Chameleon: Taming Dynamic Operator Sequences for Memory-Intensive LLM Training

Zibo Wang
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China

Yuhang Zhou
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China

Zhibin Wang
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China

Shipeng Li
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China

Xinjing Huang
Huawei Technologies Co., Ltd
Shenzhen, China

Chendong Cai
Huawei Technologies Co., Ltd
Shenzhen, China

Bingxu Mu
Huawei Technologies Co., Ltd
Shenzhen, China

Yuqing Sun
Huawei Technologies Co., Ltd
Shenzhen, China

Zhiheng Hu
Huawei Technologies Co., Ltd
Shenzhen, China

Bin She
Huawei Technologies Co., Ltd
Shenzhen, China

Shu You
Huawei Technologies Co., Ltd
Shenzhen, China

Guanghuan Fang
Huawei Technologies Co., Ltd
Shenzhen, China

Rong Gu
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China

Wanchun Dou
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China

Guihai Chen
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China

Chen Tian
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China

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 during change.

We propose Chameleon, 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. Chameleon (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 Chameleon reduces profiling overhead by 84.25%, enables training models up to 4× larger than hardware memory while adapting to changes in operator sequences, improves performance by up to 38.94% compared to recomputation or high-degree parallelism.

1 Introduction

In recent years, deep neural networks (DNNs) have been increasingly applied across various domains, with LLMs [9, 14, 18, 40, 55] currently representing the most active area of research. To achieve higher accuracy and solve more complex problems, researchers have primarily focused on increasing the number of model parameters, which has been proven to be one of the most effective strategies [29, 58]. For example, the DeepSeek-R1 model series includes multiple models of varying sizes, with the largest model containing 671B parameters. In its evaluation, models with more parameters consistently demonstrate better performance [13].

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 [25, 30, 32, 37, 61], compression [20, 34], sparsity [16, 33, 59], recomputation [12, 21, 22], etc., which are mutually orthogonal and can be used in combination [8, 10, 23, 42, 51, 57, 60]. Among them, swap is an ideal memory optimization technique. It involves swapping memory blocks 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 [50]. While swap targets dynamic memory, mature frameworks such as DeepSpeed [47] focus on offloading static memory, on which our system is built.

Mainstream swap techniques [23, 24, 42, 48, 57] typically operate under the implicit assumption that the operator sequence remains unchanged during training. This is mainly because these methods were designed for the early popular Graph Mode frameworks, such as TensorFlow 1.x [1], MXNet [11] and others [4, 27, 31, 52], which follow a define-and-run paradigm, where the training program is compiled into a computation graph and repeatedly executed. The existence of computation graphs facilitates global optimizations like computation fusion and tensor swap. However, such a paradigm introduces significant complexities in model development and debugging. Consequently, users have turned to the more flexible Eager Mode framework exemplified by Pytorch [41]. As of December 2024, only 2% of open-access implementations of machine learning papers used TensorFlow, while 60% of implementations adopted PyTorch [3].

Motivating Example: Bridging the GPU–NPU Memory Gap for Seamless PyTorch Training. In real deployments, developers often build and train models in PyTorch on GPUs, benefiting from Eager Mode's flexibility for rapid iteration and debugging. To attract more users, Ascend also offers PyTorch support, enabling near drop-in migrations with minimal code changes [5]. However, migrating to NPUs (e.g., for cost or geopolitical considerations) can involve a downgrade in available memory—for instance, NVIDIA A100 provides 80 GB HBM, while the contemporaneous Ascend 910B has 64 GB. This 16 GB gap may force users to add more devices and adjust parallelization strategies, disrupting established workflows. We address this limitation with a swap-based memory optimization that bridges the capacity gap, targeting seamless PyTorch workload migration to NPUs without extensive model refactoring or performance loss. While the example focuses on GPU-to-NPU migration, the technique we propose can also applies to transitions between GPU generations with different memory capacities.

However, the inherent flexibility of Eager Mode frameworks renders existing swap techniques ineffective. These techniques typically follows a *profiling → policy generation → policy application* workflow. Based on the assumption of consistent operator sequences, existing swap techniques collect operator and tensor usage information during a single training iteration, generate a swap policy accordingly, and then directly apply the policy to subsequent iterations. But dynamic features like conditional branches [43], mixed-precision computation [36] and others [17, 26, 56] bring varying operator sequences across different training iterations, creating conflicts with the prior assumption. When the operator sequence changes, the policy generated for earlier sequences is invalidated, resulting in several issues. (i) Undersized tensor swapped: swapped tensors may be smaller than intended, failing to provide sufficient memory savings to prevent Out-of-Memory (OOM) issues; (ii) Misaligned lifetimes: a mismatch between the actual lifetimes of swapped tensors and their assigned swap-out/in timing can lead to suboptimal memory optimization; (iii) Runtime error: failure to promptly swap back tensors before their next use can result in training crashes. In general, this unpredictability not only decreases the efficiency of memory optimization but also threatens the reliability of the training system.

To enable effective swapping in the Eager Mode frameworks, we propose Chameleon. To the best of our knowledge, this is the first work that systematically handles varying operator sequences in swap-based memory optimization. Our analysis reveals that throughout the entire swap workflow, each phase introduces distinct and progressively evolving challenges, which we detail and address below.

First, *in the profiling phase, the profiling tools of existing works[23, 24, 42, 48, 57] cannot meet our requirements for lightweight, online analysis of varying operator sequences*. Concretely, these profilers exhibit two critical limitations: (i) prohibitive profiling overhead during continuous operation (up to 219% overhead) and (ii) lack of online profiling resulting in explicit training interruption. Therefore, we implement a lightweight online profiler to monitor varying operator sequences. It features two profiling modes and a stage-adjusting module. The two modes enable low-overhead and continuous detection of operator sequence variations, while the stage-adjusting module promptly adjusts the system state upon changes in operator sequences, triggering the generation of a new swap policy.

Second, *in the policy generation phase, we face the trade-off between profiling overhead and policy performance.* To achieve lightweight online profiling, we avoid collecting high-overhead information, such as each operator's execution time. However, the lack of this information introduces challenges in generating effective swap policies, as it is essential for determining the appropriate timing to pre-trigger swap-in operations. To generate efficient swap policies with low profiling overhead, we leverage the insight that evenly grouped operator sequences allow the average group execution time to approximate individual group durations with relatively low error. We devised a method to generate swap policies from this insight, without relying on detailed per-operator timings. During policy generation, we innovatively introduce a simulator to determine precise swap-out and swap-in timings from a global perspective, aiming to minimize performance degradation caused by swaps.

Third, *in the policy application phase, accurately and efficiently applying the generated policy to the next iteration is also challenging*, since Eager Mode frameworks lack unique identifiers to track operators or tensors across iterations. To accurately and efficiently apply the generated strategy to the training process, we use multi-feature fuzzy matching to overcome the missing unique IDs in Eager Mode frameworks and locate operators and tensors across iterations. The global simulator introduced in the policy generation phase enabled new performance optimization opportunities: by simulating the swap process, we could precisely determine the reuse timing of swapped memory blocks. We use this information to implement a custom *recordStream* function to further optimize Chameleon's performance.

We implemented Chameleon with more than 8,700 lines of code. Our implementation has been successfully deployed in the production environment for the past year and will be open-sourced soon. Experiments on Ascend 910B [35] demonstrate that Chameleon can adapt to varying operator sequences without causing training errors. In scalability experiments, Chameleon achieves near-linear scalability with negligible performance degradation. When scaling up the batch size, sequence length, and hidden size of training models, Chameleon can accommodate models exceeding hardware memory capacity by up to 4×, 4×, and 1.24× respectively. Moreover, Chameleon can be leveraged to reduce the degree of parallelism or serve as an alternative to recomputation, achieving up to 38.94% performance improvement.

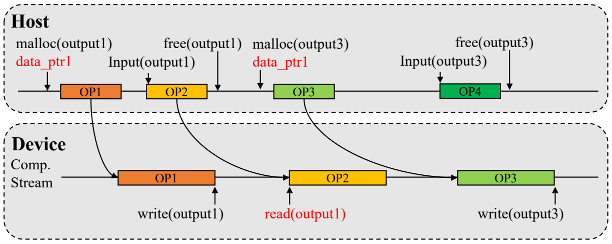


Figure 1. Illustration of PyTorch Memory Management.

2 Background and Challenges

2.1 Deep Learning Framework

Current popular deep learning frameworks can be broadly categorized into two major types.

Graph Mode is supported by TensorFlow 1.x [1], MXNet [11], Caffe [27], MindSpore [31], CNTK [52] and PyTorch 2.x [4]. In Graph Mode, model training computation is compiled as a computation graph, dispatched once, and reused throughout training. This consistency enables researchers to perform various global optimizations from a holistic perspective, thus swap on Graph Mode frameworks has already been extensively explored in existing works [24, 48, 57]. However, complex debugging and deployment have led to a decline in the popularity of Graph Mode frameworks [3].

Eager Mode, also known as dynamic graph mode, is the default execution mode in PyTorch [41] and is supported in TensorFlow 2.x [2]. Unlike graph mode, it does not construct a computation graph beforehand. Instead, each operator is dispatched individually to the device and executed sequentially but at different paces with respect to the host, as shown in Fig.1. This mode greatly simplifies debugging and deployment, appealing to researchers and developers prioritizing rapid prototyping. Consequently, Eager Mode is highly popular, contributing to PyTorch's widespread adoption: over 60% of open-sourced implementations from recent top-tier AI conference papers are based on PyTorch[3]. Given the fast evolution of AI algorithms, many companies adopt these implementations for full training rather than porting them to Graph Mode frameworks. Popular toolkits such as Megatron [54] and DeepSpeed [47] are also built on PyTorch. Therefore, Eager Mode is prevalent in both academia and industry, underscoring the need for swap-based memory optimization in Eager Mode frameworks like PyTorch.

