset properties bound the scalability of parallel machine learning training algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 32(7):1702–1712, 2021.

[7] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, Lansong Diao, Xiaoyong Liu, and Wei Lin. DAPPLE: A pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 431–445, 2021.

[8] Yangyang Feng, Minhui Xie, Zijie Tian, Shuo Wang, Youyou Lu, and Jiwu Shu. Mobius: Fine tuning large-scale models on commodity GPU servers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 489–501, 2023.

[9] Swapnil Gandhi, Mark Zhao, Athinagoras Skiadopoulos, and Christos Kozyrakis. ReCycle: Resilient training of large DNNs using pipeline adaptation. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 211–228, 2024.

[10] Mark Hildebrand, Jawad Khan, Sanjeev Trika, Jason Lowe-Power, and Venkatesh Akella. AutoTM: Automatic tensor movement in heterogeneous memory systems using integer linear programming. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 875–890, 2020.

[11] Chien-Chin Huang, Gu Jin, and Jinyang Li. 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*, pages 1341–1355, 2020.

[12] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and Zhifeng Chen. GPipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

[13] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. Checkmate: Breaking the memory wall with optimal tensor rematerialization. *Proceedings of Machine Learning and Systems*, 2:497–511, 2020.

[14] Hai Jin, Bo Liu, Wenbin Jiang, Yang Ma, Xuanhua Shi, Bingsheng He, and Shaofeng Zhao. Layer-centric memory reuse and data migration for extreme-scale deep learning on many-core architectures. *ACM Transactions on Architecture and Code Optimization (TACO)*, 15(3):1–26, 2018.

[15] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

[16] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[17] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. Dynamic tensor rematerialization. In *International Conference on Learning Representations*, 2021.

[18] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems*, 5, 2023.

[19] Tung D Le, Haruki Imai, Yasushi Negishi, and Kiyokuni Kawachiya. TFLMS: Large model support in TensorFlow by graph rewriting. *arXiv preprint arXiv:1807.02037*, 2018.

[20] Shigang Li and Torsten Hoefler. Chimera: Efficiently training large-scale neural networks with bidirectional pipelines. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

[21] Zhenxing Li, Qiang Cao, Yajie Chen, and Wenrui Yan. CoTrain: Efficient scheduling for large-model training upon GPU and CPU in parallel. In *Proceedings of the 52nd International Conference on Parallel Processing*, pages 92–101, 2023.

[22] Bo Liu, Wenbin Jiang, Hai Jin, Xuanhua Shi, and Yang Ma. Layrub: layer-centric GPU memory reuse and data migration in extreme-scale deep learning systems. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 405–406, 2018.

[23] Ziming Liu, Shenggan Cheng, Haotian Zhou, and Yang You. Hanayo: Harnessing wave-like pipeline parallelism for enhanced large model training efficiency. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13, 2023.

[24] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2019.

[25] Avinash Maurya, Jie Ye, M Mustafa Rafique, Franck Cappello, and Bogdan Nicolae. Breaking the memory wall: A study of i/o patterns and GPU memory utilization for hybrid CPU-GPU offloaded optimizers. In *Proceedings of the 14th Workshop on AI and Scientific Computing at Scale using Flexible Computing Infrastructures*, pages 9–16, 2024.

[26] Avinash Maurya, Jie Ye, M Mustafa Rafique, Franck Cappello, and Bogdan Nicolae. Deep optimizer states: Towards scalable training of transformer models using interleaved offloading. In *Proceedings of the 25th International Middleware Conference*, pages 404–416, 2024.

[27] Meta. LLaMA3. https://llama.meta.com/llama3/, 2024.

[28] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.

[29] Microsoft. DeepSpeed. https://www.deepspeed.ai/, 2024.

[30] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. PipeDream: Generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[31] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-efficient pipeline-parallel DNN training. In *International Conference on Machine Learning*, pages 7937–7947. PMLR, 2021.

[32] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 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*, pages 1–15, 2021.

[33] NVIDIA. Megatron-LM. https://github.com/NVIDIA/Megatron-LM, 2024.

