SmartSwap: Swap-Based Memory Optimization for LLM Training under Varying Operator Sequences

Zibo Wang[1], Yuhang Zhou[1], Zhibin Wang[1]*, Shipeng Li[1], Xinjing Huang[2], Chendong Cai[2], Bingxu Mu[2], Yuqing Sun[2], Zhiheng Hu[2], Bin She[2], Shu You[2], Guanghuan Fang[2], Rong Gu[1], Wanchun Dou[1], Guihai Chen[1], Chen Tian[1]
[1]*State Key Laboratory for Novel Software Technology, Nanjing University* [2]*Huawei Technologies Co., Ltd*

Abstract

The increasing size of large language models (LLMs) has led to a surge in memory requirements during training, often exceeding the capacity of high-bandwidth memory (HBM). Swap-based memory optimization incurs neither accuracy loss nor additional end-to-end overhead when effectively overlapped, thus being an attractive solution. However, existing swap methods assume consistent operator sequences, which is impractical in Eager Mode, where operator sequences can vary across iterations.

We propose SmartSwap, which redesigns the end-to-end process of swap-based memory optimization and is the first work to consider varying operator sequences in Eager Mode. SmartSwap (i) introduces a lightweight online profiler to enable continuous profiling for monitoring operator sequences, (ii) generates effective swap policies with limited operator information, and (iii) optimizes the policy execution module for accurate policy application and better performance. Experimental results demonstrate that SmartSwap reduces profiling overhead by 84.25%, enables training models up to 4× larger than hardware memory while adapting to changes in operator sequences, and improves performance by up to 38.94% compared to recomputation or high-degree parallelism.

1 Introduction

In recent years, large language models (LLMs) [5, 9, 10, 25, 30, 39] have emerged as a vibrant research domain due to their remarkable capabilities. To solve more complex problems, researchers have primarily focused on increasing the number of model parameters, which has been proven to be an effective strategy [16, 18].

However, the explosive growth in model sizes has made it impossible to complete model training with the limited HBM of a single AI accelerator. There are many methods to alleviate memory limitations, such as parallelization [8, 36, 37, 47, 50, 58], compression [40, 53], sparsity [23, 44, 48], recomputation [11, 43, 51], etc., which are mutually orthogonal and can be used in combination [26, 29, 32, 38, 41, 45, 49]. Among them, swap is an ideal memory optimization technique. It involves swapping dynamic memory (i.e., activations) to the host DRAM when they are not used for a long period to free up HBM and swapping them back to the device before the next use to ensure uninterrupted training [28].

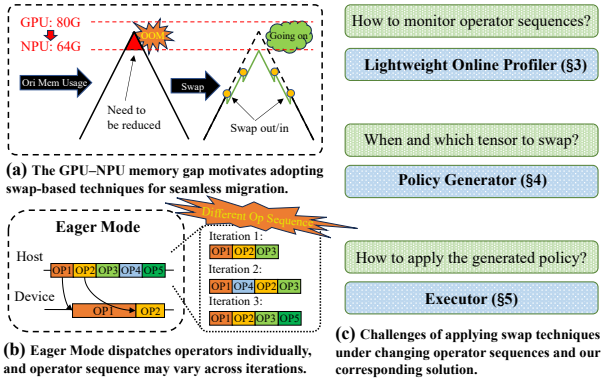


Figure 1: Leveraging swap-based techniques for seamless migration of PyTorch training from GPU to NPU.

Mainstream swap techniques [7, 19–22, 26, 28, 34, 35, 45, 49, 54] typically operate under the implicit assumption that the operator sequence remains unchanged. This is mainly because these methods were designed for the early popular Graph Mode frameworks, such as TensorFlow 1.x [27], MindSpore [52], and others [12, 17, 42, 46], where the training computation is compiled into a static computation graph that is dispatched once and reused throughout training. This static structure enables holistic optimizations such as computation fusion and tensor swapping, and many prior works have explored swap-based methods under this paradigm [7, 21, 26]. However, the complexity of debugging and deployment has led to a gradual shift away from Graph Mode frameworks, motivating the rise of Eager Mode frameworks like Pytorch [6] that favor flexibility and ease of development. As of December 2024, only 2% of open-access implementations of machine learning papers used TensorFlow, while 60% of implementations adopted PyTorch [1].

Motivating Example: Bridging the GPU–NPU Memory Gap for Seamless PyTorch Training. Developers typically prototype and train models in PyTorch on GPUs, leveraging Eager Mode for rapid iteration. Ascend NPUs now provide near drop-in PyTorch support [2], but migration often encounters a memory downgrade—for example, A100's 80 GB HBM versus Ascend 910B's 64 GB. This 16 GB gap can force changes to model parallelization or device counts, breaking otherwise stable workflows. We address this gap with a swap-based memory optimization that enables PyTorch workloads to run on lower-capacity NPUs without model refactoring or noticeable performance loss. Although motivated by GPU→NPU migration, the same approach applies to transitions across GPU generations with differing memory capacities.