As illustrated in Fig. 1, we use PyTorch as an example to describe the memory management mechanism of Eager Mode. Its memory allocation and deallocation are handled on the host side. Each stream has its own memory pool, and memory cannot be reused across streams directly. This design leverages sequential execution within a stream to improve allocation efficiency. Although host and device progress asynchronously, the device executes operators in the order dispatched by the host. This alignment eliminates the need for host-device synchronization or progress queries when allocating memory. PyTorch also uses reference counting to immediately release tensor memory when its reference count reaches zero [41], ensuring accurate memory freeing.

2.2 Existing Swap Techniques

Existing works [23, 24, 42, 48, 57] rely on the consistent operator sequence assumption of Graph Mode frameworks. Following the *Profiling* → *Policy Generation* → *Policy Application* workflow, they exhibit the following characteristics.

Single-iteration profiling. In Graph Mode frameworks, the fixed operator sequence yields consistent behavior across training iterations, allowing profiling from only a single iteration. Collected data include the operator sequence, the tensors used by each operator, and their execution times. From this information, we can determine the lifetime of each tensor, which is fundamental for swap-based memory optimization. Furthermore, operator execution times are another essential source to generate efficient swap policies.

One-Time policy generation. Policy generation is the key to swap-based memory optimization, as it directly determines the effectiveness of memory savings. It involves two key decisions: identifying tensors suitable for being swapped and selecting appropriate timing for each swap operation. Tensors with long lifetimes and extended idle periods between consecutive uses are ideal candidates for being swapped. To avoid performance degradation caused by computation stalling while waiting for swapped-out tensors, the swap-in operations must be pre-triggered such that tensors are ready before they are needed. For the Graph Mode, one-time policy generation is sufficient to guide swap-in and swap-out operations for all iterations.

Straightforward policy application. The final step in the workflow is applying the generated swap policy to subsequent training iterations for memory optimization. This process is particularly straightforward in Graph Mode frameworks because they provide unique tensor identifiers and consistent memory semantics across iterations, enabling safe reuse of swap policies without runtime adjustments, and guaranteed memory behavior consistency.

Overall, current swap techniques are implemented based on the assumption of consistent operator sequences in Graph Mode. This approach offers a simple workflow with low overhead and achieves significant memory optimization benefits.

2.3 Varying Operator Sequences in Eager Mode

Operator sequence remains fixed in Graph Mode framework. However, in Eager Mode frameworks, the integration of large-scale models and various training techniques leads to varying operator sequences, which contradicts the assumption of existing swap techniques. The reasons are as follows.

Since every operator is dispatched from the host without reuse in Eager Mode, various dynamic training techniques can impact the operator sequence. For example, the computation graph can dynamically adapt to the model state, such as executing different computations under varying conditions via conditional branches [43], which results in changes to the operator sequence. Another example is the widely-used mixed-precision training [36], which relies on loss scaling to ensure convergence. Adjustments to the loss scale—triggered by gradient overflows, underflows, or prolonged stability—can result in skipping an optimizer update, thereby shortening the operator sequence. Similarly, on-the-fly validation allows users to monitor the training progress of the model. This process initiates validation after the model reaches a certain training stage, resulting in an extended operator sequence. In addition to these, parallel training techniques involving model migration such as elastic training [26, 56] or parallelism hot switching [17] can also lead to changes in operator sequences. In practice, we almost always observe operator sequence changes during training, mainly caused by changes in the loss scale.

In Eager Mode frameworks, operator sequences may vary for multiple reasons, invalidating prior swap-based approaches that rely on the assumption of a fixed operator sequence.

2.4 Challenges

When applying swap techniques in practical model training, varying operator sequences in Eager Mode frameworks present three major challenges, as follows:

Tracking change in operator sequence incurs high overhead. Changes in the operator sequence can invalidate the generated policies, leading to suboptimal memory optimization or even runtime errors. Continuous profiling is therefore required to detect such changes and generate updated swap policies. However, current profilers (e.g. PyTorch profiler) impose high overheads for profiling. For example, enabling the PyTorch profiler during Llama2 training increases the iteration time from 4.9s to 15.7s, a 219% slowdown. Moreover, they lack online profiling support, requiring users to stop training and hardcode the profiling boundaries to initiate profiling. These limitations make continuous profiling impractical, underscoring the need for new profilng tools.

Limited profiling information will introduce new challenges to swap policy generation. To enable continuous profiling, a practical approach is to develop a new profiler that can run online and collect only essential information to minimize overhead. This means that high-cost information, such as the execution time of each operator, should be excluded. However, as discussed in §2.2, generating precise swap policies requires these execution times to determine optimal trigger moments for tensor swap-out/in. This fundamental contradiction creates a trade-off. Consequently, how to generate effective swap policies using limited profiling information is a new challenge.

Accurately and efficiently applying the generated policy to the next iteration is also challenging. In Eager Mode frameworks, operators are recompiled and dispatched in each iteration, and there are no unique identifiers to locate the same operators or tensors across multiple iterations. Therefore, correctly applying the policy generated in one iteration to the following iteration is not straightforward. Besides, the naive *recordStream* function [44] is often used to ensure the correctness during cross-stream memory reuse. However, its design involves frequent event queries between host and device, which can significantly increase host-side operator dispatch time—sometimes even exceeding the operator's execution time on the device. As a result, the device may idle while waiting for the host to dispatch the next operator, leading to a host-bound issue and performance degradation. Moreover, for host and device running at different paces, this mechanism can further slow down memory reuse. To apply swap efficiently and avoid host-bound issues, this mechanism must be carefully redesigned.

To enable the stable use of swap for memory optimization during training, it is essential to address these challenges, which form the foundation of Chameleon's design.

3 Overview

As in Fig. 2, Chameleon also follows the *Profiling* → *Policy generation* → *Policy application* workflow, comprising three modules—Lightweight Online Profiler, Policy Generator, and Executor—that address the challenges outlined in §2.4.

Lightweight Online Profiler (§4). To handle changes in operator sequences, the profiler integrates a stage-adjusting module that divides the training process into three stages: *WarmUp*, *GenPolicy*, and *Stable*, and supports two modes—

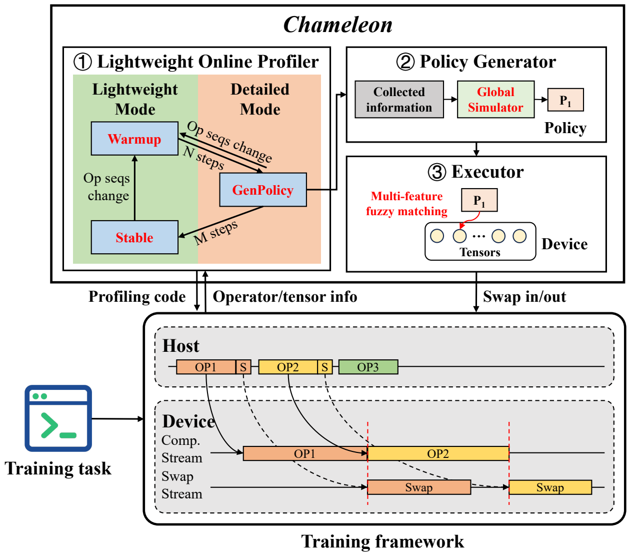


Figure 2. Overview of Chameleon and its workflow.

Lightweight, which only monitors operator sequence, and *Detailed*, which collects comprehensive operator and tensor information for policy generation. In Eager Mode frameworks, host and device operate at different paces, so collecting the execution time for individual operators requires heavyweight hardware profiling. To reduce this overhead, we avoid capturing this information in Detailed mode. Training starts in the WarmUp stage or re-enters it upon sequence changes, preparing for policy generation. If the sequence remains stable for N steps, the profiler enters the GenPolicy stage, switching to Detailed mode. After M steps, it transitions to the Stable stage and returns to Lightweight mode.

Policy Generator (§5). The policy generator uses the information provided by the profiler to generate the swap policy. In our experiments, we observed that dividing the operator sequence into evenly sized groups reduces the degree of variation in the total execution time for each group. *This allows the average execution time to serve as a proxy for each group's execution time with relatively low error.* Thanks to this insight, we can determine appropriate swap-in and swap-out timings without incurring the high cost of collecting individual operator execution times. In detail, we partition the operator sequence into multiple logical layers and arrange preemptive swap-in at the granularity of logical layers. To accurately estimate the completion time of multiple interfering swap operations, we innovatively introduce a simulator that evaluates swap overhead from a global perspective. This helps us (i) determine the precise timing of swap-out completion to enable timely memory release and reuse, and (ii) select appropriate pre-trigger points for swap-in, minimizing performance degradation caused by swap.

Executor (§6). The Executor is responsible for accurately and efficiently applying the generated policies. Specifically, it uses *multi-feature fuzzy matching* to locate operators and tensors across iterations, avoiding reliance on unique IDs. Given the high overhead of fuzzy matching, we carefully design the matching mechanism and employ various techniques to eliminate potential performance degradation. Additionally, the common practice of using a dedicated stream for swap operations can avoid blocking computations but introduces challenges with cross-stream memory reuse. To address the performance decline caused by the naive *recordStream* mechanism, we implement a custom *recordStream* function by leveraging information provided by the simulator during policy generation. Specifically, our design replaces the original host-device synchronization with intra-device synchronization between streams, thereby eliminating host-bound issues and further optimizing the performance of Chameleon.

4 Lightweight Online Profiler

As mentioned in §2.4, the built-in profiler not only has a significant overhead, which means using it to continuously collect information during the model training process would lead to substantial performance degradation, but also lack support for online usage, which means that we need to stop the training process to hardcode the profiling configuration in the training program to start profiling. This is unacceptable for real-world training processes. However, without continuous online profiling, it would be impossible to catch and adapt to changes in operator sequences. Therefore, we build a lightweight online profiler by inserting information-gathering hooks at the operator dispatching point.[1] Depending on whether a new swap policy needs to be generated, it operates in either Detailed or Lightweight mode.

Lightweight Mode: When no new policy generation is needed, the profiler runs in Lightweight mode, collecting only the operators involved in a training iteration. Inspired by tokenization [53] in language models, we assign an integer value to each operator based on its name and represent the operator sequence as an integer tensor. By comparing the tensors recorded in consecutive iterations, we can efficiently determine whether the operator sequence has changed with minimal storage and computation overhead.