[34] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. 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*, pages 891–905, 2020.

[35] Joshua Peterson, Stephan Meylan, and David Bourgin. OpenWebText. https://github.com/jcpeterson/openwebtext#openwebtext, 2019.

[36] POSIX. The Open Group Base Specifications. https://pubs.opengroup.org/onlinepubs/9699919799/, 2024.

[37] Bharadwaj Pudipeddi, Maral Mesmakhosroshahi, Jinwen Xi, and Sujeeth Bharadwaj. Training large neural networks with constant memory using a new execution algorithm. *arXiv preprint arXiv:2002.05645*, 2020.

[38] Penghui Qi, Xinyi Wan, Guangxing Huang, and Min Lin. Zero bubble pipeline parallelism. *arXiv preprint arXiv:2401.10241*, 2023.

[39] Penghui Qi, Xinyi Wan, Guangxing Huang, and Min Lin. Zero bubble (almost) pipeline parallelism. In *The Twelfth International Conference on Learning Representations*, 2024.

[40] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[41] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16, 2020.

[42] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 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*, pages 1–14, 2021.

[43] Jie Ren, Jiaolin Luo, Kai Wu, Minjia Zhang, Hyeran Jeon, and Dong Li. Sentinel: Efficient tensor migration and allocation on heterogeneous memory systems for deep learning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 598–611. IEEE, 2021.

[44] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. ZeRO-Offload: Democratizing billion-scale model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 551–564, 2021.

[45] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[46] Murray Shanahan. Talking about large language models. *Communications of the ACM*, 67(2):68–79, 2024.

[47] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[48] 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. LLaMA: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[49] 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. LLaMA 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[50] Guanhua Wang, Masahiro Tanaka, Xiaoxia Wu, Lok Chand Koppaka, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. DeepSpeed ZeRO-Offload++: 6x higher training throughput via collaborative CPU/GPU twin-flow. https://github.com/microsoft/DeepSpeed/tree/offloadppnews/blogs/deepspeed-offloadpp, 2024.

[51] Yang You, Jonathan Hseu, Chris Ying, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large-batch training for LSTM and beyond. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16, 2019.

[52] Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large batch optimization for deep learning: Training BERT in 76 minutes. *arXiv preprint arXiv:1904.00962*, 2019.

[53] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. ImageNet training in minutes. In *Proceedings of the 47th International Conference on Parallel Processing*, pages 1–10, 2018.

[54] 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. OPT: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

[55] Weigang Zhang, Biyu Zhou, Xuehai Tang, Zhaoxing Wang, and Songlin Hu. MixPipe: Efficient bidirectional pipeline parallelism for training large-scale models. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2023.

[56] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023.

[Appendix] SysX: System for Training under Memory Constraints

1 Validation of Baseline Implementation

In Section 5 Evaluation, the baselines are Mobius [1] and offloading-extended Megatron [3]. Due to the lack of public availability, we compare using our implementations for both. For SysX, Mobius, and Megatron, we modify the widely used Megatron-LM [4] framework to implement individual pipeline schedules. To efficiently leverage offloading to the CPU memory, all use separate CUDA streams for CPU-to-GPU data transfer, GPU computation, and GPU-to-CPU data transfer. We integrate CPU Adam implementation of DeepSpeed [2] in the CPU optimizer. Mobius and Megatron maintain a single CPU optimizer for each GPU that first performs pre-step synchronization and then executes all optimization steps for all its assigned stages. In contrast, SysX assigns a separate CPU optimizer thread to each backward stage to execute the optimization steps in parallel with the GPU pipeline stages and finally performs post-step synchronization and roll-back if necessary.

We demonstrate that each baseline performs as expected using profiled results obtained from NVIDIA Nsight Systems. We show the results for a 19B model, which is one of the LLaMA-2 models used in the evaluation, run on a single node with four NVIDIA V100 32GB GPUs. We use a global mini-batch size (GBS) of 8, a micro-batch size (MBS) of 1, and a sequence length of 2048.