However, the inherent flexibility of Eager Mode frameworks renders existing swap techniques ineffective. Prior works [7, 21, 26, 45, 49] follow a typical *profiling* → *policy generation* → *policy*

*Zhibin Wang is the corresponding author.

DAC '26, Long Beach, CA, USA

ACM ISBN 979-8-4007-2254-7/2026/07
https://doi.org/10.1145/3770743.3803980

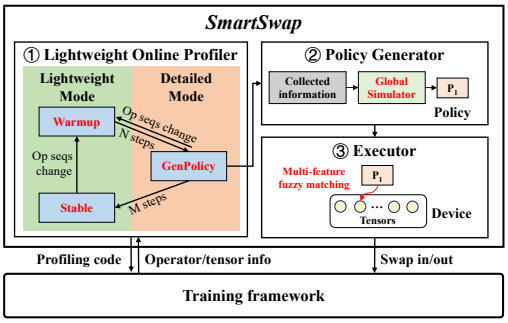


Figure 2: Overview of SmartSwap and its workflow.

application workflow built on the assumption of consistent operator sequences in Graph Mode frameworks. They profile operator and tensor behaviors from a single iteration, generate a one-time swap policy that determines which tensors to swap and when, and then directly apply this policy to subsequent iterations. This design works well under Graph Mode, where operator sequences and tensor identifiers remain stable. However, in Eager Mode, each operator is dispatched individually to the device and executed sequentially but at different paces with respect to the host. Dynamic features [13, 15, 24, 31, 56] bring varying operator sequences across different training iterations, creating conflicts with the prior assumption. This makes the policy generated for earlier sequences invalid and may not only decrease the efficiency of memory optimization but also threaten the reliability of the training system.

To enable efficient swapping in Eager Mode frameworks, we propose Smart-Swap, the first system that systematically supports varying operator sequences in swap-based memory optimization. As in Fig. 2, SmartSwap also follows the *Profiling* → *Policy generation* → *Policy application* workflow, comprising three modules—Lightweight Online Profiler, Policy Generator, and Executor—that address challenges across the full swap workflow, which are detailed as follows.

First, *in the profiling phase, the profiling tools of existing works [7, 21, 26, 45, 49] cannot meet our requirements for lightweight, online analysis of varying operator sequences.* Sequence changes can invalidate generated swap policies, causing suboptimal memory use or runtime errors. Existing profilers (i) impose high overhead-e.g., increasing iteration time from 4.9 s to 15.7 s, a 219% slowdown-and (ii) lack online support, requiring pausing training and defining profiling boundaries manually. To address this, we implement a **Lightweight Online Profiler (§3)** that continuously monitors operator sequences with minimal overhead. It supports two modes: Lightweight and Detailed, and a stage-adjusting module that transitions among WarmUp, GenPolicy, and Stable stages, automatically triggering new swap policy generation when sequences change.

Second, *in the policy generation phase, we face a trade-off between profiling overhead and policy performance.* To enable continuous online profiling, we exclude high-cost information such as per-operator execution times. However, this missing information complicates the generation of effective swap policies, as accurate timing for pre-triggering swap-in operations typically depends on operator-level timing data. We observe that dividing the operator sequence into evenly sized groups greatly reduces timing variance, allowing the average group time to serve as a low-error proxy for execution time. To balance profiling cost and policy quality, we leverage this insight to design a lightweight **Policy Generator (§4)** that partitions the operator sequence into logical layers and determines swap operations at this granularity. Moreover, we introduce a global swap simulator to determine precise swap-out and swap-in timings from a global perspective, minimizing performance degradation while maintaining low profiling overhead.

Third, *in the policy application phase, accurately and efficiently applying the generated policy to subsequent iterations is also challenging*, since Eager Mode frameworks recompile and dispatch operators in each iteration without providing unique identifiers to track them across runs. We design an **Executor (§5)** that uses multi-feature fuzzy matching to locate corresponding operators and tensors across iterations, while carefully optimizing the overhead of matching. Moreover, existing frameworks rely on the naive *recordStream* function [57] for cross-stream memory reuse, but its frequent host–device event synchronization can make the dispatch cost exceed the actual operator execution time, leading to severe host-bound issues. To eliminate this bottleneck, we leverage information from the global swap simulator to implement a custom *recordStream* mechanism that replaces host–device synchronization with efficient intra-device stream coordination, thereby avoiding host stalls and further improving overall performance.

We implemented SmartSwap with over 8,700 lines of code, which has been deployed in production for a year and will be open-sourced soon. Experiments on Ascend 910B [14] demonstrate that SmartSwap can adapt to varying operator sequences without causing training errors. In scalability experiments, SmartSwap achieves near-linear scalability with negligible performance degradation. When scaling up the batch size, sequence length, and hidden size of training models, SmartSwap can accommodate models exceeding hardware memory capacity by up to 4×, 4×, and 1.24× respectively. Moreover, SmartSwap can be leveraged to reduce the degree of parallelism or serve as an alternative to recomputation, achieving up to 38.94% performance improvement.

2 Background

In Graph Mode frameworks, the operator sequence remains fixed, but in Eager Mode frameworks, the integration of large models and dynamic training techniques leads to varying operator sequences, which contradicts the assumptions of existing swap techniques. This variation arises from several factors.