To adapt to operator sequence changes, we integrate a stage-adjusting module into the profiler, which adjusts the stage according to Algo. 1. The multi-feature fuzzy matching in the Executor in §6 enables Chameleon to tolerate minor variations in the operator sequence and only switches to the WarmUp stage when the operator sequence length changes by more than 5% or the cosine similarity between the two tensors falls below 95%. The actions performed in each stage are

[1] In PyTorch-NPU, this occurs in `OpCommand.cpp` [6]. Owing to the dynamic operator dispatch characteristic of Eager Mode frameworks, similar locations can be easily identified in other Eager Mode frameworks as well.

depicted in Fig. 2: the WarmUp stage does not perform any operations, the GenPolicy stage involves policy generation and execution, while the Stable stage reuses the previously generated policy without generating new ones.

Algorithm 1 Algorithm of Stage Adjusting.

```
Input: OpSeq: Operator sequence represented as an integer
    tensor; m, n: Iterations with stable operator sequence be-
    fore 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
      end if
  else
      Stage, StableStep ← WarmUp, 0
  end if
  PrevStage, PrevOpSeq ← Stage, OpSeq
```

Detailed Mode: During the GenPolicy stage, Chameleon generates a new policy based on collected data. In this case, the lightweight online profiler switches to a Detailed mode, collecting only a small set of information necessary for the policy generation process with relatively low overhead, including the name of each operator, the input tensor arrays, and output tensor arrays for each operator and the duration of each training iteration. For each tensor, we collect its pointer (data_ptr), data type, usage count, and call stack, which are used as features for subsequent identification.

As in §2.1, the host dispatches operators to the device asynchronously in PyTorch, and the two sides progress at different paces. Consequently, collecting operator execution times requires heavyweight hardware profiling tools such as NVIDIA CUPTI [39] or Huawei AscendCL Profiling API [7]. These tools monitor the hardware and generate massive amounts of performance data for the computation tasks on the device. Ultimately, this data must be transferred to the host at high cost and undergo intensive computation to align with operators dispatched on the host side. The entire process—starting the heavy profiling, monitoring the hardware, transferring data, and performing correlation—introduces substantial overhead, which explains the high cost of the PyTorch profiler built on top of them. Therefore, we avoid collecting the execution time of each operator. In §5, we will explain how we generate swap policy based solely on the operator sequence and the duration of each training iteration, without relying on operator execution times.

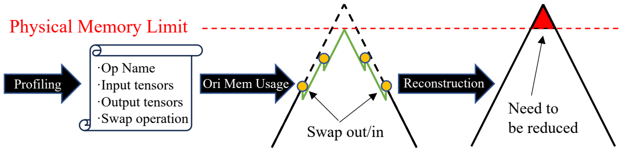


Figure 3. Reconstruction of the actual memory usage.

In addition to collecting operator- and tensor-related data, the profiler must also capture the amount of memory in use during each operator's execution. Furthermore, when a swap operation occurs (whether it's a passive swap triggered by OOM or an active swap guided by the policy), the profiler needs to log the swap location, tensor size, and other relevant details. With this information, we can reconstruct the actual memory usage of the training process without swaps and use it in policy generation, as depicted in Fig. 3.

5 Policy Generator

The swap policy is key to determining memory benefits and performance overhead. In this section, we will introduce how our policy generator leverages information from the profiler to generate an efficient swap policy.

5.1 Observation and Assumption

As explained in §4, our profiler does not collect the execution time of each operator. It is because we find the insight that dividing the operator sequence into evenly sized groups reduces the degree of variation in the total execution time for each group and this enables us to generate a swap policy without relying on the exact execution time of operators.

We trained a 32-layer Llama2 model and profiled its operator execution using the PyTorch profiler, separately analyzing forward and backward propagation. Taking forward propagation as an example, we grouped operators evenly in execution order and measured the variation in total execution time of each group using the coefficient of variation (CV). The blue solid line in Fig. 4 illustrates how CV changes with the number of groups: as the group count decreases, each group contains more operators, and CV drops—indicating reduced variation in group execution time. Once the group count falls to 32 or fewer, CV converges to zero. This is expected, since Llama2 is composed of repeated transformer layers. When each group includes at least one full transformer layer, the computations within groups remain consistent, minimizing execution time variation. A similar pattern is observed in backward propagation. This analysis indicates that by dividing the operator sequence into evenly sized groups, we can reduce the degree of variation in the total execution time of each group. Since most modern LLMs are

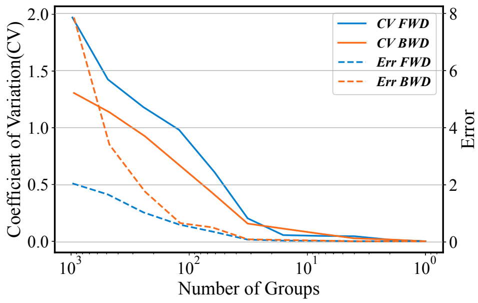


Figure 4. The relationship between the number of groups and (1) the CV of total execution time per group, and (2) the error of using the time calculated by Eq.(1) for each group.

built by stacking similar structures, we believe this observation generalizes to other LLMs. This assumption forms the foundation of our policy generator. Based on this insight, we evenly group operators in the forward phase according to execution order, as well as the backward phase. Then we estimate each group's execution time based on Eq.(1).

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

In this equation, $\overline{T_{group}}$ denotes the estimated execution time of each group, T_{iter} is the measured time for one training iteration, N_{iter} is the number of operators in one iteration, and N_{group} is the number of operators in each group. Each estimated execution time is compared against the actual execution time, and the error rate is shown by the dashed line in Fig. 4. Notably, the error remains minimal when the number of groups does not exceed the model's layer count, demonstrating the high applicability and reliability of our grouping-based estimation method. In subsequent discussions, we refer to these operator groups as *logical layers*.

5.2 Memory Reduction List (MRL)

To generate effective swap policies, a clear optimization objective is required. In Chameleon, the goal is to reduce peak memory usage during training to remain within hardware limits, thereby preventing OOM.

To achieve this, we use the profiler to collect actual memory usage data without swaps and construct a *memory reduction list(MRL)*. Specifically, we identify stages where memory usage exceeds the hardware limit, shown as the red triangular sections in Fig.3. For each operator in this stage, we create a *memory reduction entry(MRE)* indicating the memory reduction required at that operator's execution point to avoid OOM. Since model training repeats the same execution pattern, the allocation, usage, and deallocation order of tensors remain consistent as long as the operator sequence is unchanged, allowing us to reuse the MRL. Once the sequence changes and a new swap policy is required, we rebuild the MRL using newly collected memory usage data, as illustrated in Algo. 2. The detailed MRL data structure is as below:

```
memory_reduction_list: {
    op[i]: 1G,
    op[i+1]: 1.1G,
    ...
    op[j-1]: 1.1G,
    op[j]: 1G
}
```

5.3 Candidate List (CL)

During training, while every tensor can technically be swapped, not all swaps contribute to memory savings. For instance, if a tensor doesn't need to be saved until the backward phase, its lifespan won't overlap with peak memory usage periods. Swapping such a tensor would not only fail to reduce peak memory usage but would also consume valuable PCIe bandwidth between host and device, taking up opportunities for swapping other tensors that could reduce peak memory. Also, its size needs to be large enough. If the tensor size is too small, it may lead to underutilization of the PCIe bandwidth, wasting potential optimization opportunities.

Therefore, we exclude tensors whose lifespans do not overlap with peak memory usage periods and construct a *candidate list(CL)* from the remaining tensors. For each selected candidate tensor, we compute a score base on Eq.(2).

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

In Eq. (2), $N_{\hat{MRE}}$ represents the normalized number of MREs between the tensor's last forward use and first backward use. Similarly, $\hat{S}$ denotes the normalized tensor size, and C is a coefficient that adjusts the relative importance of these two factors. Intuitively, the larger the candidate tensor size is, and the more MREs the candidate tensor's lifespan covers, the greater benefit swapping the tensor would bring, and thus the tensor should be prioritized more in the policy generation phase for being selected.

5.4 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 §5.1, 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. The data structure of the logical layers in the simulator is shown in the following code snippet.

```
data_struct: {
        start_op_id : int,
        logical_layer_type : {FWD,BWD,OPT},
        candidates : [],
        remaining_time : float,
    }.
```

5.4.1 Pre-trigger Swap-in. For each swap operation, it is crucial to carefully determine the timing. We first discuss swap-in, as determining its start time is more complex due to the dependency of operators that require the swapped-in tensor on its completion, as well as the inherent delay in transferring data from the host to the device. To prevent performance degradation caused by swap-in operations blocking execution, we must carefully pre-trigger them. We use our simulator to achieve this goal from a global perspective.

The simulator takes the CL as input, processing each candidate in descending order based on their score. For a given candidate, the required swap-in time can be calculated as

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

where S is the size of the candidate and B is the bandwidth between host and device. Our strategy is to start from the previous logical layer where the tensor is first used in the backward phase and sequentially search backward for a logical layer with $T_{remaining} > T_{swap}$, which means that the logical layer can accommodate the swap-in without causing performance degradation. If such a layer is found, the swap-in is scheduled within that layer to avoid performance degradation. If no such logical layer is found until the peak memory usage time, we move on to the next candidate. If, after evaluating all the candidates, none meet the criteria, it indicates that no candidate's swap-in can be completed within the remaining time of any layer. Instead of permitting OOM to halt operations, we can still opt to perform a swap-in. Although this will introduce latency into the computation of the relevant operator, leading to NPU idle periods and a reduction in performance, we deem it preferable to an outright termination of training. In this case, we prioritize the candidate with the highest score, scheduling its swap-in within the previous logical layer where the tensor is used, thereby maximizing the memory reduction benefit.

Once a tensor's swap-in timing is determined, the simulator updates the corresponding logical layer by subtracting T_{swap} from its $T_{remaining}$ and adding the tensor to its candidate list. It then adjusts the MRL by decrementing the tensor's size from the memory reduction entries of all operations overlapping with its lifecycle. The swap-in timing simulation is complete only after these updates.