Figure 1, Figure 2, and Figure 3 are the profiled results for Megatron, Mobius, and SysX, respectively. The profiled results capture GPU forward/backward compute kernels and CPU optimization steps. Each result shows the timeline of operations across four processes, as a single process is launched for a GPU. Each operation is visualized using annotations from NVTX [5]. We capture them over the same time period of 22 seconds.

The annotations in the profiling results are interpreted as follows:

- Each process is denoted as P_i.
- Line ①: Shows the utilization of CPU cores.
- Line ②: Displays GPU kernels and memory transfer operations. Specifically, *mint* indicates parameter prefetching, while *pink* represents gradient offloading. These data transfers are effectively overlapped with GPU computations.
- Line ③: Highlights forward passes in *blue series* and backward passes in *yellow series*. Notably, the backward pass includes activation recomputation. The forward and backward passes of different stages assigned to a single GPU are distinguished by different colors.
- Line ④: Highlights CPU optimization step in *green*. In SysX, *green* indicates non-overlapping CPU optimization step.
- Line ⑤: Unique to SysX, which shows the asynchronous execution of optimizers for each backward stage.

The profiled results confirm that both baselines operate as expected according to their intended pipeline schedules, with data transfers effectively overlapped. We also verified that the model converged to the same loss value using identical initial weights. We ensure the reliability of our comparative analysis, as both baselines faithfully reflect the original designs and demonstrate correctness.

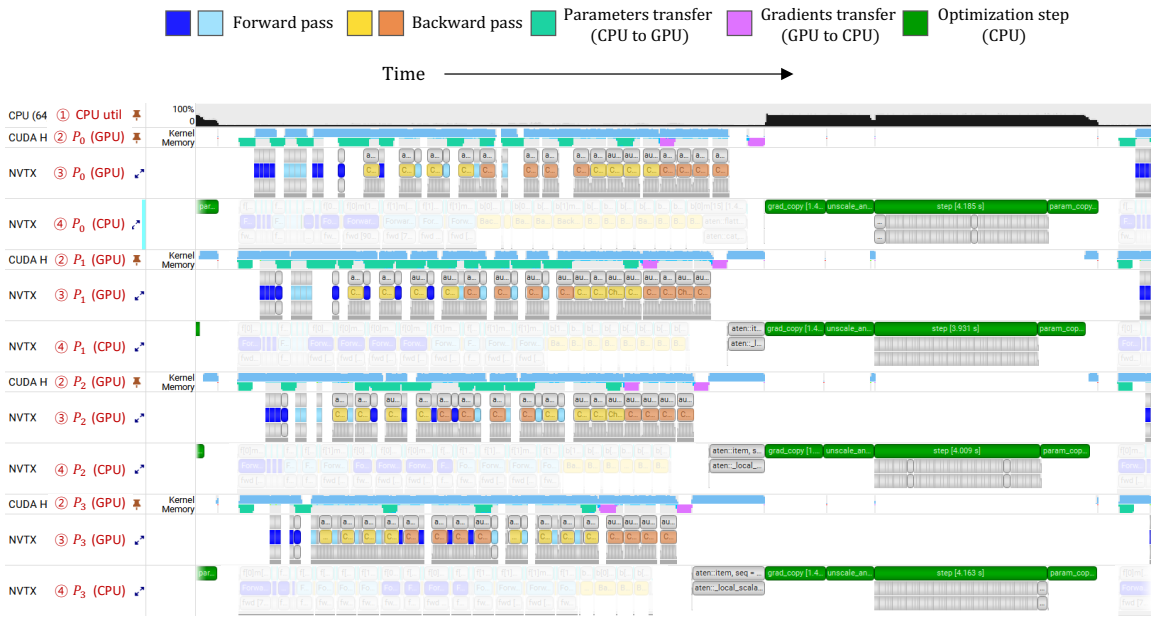


Figure 1: Profiled result of offloading-extended Megatron. It shows a clear interleaved 1F1B (one forward, one backward) schedule, as expected for Megatron. Each GPU handles two forward stages and two backward stages. It overlaps GPU-CPU data transfers with GPU computation. Due to the nature of the 1F1B schedule, it requires frequent parameter prefetching for different stages, which causes noticeable bubble times. The optimization steps do not overlap with the GPU pipeline, as shown by the green squares, and there is negligible CPU utilization during the GPU computation.