In Eager Mode, each operator is dispatched individually from the host, and dynamic training techniques can alter the sequence. For instance, the computation graph adapts to the model state, executing different operations under varying conditions via conditional branches [56], which changes the sequence. Mixed-precision training [31], which adjusts the loss scale for convergence, can shorten the operator sequence if an optimizer update is skipped. Similarly, on-the-fly validation can extend the sequence as it initiates validation at specific stages. Parallel training techniques such as elastic training [15, 24] or parallelism hot switching [13] also contribute to sequence changes. In practice, operator sequence changes are often observed, especially due to adjustments in loss scale.

When the operator sequence changes, previous swap policies become invalid, leading to issues such as: (i) undersized tensor swaps, failing to prevent Out-of-Memory (OOM) errors; (ii) misaligned lifetimes, where swap timings do not match actual tensor lifetimes,

causing suboptimal memory usage; (iii) runtime errors if tensors are not swapped back before their next use, resulting in crashes. This unpredictability not only reduces memory optimization efficiency but also threatens the reliability of the training system.

3 Lightweight Online Profiler

As mentioned in §1, existing profilers, such as the PyTorch built-in profiler, incur significant overhead, causing substantial performance degradation when used for continuous profiling. They also lack online support, requiring training to pause and hardcode profiling configurations, which is impractical for real-world training. Without continuous online profiling, it is impossible to track and adapt to changes in operator sequences. To address this, we develop a lightweight online profiler that inserts hooks at the operator dispatch point [3], operating in either Detailed or Lightweight mode based on whether a new swap policy is needed.

Lightweight Mode: When no new policy is needed, the profiler only collects operators from the current iteration. Inspired by tokenization [33], we assign an integer to each operator and represent the sequence as an integer tensor. By comparing tensors across iterations, we efficiently detect sequence changes with minimal overhead. The profiler adapts to sequence changes through a stage-adjusting module (Algo. 1). The Executor's multi-feature fuzzy matching in §5 allows SmartSwap to handle minor variations, switching to the WarmUp stage only if the sequence length changes by more than 5% or the cosine similarity drops below 95%. Fig. 2 shows the stages: WarmUp does nothing, GenPolicy generates and executes policies, and Stable reuses existing policies.

Algorithm 1 Algorithm of Stage Adjusting.

Input: *OpSeq*: Operator sequence represented as an integer tensor; *m, n*: Iterations with stable operator sequence before transferring to GenPolicy stage/Stable stage
Output: Stage

```
static StableStep ← 0                          ▷ All static variables are
static PrevOpSeq ← OpSeq                       ▷ initialized only once
static PrevStage ← WarmUp                      ▷ at the very beginning.
if diff(len(OpSeq), len(PrevOpSeq)) < 5% and
  CosineSimilarity(OpSeq, PrevOpSeq) > 95% then
    StableStep ← StableStep + 1
    if PrevStage is WarmUp & StableStep > m then
        Stage, StableStep ← GenPolicy, 0
    else if PrevStage is GenPolicy & StableStep > n then
        Stage ← Stable
else
    Stage, StableStep ← WarmUp, 0
PrevStage, PrevOpSeq ← Stage, OpSeq
```

Detailed Mode: During the GenPolicy stage, SmartSwap switches to Detailed mode, collecting essential data for policy generation with low overhead, including operator names, input and output tensor arrays, and iteration durations. For each tensor, we gather its pointer (data_ptr), type, usage count, and call stack, which are used for subsequent identification.

In PyTorch, the host dispatches operators asynchronously, with the host and device progressing at different paces. Collecting execution times requires heavy profiling tools like NVIDIA CUPTI [55] or Huawei AscendCL [4], which generate large amounts of performance data on the device and incur high costs in data transfer and computation. To avoid this overhead, we omit execution time collection and generate swap policies based on operator sequences and iteration durations, as described in §4.

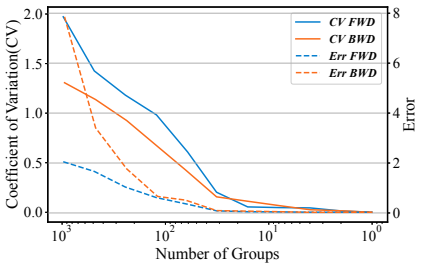


Figure 3: The relationship between the number of groups and (1) the CV of total execution time per group, and (2) the error of using the time calculated by Eq.(1) for each group.

In addition to operator and tensor data, the profiler tracks memory usage during operator execution. When a swap occurs, the profiler logs swap details, including location, tensor size, and other relevant information. This data allows us to reconstruct memory usage without swaps and use it in policy generation.

4 Policy Generator

The swap policy determines both memory savings and performance overhead. This section describes how our policy generator uses profiling information to construct an efficient swap policy.

4.1 Observation and Assumption

As explained in §3, our profiler does not collect per-operator execution times. Instead, we leverage the observation that dividing the operator sequence into evenly sized groups significantly reduces execution-time variance across groups, enabling effective policy generation without requiring exact operator timings.