5.4.2 Swap-out Completion Time. Since no computations depend on the completion of swap-out, they can be triggered immediately after the tensors' last use in the forward phase and we don't need to determine the timing. What we need to do is to determine when the swap-out completes, as this information will be utilized in §6.2.

The simulator takes the generated policy as input and processes each candidate in the order in which they are swapped out. For each tensor, the required swap-out time is calculated using Eq. (3). Then we start from the logical layer where it is last used in the forward phase and sequentially search forward for a logical layer with $T_{remaining} > T_{swap}$, which means that the logical layer can accommodate the swap out, so the tensor's swap-out is marked as completed within that layer. Then we follow the same way in §5.4.1 to update the $T_{remaining}$ and CL of the logical layer.

5.5 Complete Process

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
    end if
end while
Simulator.SetFreeTime(Policy)
```

The complete policy generation process is outlined in Algo. 2. Each time a new swap policy is generated, a new MRL is constructed using the profiling data from the previous iteration. While the MRL remains non-empty, it iteratively selects tensors for swap. In each iteration, it first builds a CL following the approach in §5.3, identifying tensors that can effectively reduce peak memory usage. If such tensors are found, it proceeds with the simulation process detailed in §5.4.1. However, if no tensor meets the criteria, it indicates that further reducing peak memory is unattainable, preventing training from continuing without triggering an OOM error. In this case, an error is raised, and the process terminates. If the MRL is cleared, it indicates that the selected tensors suffice to ensure training proceeds without OOM, so the tensor selection process is complete. After this, we determine the timing for swap-out completion and proactive memory release for each selected tensor following §5.4.2, facilitating earlier reuse of the corresponding memory blocks.

After these steps, the swap policy is finalized and passed to the executor for use in subsequent training iterations.

6 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*.

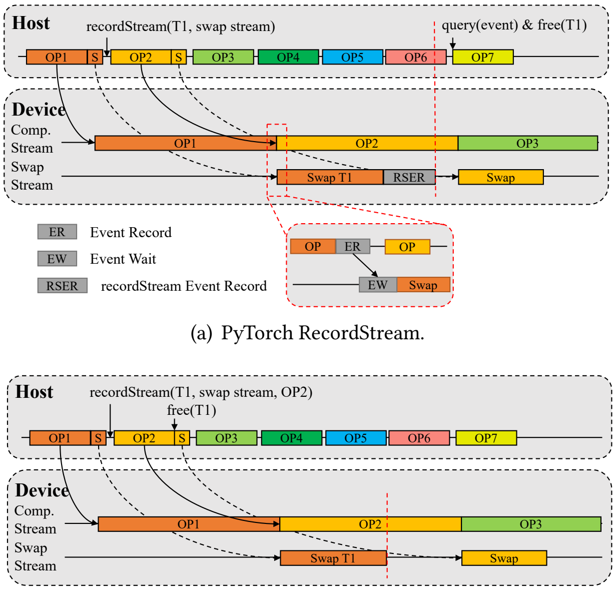
6.1 Identifying Tensors for Swap

The policy generation module produces a swap policy as input to the Executor, specifying which tensors to swap in each training iteration. Accurately identifying these tensors, however, is challenging. While it is conceptually straightforward at the level of an abstract computation graph to identify the "same" tensor across iterations—two tensors with identical usage patterns and lifecycles—in practice, these tensors reside at different physical addresses and lack unique identifiers to denote their correspondence. From the perspective of an Eager Mode framework, these tensors are treated as entirely unrelated. The same challenge applies to operators.

Although the adaptive stage-switching module described in §4 enables us to accommodate changes in the operator sequence, we aim to switch stages and regenerate policies only when significant changes occur. For minor changes, the Executor's matching mechanism is used instead. We adopt *multi-feature fuzzy matching* to identify target tensors, enabling robustness to slight sequence changes. Specifically, we locate a tensor by using features such as the operator's name, call stack and data type. However, since each tensor involved in the training process must be compared against all tensors specified in the swap policy, an unoptimized comparison process can incur substantial host-side overhead, drastically increasing the cost of operator dispatch and leading to host-bound performance degradation. To mitigate this, we implemented some tricks, with which our multi-feature fuzzy matching relies solely on integer comparisons, avoiding time-consuming operations such as string comparisons. Details are deferred to Appendix A for space considerations.

6.2 Multi-stream Memory Reuse

As described in §2.1, operators on the same stream execute serially in Eager Mode frameworks, so placing swap operations directly in the compute stream inevitably delays subsequent computations. A common workaround is to assign swap operations to a dedicated stream, but this introduces cross-stream memory reuse hazards. For instance, as shown in Fig. 5(a), after dispatching a swap-out for T1, the pointer to the physical memory of T1 is changed from a device pointer to a host pointer, and the reference count of the device memory block drops to 0, freeing it. This block may be immediately allocated to OP2 to store its output. Since tasks in different streams execute concurrently, OP2 could overwrite T1 with stale data, compromising training correctness.



(a) PyTorch RecordStream.

(b) Custom RecordStream.

Figure 5. Illustrations of multi-stream memory reusage.

As shown in Fig. 5(a), PyTorch's *recordStream* mitigates cross-stream hazards by marking tensors as “in use” by another stream, deferring release until the marked stream completes. It achieves this by issuing an event on the marked stream and querying its status whenever a new memory allocation request is made. The tensor is released only after the event is completed. While correct, this approach has two major drawbacks. First, it prolongs memory block lifetimes: memory that could be reused immediately after OP2's completion is delayed until OP7 is dispatched, due to the asynchronous progress of host and device. Second, swap operations generate numerous event queries, imposing substantial host-side overhead, potentially making the system host-bound and degrading training performance.

To address these issues, we design a custom *recordStream* informed by our Simulator. As in §5.4.2, the Simulator tells us which compute operator is running when a swap-out completes, allowing us to determine safe release points precisely. In the same scenario as in Fig. 5(a), with this information, we can determine that when the swap of T1 completes, operator OP2 is being executed on the compute stream. Consequently, as in Fig. 5(b), we can reclaim the corresponding memory block after dispatching OP2, allowing the memory block to be reused when dispatching OP3. Instead of host polling, we insert an event record/wait pair between the swap and compute streams to ensure precise reclamation and avoid memory conflicts. This method enables earlier memory reuse, shortens lifetimes, and eliminates host-bound performance bottlenecks without compromising correctness.

6.3 OOM Handling in the Warm-Up Stage

In §6.2, we shift synchronization from the swap stream–host path (different paces) to the swap stream–compute stream path (same pace), avoiding host-bound problem. We apply the same principle to OOM scenarios.

Swap is introduced to overcome memory limits and enable training of larger models. At the start of training, before an effective swap policy exists, the model inevitably exceeds memory capacity, triggering OOM. To keep training uninterrupted and profiling data intact, we designed an optimized OOM handling process: (i) Release memory blocks — Upon OOM, release all memory blocks marked by the custom RecordStream, including those in the middle of swapping or completed swaps awaiting release. (ii) Synchronize swap and compute without blocking — Insert an event record/wait mechanism between the swap stream and compute stream to ensure freed blocks are reused only after the swap finishes, avoiding host-device synchronization stalls and allowing computation to resume immediately. (iii) Defragment memory — Use the GMLake [19] memory pool to reduce fragmentation caused by frequent swaps, then retry the allocation. (iv) Passive swap on repeated OOM — If OOM persists, select a tensor whose size is closest to the required block and swap it out, again using inter-stream synchronization to avoid blocking.

This process applies to all Warm-Up stage iterations, including the first. In the initial iteration, no swap policy exists, so step (i) releases nothing; if OOM recurs, step (iv) frees sufficient memory for allocation. A detailed design of OOM handling in the Warm-Up stage is provided in Appendix B.

7 Evaluation

In the evaluation, we first present Chameleon 's overall performance and scalability across different model scales to verify its ability to meet the motivating objectives in §1. We then assess profiling overhead, adaptability to varying operator sequences, and the benefits of the custom *RecordStream*. Due to space constraints, some detailed results are provided in Appendix C for interested readers to consult.

7.1 Methodology

Implementation. We implement Chameleon 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[2] 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, Chameleon generates five different policies and selects the one with the best runtime performance as the long-term policy.

7.2 Overall Performance and Scalability

To evaluate Chameleon's performance and scalability, we conduct scalability experiments across multiple dimensions, including batch size, sequence length, and hidden size. We also evaluate in the layer dimension, which is presented in §7.5. These dimensions represent the primary approaches for expanding model size in contemporary research. Our tests aim to evaluate the practicality of Chameleon under these scenarios. Using Llama2 as the target model, we assess Chameleon 's practicality and explore the maximum model size achievable under each scaling scenario.