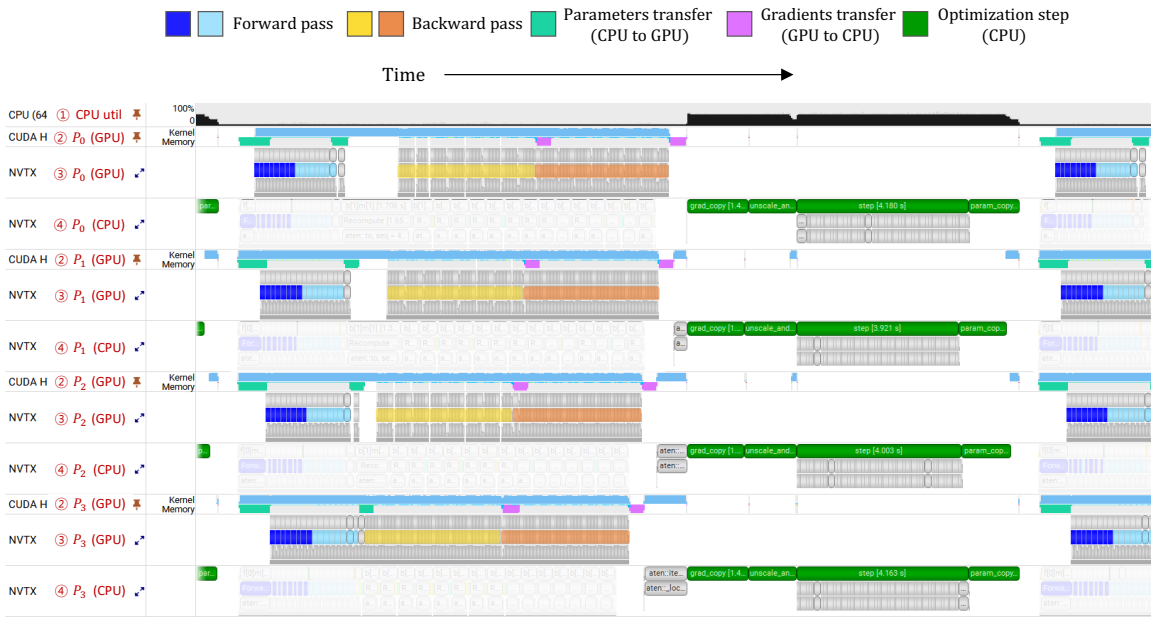


Figure 2: Profiled result of Mobius. It shows the AFAB (all forward, all backward) schedule, where GPU-CPU data transfers overlap with GPU computation, as expected for Mobius. Each GPU handles two forward and two backward stages, with parameters of the next start being prefetched at the start of the current stage. Mobius's coupled pass assignment leads to bubble times, which occur as expected. The optimization steps do not overlap with the GPU pipeline, as shown by the green squares, and there is negligible CPU utilization during the GPU computation.