We trained a 32-layer Llama2 model and profiled operator execution using the PyTorch profiler, analyzing forward and backward propagation separately. For forward propagation, we evenly grouped operators by execution order and measured variation in each group's total execution time using the coefficient of variation (CV). As shown by the blue solid line in Fig. 3, CV decreases as groups grow, indicating reduced variation in group execution time. Once the group count drops to 32 or fewer, CV approaches zero because each group includes at least one full transformer layer, whose computations are structurally identical. Backward propagation shows a similar pattern. This demonstrates that evenly grouping operators significantly reduces execution-time variation, and—since most modern LLMs are built by stacking similar blocks—this property is expected to generalize. Our policy generator leverages this by grouping operators in forward and backward phases and estimating each group's execution time with Eq. (1).

$$\overline{T_{group}} = \frac{T_{iter}}{N_{iter}} \times N_{group}. \tag{1}$$

In this equation, $\overline{T_{group}}$ is the estimated group execution time, T_{iter} is the iteration time, N_{iter} is the number of operators per iteration, and N_{group} is the number of operators per group. The dashed line

in Fig. 3 shows the estimation error, which remains low as long as the group count does not exceed the model's layer count. This confirms the reliability of our grouping-based estimation, and we refer to these groups as *logical layers*.

4.2 Memory Reduction List (MRL)

To generate effective swap policies, SmartSwap requires a clear optimization objective: reducing peak memory usage to stay within hardware limits and prevent OOM. Using data collected by the profiler, we reconstruct the memory usage without swaps and build a *memory reduction list (MRL)* by identifying stages where memory exceeds the hardware limit. For each operator in these stages, we create a *memory reduction entry (MRE)* specifying how much memory must be reduced at that point. Since training follows a repeated execution pattern, the allocation–usage–deallocation order remains consistent as long as the operator sequence is unchanged, allowing the MRL to be reused. When the sequence changes and a new policy is needed, we rebuild the MRL using newly collected memory traces, as outlined in Algo. 2.

4.3 Candidate List (CL)

During training, although any tensor can be swapped in principle, many swaps provide no memory benefit. For example, tensors used only in the forward phase have lifespans that never overlap with peak memory usage; swapping them wastes PCIe bandwidth and sacrifices opportunities to swap tensors that actually reduce peak memory. Tensors that are too small also lead to poor bandwidth utilization and limited benefit. Thus, we exclude tensors whose lifespans do not overlap with peak memory periods and construct a *candidate list (CL)* from the rest. For each candidate, we compute a score using Eq. (2):

$$Score = \hat{N_{MRE}} + C \times \hat{S}. \tag{2}$$

Here, $\hat{N_{MRE}}$ is the normalized number of MREs between the tensor's last forward use and first backward use, $\hat{S}$ is its normalized size, and C controls their relative importance. Intuitively, larger tensors and those that cover more MRE yield greater potential memory reduction and should be prioritized during policy generation.

4.4 Simulator

We introduce a simulator to accomplish two tasks: determining the time to pre-trigger swap-in and calculating the time at which swap-out is completed. As described in §4.1, we evenly divided the forward operator sequence into logical layers, as well as the backward operator sequence. Within the simulator, each logical layer is represented by a data structure that records key attributes, including the starting operator ID, layer type (forward, backward, or optimizer), assigned swap candidates, and the remaining time available for swap operations within the layer.

4.4.1 Pre-trigger Swap-in. Determining the timing of each swap operation is crucial. We first focus on swap-in, which is more complex due to dependencies on the tensor and the inherent host-to-device transfer delay. To avoid execution stalls, swap-ins must be carefully pre-triggered, which we achieve using our simulator.

The simulator processes the CL in descending score order. For each candidate, the required swap-in time is

$$T_{swap} = S/B, \tag{3}$$

where S is the tensor size and B is the host-device bandwidth. Starting from the logical layer where the tensor is first used in the backward phase, the simulator searches backward for a layer with $T_{remaining} > T_{swap}$, indicating sufficient time for swap-in without performance degradation. If no suitable layer is found before the peak memory period, the next candidate is considered. If none meet the criteria, we schedule the highest-scoring candidate within its previous logical layer, accepting some latency rather than halting training due to OOM.

Once the swap-in timing is determined, the simulator updates the logical layer by subtracting T_{swap} from $T_{remaining}$, adds the tensor to the layer's candidate list, and adjusts the MRL by decrementing the tensor's size from all overlapping memory reduction entries. Swap-in simulation is complete after these updates.

4.4.2 Swap-out Completion Time. Since no computation depends on swap-out completion, swaps can be triggered immediately after a tensor's last forward use. The key is determining when swap-out finishes, which is needed in §5.2. The simulator processes each candidate in swap-out order. For each tensor, T_{swap} is computed using Eq. (3). Starting from the logical layer of its last forward use, the simulator searches forward for a layer with $T_{remaining} > T_{swap}$ and records the swap-out completion there. The simulator then updates $T_{remaining}$ and the layer's candidate list as in §4.4.1.