We record the training performance as the model scales, shown in Fig. 6. Across all three expansion dimensions and the layer expansion in Fig. 8(a), Chameleon consistently achieves our target in §1. For all expansion dimensions, Chameleon maintains linear performance scaling to 80/64 of the maximum value achievable by PyTorch, and can even scale linearly to larger models. Since swap operations can overlap with computation tasks, the associated overhead is effectively masked. Consequently, the performance of Chameleon is significantly better than that of full recomputation. When native PyTorch can train a model without triggering OOM, Chameleon introduces negligible additional overhead. In the three different model scaling directions, Chameleon achieves average performance improvements of 18.78%, 16.69%, and 19.32% compared to full recomputation.

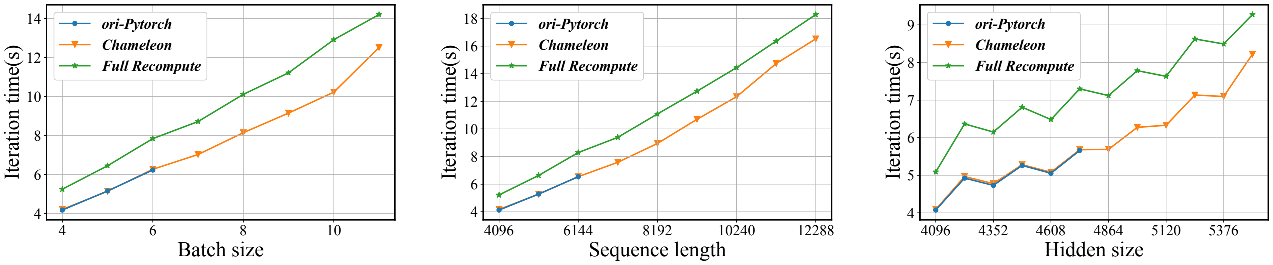
Besides, we further explore the upper bound of Chameleon's capability. We initialize the batch size, number of layers, sequence length, and hidden size to 4, 5, 4096, and 4096. Then we fix three of these variables and increase the fourth until an OOM occurrs, recording the maximum value before failure. The results show that compared to the original PyTorch implementation, Chameleon can accommodate models exceeding the hardware memory capacity by up to 4×, 1.83×, 4×, and 1.24× along the three dimensions, respectively.

Under identical hardware constraints, Chameleon can not only train larger models, but also use fewer NPUs for the same model size, reducing reliance on high-communication TP/PP in favor of DP, which improves computation ratio and hardware utilization. Experiments show that Chameleon can deliver up to 38.94% performance improvement across various scenarios. Detailed results demonstrating Chameleon's performance and scalability are provided in Appendix C.

7.3 Profiling Overhead

To verify the effectiveness of the lightweight profiler, we conduct information collection using both the PyTorch profiler and the lightweight profiler on the same Llama2 training task, comparing their time overhead. This training task can be executed on a single NPU without triggering an OOM

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



(a) Batch size scaling.

(b) Sequence length scaling.

(c) Hidden size scaling.

Figure 6. 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%

error. Therefore, Chameleon does not generate a policy for this task, meaning the recorded time includes only computation overhead and profiling overhead. For each experiment, we repeat the process five times and compute the average. The results are presented in Table 1. Here, *Baseline* refers to the time required to complete one training iteration using the native PyTorch framework.

The table shows that our lightweight profiler incurs only a 0.9% overhead in low-overhead mode, which is virtually negligible. Even in detailed mode, it introduces just 34.6% overhead, representing an 84.25% reduction compared to the 219.7% overhead of the built-in profiler. This demonstrates that our lightweight online profiler introduces minimal overhead in both lightweight and detailed modes, making it well-suited for continuous operator sequence monitoring.

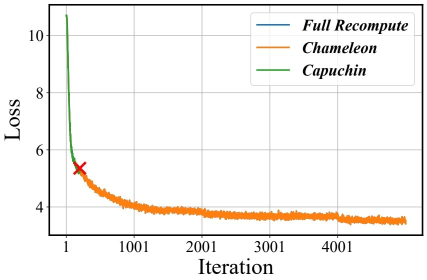
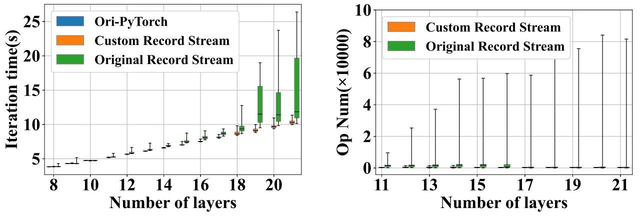


Figure 7. Long-term stability experiment result.

7.4 Long-term Stability Experiment

We conduct a long-term stability experiment to verify that Chameleon does not compromise the correctness of model training. Using the Llama2 model, scaled to approximately 80GB of memory usage during training, we train the model for 5,000 steps on a single NPU with loss scaling and perform on-the-fly validation every 200 steps. We record the loss throughout the process as shown in Fig. 7. The loss curve of the same model trained with full recomputation enabled is used as the baseline and also shown in Fig. 7. It can be observed that the loss curve of Chameleon completely overlaps with that of full recompute, indicating that the introduction of Chameleon does not affect the correctness of training.



(a) Changes in training time as the model size increases.

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

Figure 8. Results of comparison experiment between the custom recordStream and the original recordStream.

In this experiment, we also include the result from Capuchin [42]. Since Capuchin's code is not open-sourced, we implement a PyTorch version based on its paper's description, identifying selected tensors using a (operator ID, i-th tensor used by the operator) pair. The loss for this implementation is also shown in Fig. 7. It can be observed that since Capuchin does not adapt to changing operator sequences, it causes the training program to crash in round 201, due to the addition of an operator sequence introduced by an on-the-fly validation. Thanks to Chameleon's multi-feature fuzzy matching mechanism, minor changes in the operator sequence can be effectively tolerated. For more significant changes, Chameleon automatically triggers the regeneration of swap policies. These mechanisms enable Chameleon to adapt to variations in operator sequences and prevent runtime errors caused by such changes.

7.5 Benefit from Custom RecordStream

To validate the benefits of the custom recordStream implementation, we conduct a comparative experiment using the Llama2 model. We increase the number of model layers to scale its size and compare the performance of training under Chameleon with the custom recordStream and the original recordStream. The results are shown in Fig. 8(a). When using the custom recordStream, the training time per step scales nearly linearly as the model size increased. In contrast, with the original recordStream, training time exhibits significant fluctuations as the model size grows. Profiling analysis reveales that these fluctuations stem from frequent event queries, which increase the host-side overhead of operator dispatch. This slowdown in dispatch leads to idle time on the NPU, resulting in host-bound scenarios.

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. The results, shown in Fig. 8(b), indicate that with the original recordStream, the maximum memory block reuse interval is two to three orders of magnitude higher compared to the custom recordStream. On average, the memory block reuse interval is 3× to 4× higher with the original recordStream than with the custom implementation. This not only means that when using the original recordStream, the memory block reuse interval is significantly prolonged, but also that each operator dispatch within a memory block's reuse interval requires querying the status of the associated event. This greatly increases the CPU resource overhead for operator dispatch, leading to a host-bound scenario, which is the source of the performance fluctuations shown in Fig. 8(a).

8 Related Work

Parallelism has been widely used to scale model training [25, 30, 32, 37, 61]. While it enables larger models and faster training by leveraging multiple accelerators, it also raises hardware costs and incurs communication overhead that cannot be fully hidden [28], potentially degrading performance and utilization. Integrating swap with parallelization alleviates these issues, lowering costs while improving utilization.

Other approaches reduce memory requirements via recomputation [12], model compression [34], sparsity [33], or their combinations [8, 10, 23, 42, 51, 57, 60]. These methods effectively shrink the memory footprint but incur trade-offs: recomputation adds unavoidable overhead to the critical path, while compression and sparsity may compromise accuracy, potentially affecting final model quality.

A number of works have explored swap-based training to scale model size. The ZeRO series [45, 46, 49] and PatrickStar [15] primarily target static memory (parameters and optimizer states), while Chameleon focuses on dynamic memory (activations). These approaches are orthogonal and can be combined—we build our work upon DeepSpeed [47] and enable ZeRO-2 in our evaluation. NVIDIA NeMo [38] supports offloading but requires manual specification of targets, functioning as a low-level backend; Chameleon instead automates this using runtime behavior. vDNN [50] pioneered swap-based optimization but suffered from synchronization stalls, which later systems such as SuperNeurons [57], SwapAdvisor [24], and Sentinel [48] addressed with asynchronous prefetching. However, these were built for Graph Mode frameworks, limiting applicability to modern Eager execution. More recent efforts like Capuchin [42] and MegTaiChi [23] extended support to Eager Mode but assumed fixed operator sequences. To our knowledge, Chameleon is the first swap-based approach to explicitly accommodate dynamic operator sequences during training.

9 Conclusion

In this work, we present Chameleon, a swap-based memory optimization framework redesigned end-to-end to handle the varying operator sequences of Eager Mode frameworks. Chameleon 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 Chameleon 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, Chameleon can reduce parallelism requirements or replace recomputation, yielding up to 38.94% performance improvement.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for Large-Scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, November 2016. USENIX Association.

[2] Akshay Agrawal, Akshay Modi, Alexandre Passos, Allen Lavoie, Ashish Agarwal, Asim Shankar, Igor Ganichev, Josh Levenberg, Mingsheng Hong, Rajat Monga, and Shanqing Cai. Tensorflow eager: A multi-stage, python-embedded dsl for machine learning. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *Proceedings of Machine Learning and Systems*, volume 1, pages 178–189, 2019.

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

[4] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang,

Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 929–947, New York, NY, USA, 2024. Association for Computing Machinery.