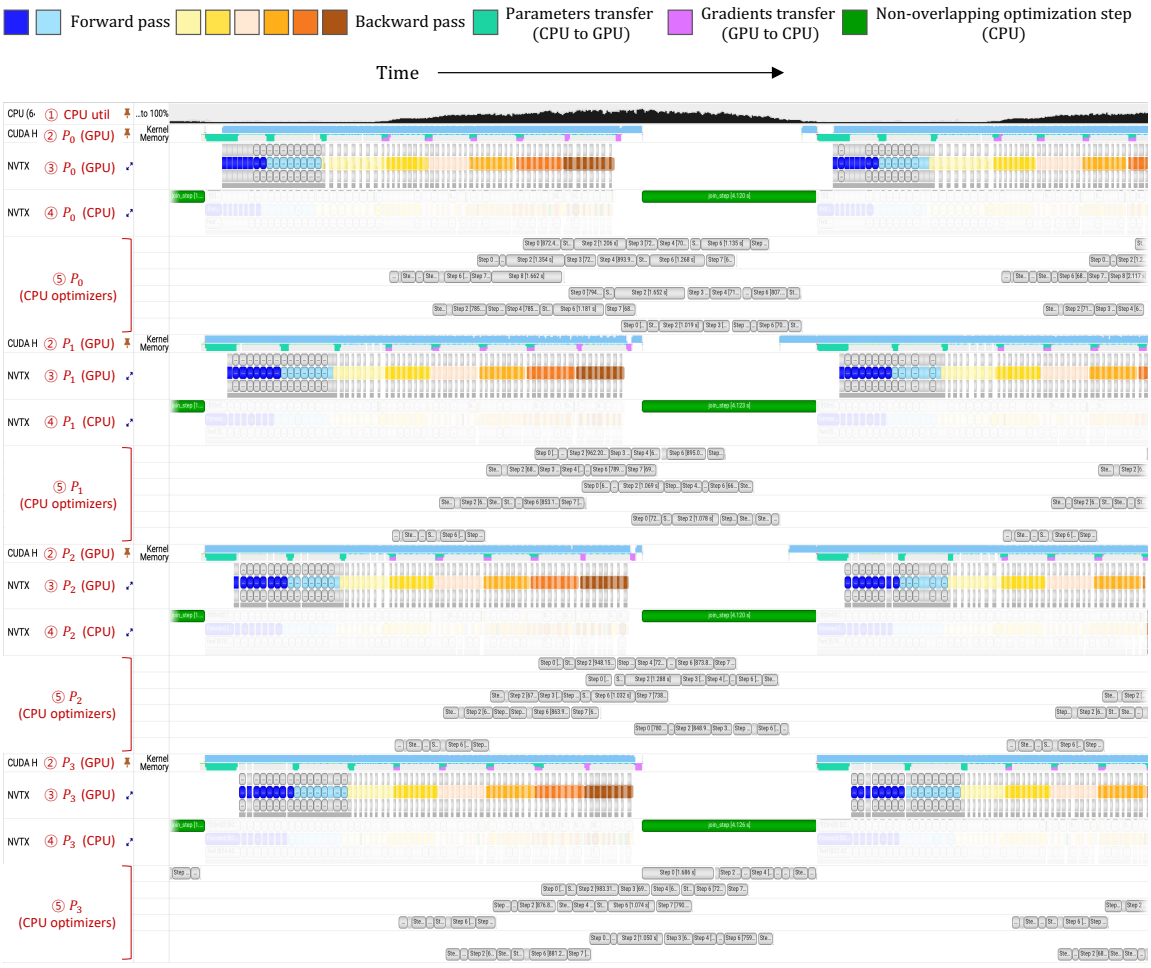


Figure 3: Profiled result of SysX. Each GPU are assigned two forward stages which are further partitioned into six backward stages. Its GPU pipeline shows that eight micro-batches are first pipelined across the first forward stage assigned to each GPU, followed by the second forward stage. Then six different backward stages are pipelined similarly. These forward and backward passes have similar execution time due to fine-grained backward stage partitioning, and the bubble times are significantly reduced due to decoupled pass assignment. SysX also employs an asynchronous CPU optimizer. The six optimizers, one for each backward stage, are executed independently. Each optimization step includes event synchronization, floating-point precision conversion, unscaling, and local numerical validations.

[1] Yangyang Feng, Minhui Xie, Zijie Tian, Shuo Wang, Youyou Lu, and Jiwu Shu. Mobius: Fine tuning large-scale models on commodity GPU servers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 489–501, 2023.

[2] Microsoft. DeepSpeed. `https://www.deepspeed.ai/`, 2024.

[3] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 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*, pages 1–15, 2021.

[4] NVIDIA. Megatron-LM. `https://github.com/NVIDIA/Megatron-LM`, 2024.

[5] NVIDIA. NVTX. `https://github.com/NVIDIA/NVTX`, 2024.