4.5 Complete Process

The complete policy generation workflow is summarized in Algo. 2. At a high level, our method repeatedly builds an MRL from profiling data, constructs a CL that identifies tensors capable of reducing peak memory, and invokes the simulator to decide viable swap timings. This iterative process continues until all memory reduction requirements are resolved. Finally, swap-out completion times and proactive free events are computed, forming the final swap policy used in subsequent iterations.

Algorithm 2 Algorithm of Policy Generation.

Input: ProfData
Output: Policy

```
MRL ← ConstructMemoryReductionList(ProfData)
while MRL.isNotEmpty() do
    CL ← ConstructCandidateList(ProfData, MRL)
    if CL.isNotEmpty() then
        PolicyItems ← Simulator.simulate(CL, MRL)
        Policy.extend(PolicyItems)
    else
        Raise Error
Simulator.SetFreeTime(Policy)
```

5 Executor

Once the swap policy is generated, the next step is to dispatch the swap operations at the designated positions in the subsequent iteration to realize memory reduction. To ensure precision and efficiency, we developed an *Executor*.

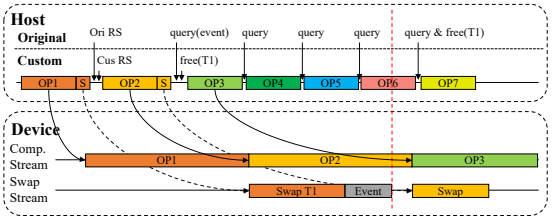


Figure 4: Comparison of original and custom RecordStream.

5.1 Identifying Tensors for Swap

The policy generator outputs a swap policy specifying which tensors should be swapped in each iteration. Executing this policy requires accurately identifying these tensors at runtime. Although two tensors may be equivalent at the abstract computation-graph level—sharing the same usage patterns and lifecycles—they occupy different physical addresses across iterations and lack persistent identifiers. Thus, Eager Mode frameworks treat them as distinct objects. The same difficulty applies to operators.

While the adaptive stage-switching mechanism in §3 handles major operator-sequence changes by regenerating policies, we avoid frequent regeneration. The Executor performs *multi-feature fuzzy matching* to locate target tensors, which can adapt to minor sequence variations. It matches tensors using features such as operator name, call stack, and data type, providing robustness to small sequencing shifts. However, naïvely comparing every runtime tensor against every tensor in the swap policy would incur substantial host-side overhead, slowing operator dispatch and causing host-bound performance degradation. To avoid this, we implement several optimizations that allow our fuzzy matching to rely solely on integer comparisons, eliminating expensive operations such as string comparisons.

5.2 Multi-stream Memory Reuse

In Eager Mode frameworks, operators on the same stream execute serially, so placing swap operations directly on the compute stream delays computation. A common workaround is to issue swaps on a dedicated stream, but this creates cross-stream memory reuse hazards. PyTorch manages memory allocation and deallocation on the host side, with each stream having its own memory pool, and memory cannot be reused across streams directly. This design leverages sequential execution within a stream to improve allocation efficiency. PyTorch uses reference counting to automatically release tensor memory when its reference count reaches zero [6], ensuring precise memory freeing. As illustrated in Fig. 4, when T1 is being swapped out in the swap stream, its reference count in the compute stream becomes zero, and its device buffer is freed and may be immediately reallocated to OP2, which can overwrite stale data because the two streams run concurrently.

PyTorch's *recordStream* avoids this hazard by marking tensors as "in use" by another stream and releasing them only after that stream completes. This ensures correctness but has two drawbacks: it prolongs memory lifetimes—delaying reuse due to asynchronous host/device progress—and it triggers frequent event queries, adding host-side overhead and risking host-bound slowdowns.

To overcome these issues, we implement a custom *recordStream* guided by our Simulator. As described in §4.4.2, the Simulator identifies the compute operator active when a swap-out finishes, giving us precise safe-release points. In the example in Fig. 4, it determines that OP2 is running when T1's swap-out completes; thus we reclaim T1's memory immediately after dispatching OP2 and allow reuse as early as OP3 (instead of waiting until OP7 as with the original *recordStream*). This approach eliminates the need for host polling and avoids unnecessary delays in memory reuse. This design enables earlier memory reuse, shortens memory lifetimes, and removes host-bound overhead while maintaining correctness.

6 Evaluation

Implementation. We implement SmartSwap with over 8,700 lines of Python and C++ code on top of PyTorch-NPU. Our implementation has been successfully deployed in the production environment for the past year, and we are working towards open-sourcing it in the near future.

Experimental setup. Our experiments are conducted on a server equipped with four ARM-based HiSilicon Kunpeng 920 CPUs, 2 TB RAM, and eight Ascend 910B NPUs, each with 64 GB of HBM. We use Compute Architecture for Neural Networks (CANN) 8.0[1] and PyTorch version 2.1.0 for our experiments. For hyperparameters in Algo. 1, we empirically set m to 2 and n to 5. With n = 5, SmartSwap generates five different policies and selects the one with the best runtime performance as the long-term policy.