[5] Ascend. Accessed: 2025-06. ascend extension for pytorch > pytorch training > model migration and tuning > model migration > model script migration > (recommended) automatic migration. *https://www.hiascend.com/document/detail/zh/Pytorch/700/ptmoddevg/trainingmigrguide/PT_LMTMOG_0014.html*.

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

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

[8] Shriram S. B, Anshuj Garg, and Purushottam Kulkarni. Dynamic memory management for gpu-based training of deep neural networks. In *2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2019, Rio de Janeiro, Brazil, May 20-24, 2019*, pages 200–209. IEEE, 2019.

[9] Shuai Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Sibo Song, Kai Dang, Peng Wang, Shijie Wang, Jun Tang, Humen Zhong, Yuanzhi Zhu, Mingkun Yang, Zhaohai Li, Jianqiang Wan, Pengfei Wang, Wei Ding, Zheren Fu, Yiheng Xu, Jiabo Ye, Xi Zhang, Tianbao Xie, Zesen Cheng, Hang Zhang, Zhibo Yang, Haiyang Xu, and Junyang Lin. Qwen2.5-vl technical report, 2025.

[10] Ping Chen, Shuibing He, Xuechen Zhang, Shuaiben Chen, Peiyi Hong, Yanlong Yin, Xian-He Sun, and Gang Chen. CSWAP: A self-tuning compression framework for accelerating tensor swapping in gpus. In *IEEE International Conference on Cluster Computing, CLUSTER 2021, Portland, OR, USA, September 7-10, 2021*, pages 271–282. IEEE, 2021.

[11] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv:1512.01274*, 2015.

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

[13] DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025.

[14] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wanjia Zhao, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaokang Zhang, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xinnan Song, Xinxia Shan, Xinyi Zhou, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Yang Zhang, Yanhong Xu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Yu, Yi Zheng, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Ying Tang, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yu Wu, Yuan Ou, Yuchen Zhu, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yukun Zha, Yunfan Xiong, Yunxian Ma, Yuting Yan, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhipeng Xu, Zhiyu Wu, Zhongyu Zhang, Zhuoshu Li, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Ziyi Gao, and Zizheng Pan. Deepseek-v3 technical report, 2025.

[15] Jiarui Fang, Zilin Zhu, Shenggui Li, Hui Su, Yang Yu, Jie Zhou, and Yang You. Parallel training of pre-trained models via chunk-based dynamic memory management. *IEEE Transactions on Parallel and Distributed Systems*, 34(1):304–315, 2023.

[16] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022.

[17] Hao Ge, Fangcheng Fu, Haoyang Li, Xuanyu Wang, Sheng Lin, Yujie Wang, Xiaonan Nie, Hailin Zhang, Xupeng Miao, and Bin Cui. Enabling parallelism hot switching for efficient training of large language models. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 178–194, New York, NY, USA, 2024. Association for Computing Machinery.

[18] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurelien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Roziere, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, Danny Wyatt, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Francisco Guzmán, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Govind Thattai, Graeme Nail, Gregoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel Kloumann, Ishan Misra, Ivan Evtimov, Jack Zhang, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Karthik Prasad, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, Khalid El-Arini, Krithika Iyer, Kshitiz Malik, Kuenley Chiu, Kunal Bhalla, Kushal Lakhotia, Lauren Rantala-Yeary, Laurens van der Maaten, Lawrence Chen, Liang Tan, Liz Jenkins, Louis Martin, Lovish Madaan, Lubo Malo, Lukas Blecher, Lukas Landzaat, Luke de Oliveira, Madeline Muzzi, Mahesh Pasupuleti, Mannat Singh, Manohar Paluri, Marcin Kardas, Maria Tsimpoukelli, Mathew Oldham, Mathieu Rita, Maya Pavlova, Melanie Kambadur, Mike Lewis, Min Si, Mitesh Kumar Singh, Mona Hassan, Naman Goyal, Narjes Torabi, Nikolay Bashlykov, Nikolay Bogoychev, Niladri Chatterji, Ning Zhang, Olivier Duchenne, Onur Çelebi, Patrick Alrassy, Pengchuan Zhang, Pengwei Li, Petar Vasic, Peter Weng, Prajjwal Bhargava, Pratik Dubal, Praveen Krishnan, Punit Singh Koura, Puxin Xu, Qing He, Qingxiao Dong, Ragavan Srinivasan, Raj Ganapathy, Ramon Calderer, Ricardo Silveira Cabral, Robert Stojnic, Roberta Raileanu, Rohan Maheswari, Rohit Girdhar, Rohit Patel, Romain Sauvestre, Ronnie Polidoro, Roshan Sumbaly, Ross Taylor, Ruan Silva, Rui Hou, Rui Wang, Saghar Hosseini, Sahana Chennabasappa, Sanjay Singh, Sean Bell, Seohyun Sonia Kim, Sergey Edunov, Shaoliang Nie, Sharan Narang, Sharath Raparthy, Sheng Shen, Shengye Wan, Shruti Bhosale, Shun Zhang, Simon Vandenhende, Soumya Batra, Spencer Whitman, Sten Sootla, Stephane Collot, Suchin Gururangan, Sydney Borodinsky, Tamar Herman, Tara Fowler, Tarek Sheasha, Thomas Georgiou, Thomas Scialom, Tobias Speckbacher, Todor Mihaylov, Tong Xiao, Ujjwal Karn, Vedanuj Goswami, Vibhor Gupta, Vignesh Ramanathan, Viktor Kerkez, Vincent Gonguet, Virginie Do, Vish Vogeti, Vítor Albiero, Vladan Petrovic, Weiwei Chu, Wenhan Xiong, Wenyin Fu, Whitney Meers, Xavier Martinet, Xiaodong Wang, Xiaofang Wang, Xiaoqing Ellen Tan, Xide Xia, Xinfeng Xie, Xuchao Jia, Xuewei Wang, Yaelle Goldschlag, Yashesh Gaur, Yasmine Babaei, Yi Wen, Yiwen Song, Yuchen Zhang, Yue Li, Yuning Mao, Zacharie Delpierre Coudert, Zheng Yan, Zhengxing Chen, Zoe Papakipos, Aaditya Singh, Aayushi Srivastava, Abha Jain, Adam Kelsey, Adam Shajnfeld, Adithya Gangidi, Adolfo Victoria, Ahuva Goldstand, Ajay Menon, Ajay Sharma, Alex Boesenberg, Alexei Baevski, Allie Feinstein, Amanda Kallet, Amit Sangani, Amos Teo, Anam Yunus, Andrei Lupu, Andres Alvarado, Andrew Caples, Andrew Gu, Andrew Ho, Andrew Poulton, Andrew Ryan, Ankit Ramchandani, Annie Dong, Annie Franco, Anuj Goyal, Aparajita Saraf, Arkabandhu Chowdhury, Ashley Gabriel, Ashwin Bharambe, Assaf Eisenman, Azadeh Yazdan, Beau James, Ben Maurer, Benjamin Leonhardi, Bernie Huang, Beth Loyd, Beto De Paola, Bhargavi Paranjape, Bing Liu, Bo Wu, Boyu Ni, Braden Hancock, Bram Wasti, Brandon Spence, Brani Stojkovic, Brian Gamido, Britt Montalvo, Carl Parker, Carly Burton, Catalina Mejia, Ce Liu, Changhan Wang, Changkyu Kim, Chao Zhou, Chester Hu, Ching-Hsiang Chu, Chris Cai, Chris Tindal, Christoph Feichtenhofer, Cynthia Gao, Damon Civin, Dana Beaty, Daniel Kreymer, Daniel Li, David Adkins, David Xu, Davide Testuggine, Delia David, Devi Parikh, Diana Liskovich, Didem Foss, Dingkang Wang, Duc Le, Dustin Holland, Edward Dowling, Eissa Jamil, Elaine Montgomery, Eleonora Presani, Emily Hahn, Emily Wood, Eric-Tuan Le, Erik Brinkman, Esteban Arcaute, Evan Dunbar, Evan Smothers, Fei Sun, Felix Kreuk, Feng Tian, Filippos Kokkinos, Firat Ozgenel, Francesco Caggioni, Frank Kanayet, Frank Seide, Gabriela Medina Florez, Gabriella Schwarz, Gada Badeer, Georgia Swee, Gil Halpern, Grant Herman, Grigory Sizov, Guangyi, Zhang, Guna Lakshminarayanan, Hakan Inan, Hamid Shojanazeri, Han Zou, Hannah Wang, Hanwen Zha, Haroun Habeeb, Harrison Rudolph, Helen Suk, Henry Aspegren, Hunter Goldman, Hongyuan Zhan, Ibrahim Damlaj, Igor Molybog, Igor Tufanov, Ilias Leontiadis, Irina-Elena Veliche, Itai Gat, Jake Weissman, James Geboski, James Kohli, Janice Lam, Japhet Asher, Jean-Baptiste Gaya, Jeff Marcus, Jeff Tang, Jennifer Chan, Jenny Zhen, Jeremy Reizenstein, Jeremy Teboul, Jessica Zhong, Jian Jin, Jingyi Yang, Joe Cummings, Jon Carvill, Jon Shepard, Jonathan McPhie, Jonathan Torres, Josh Ginsburg, Junjie Wang, Kai Wu, Kam Hou U, Karan Saxena, Kartikay Khandelwal, Katayoun Zand, Kathy Matosich, Kaushik Veeraraghavan, Kelly Michelena, Keqian Li, Kiran Jagadeesh, Kun Huang, Kunal Chawla, Kyle Huang, Lailin Chen, Lakshya Garg, Lavender A, Leandro Silva, Lee Bell, Lei Zhang, Liangpeng Guo, Licheng Yu, Liron Moshkovich, Luca Wehrstedt, Madian Khabsa, Manav Avalani, Manish Bhatt, Martynas Mankus, Matan Hasson, Matthew Lennie, Matthias Reso, Maxim Groshev, Maxim Naumov, Maya Lathi, Meghan Keneally, Miao Liu, Michael L. Seltzer, Michal Valko, Michelle Restrepo, Mihir Patel, Mik Vyatskov, Mikayel Samvelyan, Mike Clark, Mike Macey, Mike Wang, Miquel Jubert Hermoso, Mo Metanat, Mohammad Rastegari, Munish Bansal, Nandhini Santhanam, Natascha Parks, Natasha White, Navyata Bawa, Nayan Singhal, Nick Egebo, Nicolas Usunier, Nikhil Mehta, Nikolay Pavlovich Laptev, Ning Dong, Norman Cheng, Oleg Chernoguz, Olivia Hart, Omkar Salpekar, Ozlem Kalinli, Parkin Kent, Parth Parekh, Paul Saab, Pavan Balaji, Pedro Rittner, Philip Bontrager, Pierre Roux, Piotr Dollar, Polina Zvyagina, Prashant Ratanchandani, Pritish Yuvraj, Qian Liang, Rachad Alao, Rachel Rodriguez, Rafi Ayub, Raghotham Murthy, Raghu Nayani, Rahul Mitra, Rangaprabhu Parthasarathy, Raymond Li, Rebekkah Hogan, Robin Battey, Rocky Wang, Russ Howes, Ruty Rinott, Sachin Mehta, Sachin Siby, Sai Jayesh Bondu, Samyak Datta, Sara Chugh, Sara Hunt, Sargun Dhillon, Sasha Sidorov, Satadru Pan, Saurabh Mahajan, Saurabh Verma, Seiji Yamamoto, Sharadh Ramaswamy, Shaun Lindsay, Shaun Lindsay, Sheng Feng, Shenghao Lin, Shengxin Cindy Zha, Shishir Patil, Shiva Shankar, Shuqiang Zhang, Shuqiang Zhang, Sinong Wang, Sneha Agarwal, Soji Sajuyigbe, Soumith Chintala, Stephanie Max, Stephen Chen, Steve Kehoe, Steve Satterfield, Sudarshan Govindaprasad, Sumit Gupta, Summer Deng, Sungmin Cho, Sunny Virk, Suraj Subramanian, Sy Choudhury, Sydney Goldman, Tal Remez, Tamar Glaser, Tamara Best, Thilo Koehler, Thomas Robinson, Tianhe Li, Tianjun Zhang, Tim Matthews, Timothy Chou, Tzook Shaked, Varun Vontimitta, Victoria Ajayi, Victoria Montanez, Vijai Mohan, Vinay Satish Kumar, Vishal Mangla, Vlad Ionescu, Vlad Poenaru, Vlad Tiberiu Mihailescu, Vladimir Ivanov, Wei Li, Wenchen Wang, Wenwen Jiang, Wes Bouaziz, Will Constable,