6.1 Overall Performance and Scalability

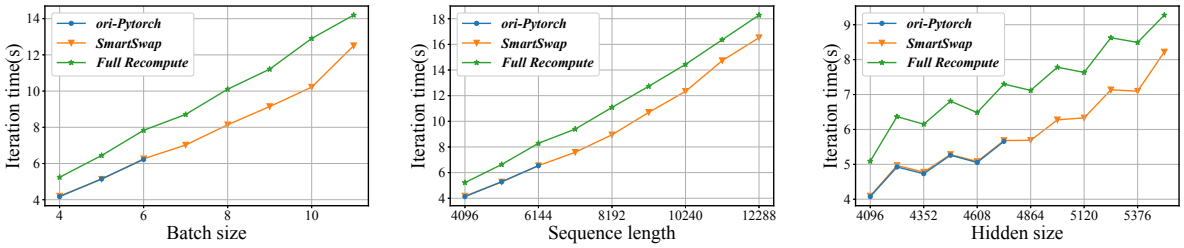
To evaluate SmartSwap's performance and scalability, we run experiments along the main model-scaling dimensions: batch size, sequence length, and hidden size, plus a layer-scaling study in §6.4. These dimensions represent standard ways of enlarging models in current practice. Using Llama2 as the target model, we assess SmartSwap 's practicality and the maximum model size achievable under each scaling direction.

We record the training performance as the model scales, shown in Fig. 5. Across all expansion dimensions, SmartSwap consistently meets our targets in §1. SmartSwap maintains near-linear scaling to 80/64 of PyTorch's maximum and beyond, as swap operations overlap with computation, effectively masking overhead. Compared to full recomputation, SmartSwap improves average performance by 16–19%, introducing negligible overhead when PyTorch can train without OOM. We further probe SmartSwap 's upper bound. Starting from batch size, layer count, sequence length, and hidden size of 4, 5, 4096, and 4096, we fix three variables and increase the fourth until OOM, recording the last feasible value. Compared to native PyTorch, SmartSwap supports models up to 4×, 1.83×, 4×, and 1.24× larger along the respective dimensions. Under the same hardware budget, SmartSwap not only trains larger models but also reduces the number of NPUs needed for a given model size, shifting reliance from high-communication TP/PP to DP, improving compute ratio and overall utilization. Across scenarios, SmartSwap provides up to 38.94% performance improvement.

6.2 Profiling Overhead

To evaluate the lightweight profiler, we compare its overhead against the PyTorch profiler on the same Llama2 training task. This task fits on a single NPU without OOM, so SmartSwap produces no policy;

[1] https://www.hiascend.com/en/software/cann



(a) Batch size scaling. (b) Sequence length scaling. (c) Hidden size scaling.

Figure 5: Performance under batch size, sequence length, and hidden size scaling.

Table 1: Comparison of profiling overheads.

	Time (ms)	extra Overhead
Baseline	4,911.1	/
Ours-Low Overhead Mode	4,952.6	0.9%
Ours-Detail Mode	6,612.0	34.6%
Built-in Profiler	15,699.7	219.7%

thus, all measured time consists solely of computation plus profiling overhead. Each configuration is run five times, and we report the average. Table 1 summarizes the results, where *Baseline* is the native PyTorch iteration time. Our lightweight profiler adds only 0.9% overhead in low-overhead mode—effectively negligible. Even in detailed mode it incurs just 34.6% overhead, an 84.25% reduction relative to the PyTorch profiler's 219.7%. These results show that SmartSwap 's online profiler remains lightweight in both modes and is suitable for continuous operator-sequence tracking.

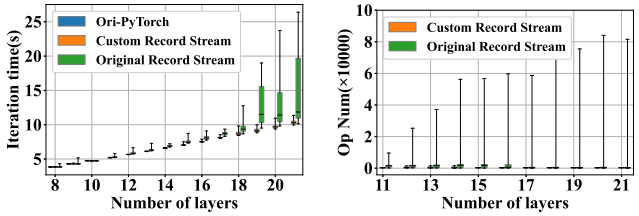
6.3 Long-term Stability Experiment

We evaluate the long-term stability of SmartSwap to ensure that swapping does not compromise training correctness. Using Llama2 scaled to 80 GB peak memory, we train for 5,000 steps on a single NPU with loss scaling and run online validation every 200 steps. When comparing the loss curves, the curve produced by SmartSwap fully overlaps with that of full recomputation, indicating that SmartSwap preserves training semantics.

We also include a reproduction of Capuchin [45]. Because its code is not available, we implement a PyTorch version following the paper and identify swap targets using an (operator ID, i-th tensor) tuple. Lacking any mechanism to tolerate sequence drift, Capuchin misidentifies tensors and causes the training program to crash in round 201, due to the new operator sequence introduced by on-the-fly validation. In contrast, SmartSwap 's multi-feature fuzzy matching absorbs minor sequence variations and automatically regenerates swap policies when larger changes occur, enabling stable long-running training without runtime errors.

6.4 Benefit from Custom RecordStream

To validate the benefits of our custom recordStream, we conduct a comparative experiment using Llama2, scaling the model by increasing layer count. Fig. 6(a) shows that with the custom recordStream, training time per step scales nearly linearly. In contrast, the original recordStream exhibits significant fluctuations as model size grows. Profiling reveals that frequent event queries increase host-side operator dispatch overhead, causing NPU idle time and host-bound scenario.



(a) Changes in training time as the model size increases. (b) Changes in memory block reuse interval as the model size increases.

Figure 6: Results of comparison experiment between the custom recordStream and the original recordStream.

To further investigate, we measure the number of operators dispatched between the recordStream call of a memory block and its eventual release back to the memory pool, which we call the memory block reuse interval. As shown in Fig. 6(b), the original recordStream yields reuse intervals two to three orders of magnitude longer at the tail and 3–4× larger on average than our custom design. These prolonged intervals force every operator dispatch within them to query event status, substantially increasing CPU overhead and pushing the system into a host-bound scenario, which is the source of the performance fluctuations shown in Fig. 6(a).

7 Conclusion

In this work, we present SmartSwap, a swap-based memory optimization framework redesigned end-to-end to handle the varying operator sequences of Eager Mode frameworks. SmartSwap features a lightweight online profiler for continuous monitoring, generates swap policies with limited operator information, and refines the policy application module for higher accuracy and performance. Experiments show that the profiler reduces overhead by 84.25%, enabling SmartSwap to track operator sequence changes in real time without introducing training errors. Scalability tests demonstrate near-linear speedup with minimal performance loss, allowing models up to 4× larger than device memory across multiple scaling dimensions. Moreover, SmartSwap can reduce parallelism requirements or replace recomputation, yielding up to 38.94% performance improvement.

Acknowledgments

This research is supported by the National Natural Science Foundation of China under Grant Numbers 62325205, 62502198 and U25B2035, and the Postgraduate Research & Practice Innovation Program of Jiangsu Province (No. KYCX25_0306).

References

[1] Meta AI. Accessed: 2024-12. papers with code trends. *https://paperswithcode.com/trends*.

[2] Ascend. Accessed: 2025-06. automatic migration. *https://www.hiascend.com/document/detail/zh/Pytorch/700/ptmoddevg/trainingmigrguide/PT_LMTMOG_0014.html*.

[3] Ascend. Accessed: 2025-06. opcommand.cpp. *https://gitee.com/ascend/pytorch/blob/master/torch_npu/csrc/framework/OpCommand.cpp*.

[4] Ascend. Accessed: 2025-08. ascendcl profiling api. *https://www.hiascend.com/document/detail/zh/canncommercial/82RC1/devaids/Profiling/atlasprofiling_16_0042.html*.

[5] Aaron Grattafiori *et al.* The llama 3 herd of models. *arXiv:2407.21783*, 2024.

[6] Adam Paszke *et al.* Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*, 2019.

[7] Chien-Chin Huang *et al.* Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *ASPLOS 20*, page 1341–1355, 2020.

[8] Deepak Narayanan *et al.* Pipedream: generalized pipeline parallelism for dnn training. In *SOSP*, page 1–15, 2019.

[9] DeepSeek-AI *et al.* Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025.

[10] DeepSeek-AI *et al.* Deepseek-v3 technical report. *arXiv:2412.19437*, 2025.

[11] Ding-Yong Hong *et al.* Gpu memory usage optimization for backward propagation in deep network training. *JPDC*, 199:105053, 2025.

[12] Frank Seide *et al.* Cntk: Microsoft's open-source deep-learning toolkit. In *KDD*, 2016.

[13] Hao Ge *et al.* Enabling parallelism hot switching for efficient training of large language models. In *SOSP*, page 178–194, 2024.

[14] Heng Liao *et al.* Ascend: a scalable and unified architecture for ubiquitous deep neural network computing : Industry track paper. In *HPCA*, 2021.

[15] Insu Jang *et al.* Oobleck: Resilient distributed training of large models using pipeline templates. In *SOSP*, page 382–395, 2023.

[16] Jared Kaplan *et al.* Scaling laws for neural language models. *arXiv:2001.08361*, 2020.

[17] Jason Ansel *et al.* Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *ASPLOS*, 2024.

[18] Jason Wei *et al.* Emergent abilities of large language models. *arXiv:2206.07682*, 2022.

[19] Jeff Rasley *et al.* Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *KDD*, page 3505–3506, 2020.

[20] Jiarui Fang *et al.* Parallel training of pre-trained models via chunk-based dynamic memory management. *IEEE TPDS*, 34(1):304–315, 2023.

[21] Jie Ren *et al.* Sentinel: Efficient tensor migration and allocation on heterogeneous memory systems for deep learning. In *HPCA 2021*, pages 598–611, 2021.

[22] Jie Ren *et al.* ZeRO-Offload: Democratizing Billion-Scale model training. In *USENIX ATC 21*, pages 551–564. USENIX Association, July 2021.

[23] Jingyang Yuan *et al.* Native sparse attention: Hardware-aligned and natively trainable sparse attention. *arXiv:2502.11089*, 2025.