Xiaocheng Tang, Xiaojian Wu, Xiaolan Wang, Xilun Wu, Xinbo Gao, Yaniv Kleinman, Yanjun Chen, Ye Hu, Ye Jia, Ye Qi, Yenda Li, Yilin Zhang, Ying Zhang, Yossi Adi, Youngjin Nam, Yu, Wang, Yu Zhao, Yuchen Hao, Yundi Qian, Yunlu Li, Yuzi He, Zach Rait, Zachary DeVito, Zef Rosnbrick, Zhaoduo Wen, Zhenyu Yang, Zhiwei Zhao, and Zhiyu Ma. The llama 3 herd of models. *arXiv:2407.21783*, 2024.

[19] Cong Guo, Rui Zhang, Jiale Xu, Jingwen Leng, Zihan Liu, Ziyu Huang, Minyi Guo, Hao Wu, Shouren Zhao, Junping Zhao, and Ke Zhang. Gmlake: Efficient and transparent gpu memory defragmentation for large-scale dnn training with virtual memory stitching. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 450–466, New York, NY, USA, 2024. Association for Computing Machinery.

[20] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding, 2016.

[21] Horace He and Shangdi Yu. Transcending runtime-memory tradeoffs in checkpointing by being fusion aware. In D. Song, M. Carbin, and T. Chen, editors, *Proceedings of Machine Learning and Systems*, volume 5, pages 414–427. Curan, 2023.

[22] Ding-Yong Hong, Tzu-Hsien Tsai, Ning Wang, Pangfeng Liu, and Jan-Jan Wu. Gpu memory usage optimization for backward propagation in deep network training. *Journal of Parallel and Distributed Computing*, 199:105053, 2025.

[23] Zhongzhe Hu, Junmin Xiao, Zheye Deng, Mingyi Li, Kewei Zhang, Xiaoyang Zhang, Ke Meng, Ninghui Sun, and Guangming Tan. Megtaichi: dynamic tensor-based memory management optimization for DNN training. In Lawrence Rauchwerger, Kirk W. Cameron, Dimitrios S. Nikolopoulos, and Dionisios N. Pnevmatikatos, editors, *ICS '22: 2022 International Conference on Supercomputing, Virtual Event, June 28 - 30, 2022*, pages 25:1–25:13. ACM, 2022.

[24] 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*, ASPLOS '20, page 1341–1355, New York, NY, USA, 2020. Association for Computing Machinery.

[25] 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. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[26] Insu Jang, Zhenning Yang, Zhen Zhang, Xin Jin, and Mosharaf Chowdhury. Oobleck: Resilient distributed training of large models using pipeline templates. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 382–395, New York, NY, USA, 2023. Association for Computing Machinery.

[27] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia*, MM '14, page 675–678, New York, NY, USA, 2014. Association for Computing Machinery.

[28] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. MegaScale: Scaling large language model training to more than 10,000 GPUs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 745–760, Santa Clara, CA, April 2024. USENIX Association.

[29] 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:2001.08361*, 2020.

[30] Seunghak Lee, Jin Kyu Kim, Xun Zheng, Qirong Ho, Garth A Gibson, and Eric P Xing. On model parallelization and scheduling strategies for distributed machine learning. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014.

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

[32] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. Pytorch distributed: Experiences on accelerating data parallel training. *Proc. VLDB Endow.*, 13(12):3005–3018, 2020.

[33] Yuchao Li, Fuli Luo, Chuanqi Tan, Mengdi Wang, Songfang Huang, Shen Li, and Junjie Bai. Parameter-efficient sparsity for large language models fine-tuning. In Luc De Raedt, editor, *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*, pages 4223–4229. ijcai.org, 2022.

[34] Zhuo Li, Hengyi Li, and Lin Meng. Model compression for deep neural networks: A survey. *Computers*, 12(3), 2023.

[35] Heng Liao, Jiajin Tu, Jing Xia, Hu Liu, Xiping Zhou, Honghui Yuan, and Yuxing Hu. Ascend: a scalable and unified architecture for ubiquitous deep neural network computing : Industry track paper. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 789–801, 2021.

[36] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory F. Diamos, Erich Elsen, David García, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.

[37] 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*, SOSP '19, page 1–15, New York, NY, USA, 2019. Association for Computing Machinery.

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

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

[40] OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni,

Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Jan Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr H. Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine B. Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. Gpt-4 technical report, 2024.

[41] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[42] 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*, ASPLOS '20, page 891–905, New York, NY, USA, 2020. Association for Computing Machinery.

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

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

[45] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16, 2020.

[46] 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*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[47] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, page 3505–3506, New York, NY, USA, 2020. Association for Computing Machinery.

[48] 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, 2021.

[49] 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. USENIX Association, July 2021.

[50] 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 *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*, pages 18:1–18:13. IEEE Computer Society, 2016.

[51] Minsoo Rhu, Mike O'Connor, Niladrish Chatterjee, Jeff Pool, Youngeun Kwon, and Stephen W. Keckler. Compressing DMA engine: Leveraging activation sparsity for training deep neural networks. In *IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24-28, 2018*, pages 78–91. IEEE Computer Society, 2018.

[52] Frank Seide and Amit Agarwal. Cntk: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 2135, New York, NY, USA, 2016. Association for Computing Machinery.

[53] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units, 2016.

[54] 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:1909.08053*, 2019.

[55] Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, Zhuofu Chen, Jialei Cui, Hao Ding, Mengnan Dong, Angang Du, Chenzhuang Du, Dikang Du, Yulun Du, Yu Fan, Yichen Feng, Kelin Fu, Bofei Gao, Hongcheng Gao, Peizhong Gao, Tong Gao, Xinran Gu, Longyu Guan, Haiqing Guo, Jianhang Guo, Hao Hu, Xiaoru Hao, Tianhong He, Weiran He, Wenyang He, Chao Hong, Yangyang Hu, Zhenxing Hu, Weixiao Huang, Zhiqi Huang, Zihao Huang, Tao Jiang, Zhejun Jiang, Xinyi Jin, Yongsheng Kang, Guokun Lai, Cheng Li, Fang Li, Haoyang Li, Ming Li, Wentao Li, Yanhao Li, Yiwei Li, Zhaowei Li, Zheming Li, Hongzhan Lin, Xiaohan Lin, Zongyu Lin, Chengyin Liu, Chenyu Liu, Hongzhang Liu, Jingyuan Liu, Junqi Liu, Liang Liu, Shaowei Liu, T. Y. Liu, Tianwei Liu, Weizhou Liu, Yangyang Liu, Yibo Liu, Yiping Liu, Yue Liu, Zhengying Liu, Enzhe Lu, Lijun Lu, Shengling Ma, Xinyu Ma, Yingwei Ma, Shaoguang Mao, Jie Mei, Xin Men, Yibo Miao,

Siyuan Pan, Yebo Peng, Ruoyu Qin, Bowen Qu, Zeyu Shang, Lidong Shi, Shengyuan Shi, Feifan Song, Jianlin Su, Zhengyuan Su, Xinjie Sun, Flood Sung, Heyi Tang, Jiawen Tao, Qifeng Teng, Chensi Wang, Dinglu Wang, Feng Wang, Haiming Wang, Jianzhou Wang, Jiaxing Wang, Jinhong Wang, Shengjie Wang, Shuyi Wang, Yao Wang, Yejie Wang, Yiqin Wang, Yuxin Wang, Yuzhi Wang, Zhaoji Wang, Zhengtao Wang, Zhexu Wang, Chu Wei, Qianqian Wei, Wenhao Wu, Xingzhe Wu, Yuxin Wu, Chenjun Xiao, Xiaotong Xie, Weimin Xiong, Boyu Xu, Jing Xu, Jinjing Xu, L. H. Xu, Lin Xu, Suting Xu, Weixin Xu, Xinran Xu, Yangchuan Xu, Ziyao Xu, Junjie Yan, Yuzi Yan, Xiaofei Yang, Ying Yang, Zhen Yang, Zhilin Yang, Zonghan Yang, Haotian Yao, Xingcheng Yao, Wenjie Ye, Zhuorui Ye, Bohong Yin, Longhui Yu, Enming Yuan, Hongbang Yuan, Mengjie Yuan, Haobing Zhan, Dehao Zhang, Hao Zhang, Wanlu Zhang, Xiaobin Zhang, Yangkun Zhang, Yizhi Zhang, Yongting Zhang, Yu Zhang, Yutao Zhang, Yutong Zhang, Zheng Zhang, Haotian Zhao, Yikai Zhao, Huabin Zheng, Shaojie Zheng, Jianren Zhou, Xinyu Zhou, Zaida Zhou, Zhen Zhu, Weiyu Zhuang, and Xinxing Zu. Kimi k2: Open agentic intelligence, 2025.

[56] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. Bamboo: Making preemptible instances resilient for affordable training of large DNNs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 497–513, Boston, MA, April 2023. USENIX Association.

[57] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. Superneurons: dynamic gpu memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, page 41–53, New York, NY, USA, 2018. Association for Computing Machinery.

[58] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. Emergent abilities of large language models, 2022.

[59] Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, Zhengyan Zhang, Zhenda Xie, Y. X. Wei, Lean Wang, Zhiping Xiao, Yuqing Wang, Chong Ruan, Ming Zhang, Wenfeng Liang, and Wangding Zeng. Native sparse attention: Hardware-aligned and natively trainable sparse attention, 2025.

[60] Tailing Yuan, Yuliang Liu, Xucheng Ye, Shenglong Zhang, Jianchao Tan, Bin Chen, Chengru Song, and Di Zhang. Accelerating the training of large language models using efficient activation rematerialization and optimal hybrid parallelism. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 545–561, Santa Clara, CA, July 2024. USENIX Association.

[61] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. Pytorch fsdp: Experiences on scaling fully sharded data parallel. *Proc. VLDB Endow.*, 16(12):3848–3860, August 2023.

A Optimization in the matching mechanism within the Executor

First, we use multi-feature matching to replace the naive (operator ID, the i-th tensor used by the operator) to adjust to minor changes occur in the sequence.