[24] John Thorpe *et al.* Bamboo: Making preemptible instances resilient for affordable training of large DNNs. In *NSDI 23*, pages 497–513, April 2023.

[25] Kimi Team *et al.* Kimi k2: Open agentic intelligence. *arXiv:2507.20534*, 2025.

[26] Linnan Wang *et al.* Superneurons: dynamic gpu memory management for training deep neural networks. In *PPoPP 18*, page 41–53, 2018.

[27] Martín Abadi *et al.* TensorFlow: A system for Large-Scale machine learning. In *OSDI 16*, pages 265–283, Savannah, GA, November 2016. USENIX Association.

[28] Minsoo Rhu *et al.* vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *MICRO*, pages 18:1–18:13, 2016.

[29] Minsoo Rhu *et al.* Compressing DMA engine: Leveraging activation sparsity for training deep neural networks. In *HPCA 2018*, pages 78–91, 2018.

[30] OpenAI *et al.* Gpt-4 technical report. *arXiv:2303.08774*, 2024.

[31] Paulius Micikevicius *et al.* Mixed precision training. In *ICLR*. OpenReview.net, 2018.

[32] Ping Chen *et al.* CSWAP: A self-tuning compression framework for accelerating tensor swapping in gpus. In *CLUSTER*, pages 271–282. IEEE, 2021.

[33] Rico Sennrich *et al.* Neural machine translation of rare words with subword units. *arXiv:1508.07909*, 2016.

[34] Samyam Rajbhandari *et al.* Zero: Memory optimizations toward training trillion parameter models. In *SC 20*, pages 1–16, 2020.

[35] Samyam Rajbhandari *et al.* Zero-infinity: breaking the gpu memory wall for extreme scale deep learning. In *SC 21*, 2021.

[36] Seunghak Lee *et al.* On model parallelization and scheduling strategies for distributed machine learning. In *NeurIPS*, volume 27, 2014.

[37] Shen Li *et al.* Pytorch distributed: Experiences on accelerating data parallel training. *Proc. VLDB Endow.*, 13(12):3005–3018, 2020.

[38] Shriram S. B *et al.* Dynamic memory management for gpu-based training of deep neural networks. In *IPDPS*, pages 200–209. IEEE, 2019.

[39] Shuai Bai *et al.* Qwen2.5-vl technical report. *arXiv:2502.13923*, 2025.

[40] Song Han *et al.* Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding, 2016.

[41] Tailing Yuan *et al.* Accelerating the training of large language models using efficient activation rematerialization and optimal hybrid parallelism. In *USENIX ATC*, 2024.

[42] Tianqi Chen *et al.* Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv:1512.01274*, 2015.

[43] Tianqi Chen *et al.* Training deep nets with sublinear memory cost. *arXiv:1604.06174*, 2016.

[44] William Fedus *et al.* Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *JMLR*, 23(120):1–39, 2022.

[45] Xuan Peng *et al.* Capuchin: Tensor-based gpu memory management for deep learning. In *ASPLOS 20*, page 891–905, 2020.

[46] Yangqing Jia *et al.* Caffe: Convolutional architecture for fast feature embedding. In *MM*, page 675–678, 2014.

[47] Yanping Huang *et al.* Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *NeurIPS 2019*, volume 32. Curran Associates, Inc., 2019.

[48] Yuchao Li *et al.* Parameter-efficient sparsity for large language models fine-tuning. In *IJCAI*, pages 4223–4229, 2022.

[49] Zhongzhe Hu *et al.* Megtaichi: dynamic tensor-based memory management optimization for DNN training. In *ICS 22*, pages 25:1–25:13. ACM, 2022.

[50] Ziheng Jiang *et al.* MegaScale: Scaling large language model training to more than 10,000 GPUs. In *NSDI 24*, pages 745–760, Santa Clara, CA, 2024.

[51] Horace He*et al.* Transcending runtime-memory tradeoffs in checkpointing by being fusion aware. In *MLSys*, volume 5, pages 414–427. Curan, 2023.

[52] Chen Lei. *Deep Learning and Practice with MindSpore*. Cognitive Intelligence and Robotics. Springer, 2021.

[53] Zhuo Li*et al.* Model compression for deep neural networks: A survey. *Computers*, 12(3), 2023.

[54] NVIDIA. Accessed: 2025-06. nvidia nemo framework developer docs. *https://docs.nvidia.com/nemo-framework/user-guide/latest/nemotoolkit/features/optimizations/cpu_offloading.html*.

[55] NVIDIA. Accessed: 2025-08. nvidia cuda profiling tools interface (cupti) - cuda toolkit. *https://developer.nvidia.com/cupti*.

[56] PyTorch. Accessed: 2024-12. control flow - cond. *https://pytorch.org/docs/stable/cond.html*.

[57] PyTorch. Accessed: 2024-12. torch.tensor.record_stream. *https://pytorch.org/docs/stable/generated/torch.Tensor.record_stream.html*.

[58] Yanli Zhao*et al.* Pytorch fsdp: Experiences on scaling fully sharded data parallel. *Proc. VLDB Endow.*, 16(12):3848–3860, August 2023.