```
bool Tensor::operator==(Tensor& other) {
  return opCount==other.opCount &&
    opTag==other.opTag &&
    dtype==other.dtype &&
    opCallStack==other.opCallStack &&
    ...;
}
```

Then, we implement the following trick to accelerate the matching process and avoid host-bound situations.

```
void Tensor::update(...) {
  opCount++;
  // Record operators that have used this tensor
  opTag|=opOneHot;
  opCallStack=(opCallStack<<8)+opIndex;
  ...
}
```

- From the perspective of tensors, when comparing the operators that have used the tensor, instead of using string arrays for the comparison, we first identify the 32 most frequently occurring operators during profiling. Each operator is then assigned a one-hot encoding, and operator matching is performed using bitwise operations.
- We also assign each operator an index based on its frequency of occurrence during training. When recording the tensor's call stack, we use an integer variable, opCallStack, to track calls through bit-shifting. Each call is allocated 8 bits, and only the last 8 calls are recorded. This approach is sufficient for our needs.

B Algorithm of OOM Handling in the Warm-Up Stage

In §6.2, we shift synchronization from the swap stream–host path (different paces) to the swap stream–compute stream path (same pace), avoiding host-bound overhead. We apply the same principle to OOM scenarios.

Swap is introduced to overcome device memory limits and enable training of larger models. At the start of training, before an effective swap policy exists, the model inevitably exceeds device capacity, triggering OOM. To keep training uninterrupted and profiling data intact, we designed an optimized OOM handling process (Algo. 3). Upon OOM, all memory blocks marked by the custom RecordStream are released, including those in the middle of swapping or completed swaps awaiting release. Instead of incurring a host-device synchronization and blocking the host, an event record/wait pair is inserted between the swap and compute streams to ensure these blocks are reused only after swapping completes. This prevents conflicts and keeps the host free to dispatch operators. Once the swap stream event is triggered, the compute stream proceeds immediately without waiting for further host actions, as in Fig. 9.

We use GMLake [19] as the memory pool to reduce fragmentation during frequent swaps. After leveraging it for defragmentation, we retry the allocation. If OOM persists,

Table 2. Performance Benefit Data from Chameleon. (SS. is short for Chameleon.)

Model	#NPU	TP	PP	DP	Seq Len	Hidden Size	FFN Size	Num Heads	Layers	MBS	GBS	enable Recomp.	enable SS.	Time (s/step)	Perf. Benefit(%)
Llama2	8	8	1	1	8,192	4,096	11,008	32	32	1	16	0	0	6.83	
Llama2	8	1	1	8	8,192	4,096	11,008	32	32	1	16	0	1	5.42	25.63
Llama2	8	8	1	1	4,096	5,120	13,824	40	40	1	16	0	0	6.30	
Llama2	8	2	1	4	4,096	5,120	13,824	40	40	1	16	0	1	5.88	7.14
Llama2	8	1	2	4	4,096	4,096	11,008	32	32	1	16	0	0	3.20	
Llama2	8	1	1	8	4,096	4,096	11,008	32	32	1	16	0	1	3.02	5.96
Llama2	32	1	2	16	4,096	4,096	11,008	32	32	1	64	0	0	2.53	
Llama2	32	1	1	32	4,096	4,096	11,008	32	32	1	64	0	1	2.34	8.06
Llama2	8	1	2	4	16,384	4,096	11,008	32	14	1	8	0	0	4.46	
Llama2	8	1	1	8	16,384	4,096	11,008	32	14	1	8	0	1	3.21	38.94
Llama2	8	4	1	2	16,384	5,120	13,824	40	40	1	8	1	0	18.19	
Llama2	8	4	1	2	16,384	5,120	13,824	40	40	1	8	0	1	14.13	28.73
Llama3	4	4	1	1	8192	4096	14336	32	32	2	8	1	0	9.41	
Llama3	4	4	1	1	8192	4096	14336	32	32	2	8	0	1	7.31	28.73
Mixtral	4	4	1	1	16384	4096	14336	32	36	1	4	1	0	11.79	
Mixtral	4	4	1	1	16384	4096	14336	32	36	1	4	0	1	9.06	30.13

Table 3. Results Demonstrating Chameleon's Scalability. (M is short for Measured and E is short for Estimated.)

#NPU	TP	PP	DP	Seq Len	Hidden Size	FFN Size	Num Heads	Layers	MBS	GBS	enable SS.	M-/E-Mem Used(MB)	M-Time (s/step)	E-Time (s/step)	Perf. Impact(%)
8	1	2	4	4,096	4,096	11,008	32	20	1	32	0	37,066(M)	3.50		
8	1	2	4	4,096	4,096	11,008	32	28	1	32	0	49,206(M)	4.83		
8	1	2	4	4,096	4,096	11,008	32	42	1	32	1	70,451(E)	7.33	7.16	-2.32
8	4	1	2	16,384	5,120	20,480	32	4	4	8	0	34,256(M)	2.06		
8	4	1	2	16,384	5,120	20,480	32	4	6	8	0	45,216(M)	3.00		
8	4	1	2	16,384	5,120	20,480	32	4	11	8	1	72,616(E)	5.36	5.35	-0.18
8	4	2	1	16,384	8,192	32,768	32	6	2	8	0	42,918(M)	6.33		
8	4	2	1	16,384	8,192	32,768	32	8	2	8	0	54,259(M)	8.27		
8	4	2	1	16,384	8,192	32,768	32	12	2	8	1	76,941(E)	12.2	12.15	-0.41

Algorithm 3 Algorithm of handling OOM.

```
1:  Try malloc(args)
2:  Catch OOM:
3:      FreeSwappingOutBlock()
4:      SwapStream.eventRecord()
5:      ComputeStream.eventWait()
6:      MemoryPool.Defragment()
7:      Try malloc(args)
8:      Catch OOM:
9:          PassiveSwap()
10:         FreeSwappingOutBlock()
11:         SwapStream.eventRecord()
12:         ComputeStream.eventWait()
13:         Try malloc(args)
14:         Catch OOM:
15:             Throw Error
```

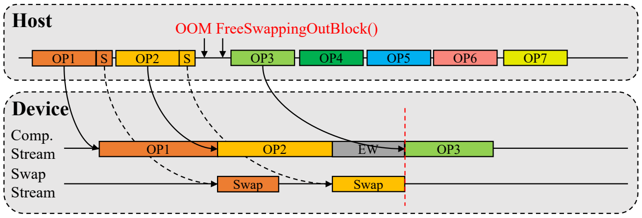


Figure 9. Illustration of optimized OOM Handling Process.

we select the tensor whose size is closest to the required block for a passive swap as illustrated in line 7 of Algo. 3, again using inter-stream synchronization to avoid host blocking. This process handles OOM in any Warm-Up stage iteration, including the very first. For the first iteration, Chameleon generates no swap policy, so FreeSwappingOutBlock() in line 3 frees nothing. Upon OOM recurrence, PassiveSwap() in line 7 is invoked to release enough space for allocation.

C Additional Experimental Results

C.1 Performance Benefit from Chameleon

Under identical hardware constraints, Chameleon not only enables training of larger models but also allows training of the same-sized models using fewer NPUs. This, in turn, reduces the reliance on expensive tensor parallelism (TP) and

Table 4. Maximum supported model per dimension.

Test	Ori PyTorch	Chameleon
Batch size	6	24
Number of layers	6	11
Seq. length	6144	24576
Hidden size	4864	6016

pipeline parallelism (PP), replacing them with data parallelism (DP). Since the communication volume involved in TP and PP is significantly greater than that in DP, substituting TP and PP with DP decreases the proportion of training time spent on non-overlappable communication and increases the proportion devoted to computation, thereby accelerating training and improving hardware utilization. Furthermore, by enabling Chameleon and disabling recomputation, we can avoid the additional computation introduced by recomputation on the critical path, yielding substantial performance gains. We conduct experiments to evaluate all of the above scenarios in which Chameleon may deliver performance improvements, with the results presented in Table 2.

The data in Table 2 are organized in pairs, each corresponding to performance evaluations under identical model configurations, where Chameleon is enabled to reduce the TP domain, reduce the PP domain, or disable recomputation, thereby assessing the resulting performance benefits.

C.2 Results Demonstrating Chameleon's Scalability

Table 3 presents detailed results of our layer-wise scaling experiments under various model configurations. Modern LLMs are typically constructed by stacking transformer layers, and increasing the number of layers is a common approach to enhance model capability. In this experiment, we evaluate the performance of Chameleon when scaling models by adding layers under different model configurations.

In the table, data are grouped in triplets. For each configuration, we first train the model with two different layer counts, with which the model size does not exceed device memory, and we measure both the peak memory usage and the per-step runtime for each. We then apply linear extrapolation to estimate the memory usage and per-step runtime for a hypothetical model whose layer count would cause an OOM error. Comparing the predicted per-step runtime to the actual runtime measured with Chameleon enabled allows us to quantify its performance impact.

The results demonstrate that Chameleon can train substantially larger models across diverse configurations, incurring only minimal performance degradation.

In addition to the above results, we further explore the upper bound of Chameleon's capability. We initialize the batch size, number of layers, sequence length, and hidden size to 4, 5, 4096, and 4096, respectively. In each experiment, we fixe three of these variables and increase the fourth until an OOM occurred, recording the maximum value before failure. The results, presented in Table 4, show that compared to the original PyTorch implementation, Chameleon can accommodate models exceeding the hardware memory capacity by up to 4×, 1.83×, 4×, and 1.24× along the three dimensions, respectively.