A Study of Data-Path Bugs in PyTorch with a Focus on Memory Management

Rubayet Rahman Rongon
Washington State University
Vancouver, Washington
r.rongon@wsu.edu

Chen Cao
Pennsylvania State University Behrend
Erie, Pennsylvania
ccao@psu.edu

Xuechen Zhang
Washington State University
Vancouver, Washington
xuechen.zhang@wsu.edu

***Abstract*—This paper presents a comprehensive and quantitative study of bugs related to *Data Path* in PyTorch with a focus on tensor management in memory. The bugs were reported from 2017 to 2024. Analyzing 3,089 closed issues, we identified 11 distinct bug types affecting the data storage, allocation, and loading, including memory bugs, indexing errors, and tensor contiguity violations. Our analysis reveals that data-path bugs have more occurrences than bugs related to computation in PyTorch in recent years. Among the memory bugs, non-contiguity bugs account for 30.2% of the total number of bugs and they have the most significant impact, leading to both crashes and silent correctness failures. One of the common solutions to addressing non-contiguity bugs is transforming from non-contiguous data to contiguous data in memory before machine-learning computation. To assess the impact of memory layout transformation, we conducted experiments involving tensor augmentation and non-contiguous tensor conversion. Our findings demonstrate that maintaining tensor contiguity throughout the augmentation process can improve performance by up to 73.9%, while the time required for non-contiguous tensor conversion varies significantly based on the number and order of dimensions.**

Our research provides valuable insights for developers and researchers working with PyTorch, helping them to identify and address potential bugs in data paths and tensor memory management.

***Index Terms*—Deep Learning, Bug Analysis, PyTorch**

I. Introduction

Deep Neural networks (DNNs) have been successfully used in various domains, such as computer vision, recommendation systems, speech recognition, etc. PyTorch is one of the dominating open-source machine learning libraries for training DNNs [1]. PyTorch needs to manage a large amount of data in host servers and GPUs for tensor creation, transformation, and computation. As a result, data paths and memory management in Pytorch affect the performance and correctness of DNN model training and inference.

Today's data paths (i.e., storage and memory systems) in PyTorch suffer from various types of errors, including segment faults, indexing errors, concurrency errors, etc. In this paper, we perform a comprehensive study of the open-source PyTorch software. We examine the patches committed over seven years from 2017 to 2024. The study covers 3,089 issues across PyTorch versions from 1.3.0 to 2.4. We manually label each issue after carefully checking the patch, its descriptions, and follow-up discussions posted by developers. The insights derived from the study can help developers build more reliable and efficient data paths and memory systems in PyTorch and develop associated debugging tools.

We first investigated the bugs related to the data path in PyTorch from 2017 to 2024 by analyzing 3,089 closed issues from the PyTorch issue repository. We identified 101 issues categorized as bugs, silent bugs, crashes, or those producing NaNs or Infinity. Focusing on those with at least one patch, we conducted a detailed analysis of 73 bugs. We discovered 12 distinct types of bugs affecting the data path. Among these, memory-related bugs were the most prevalent, accounting for 48.4% of the total bugs with patches. Notably, these memory bugs have been predominantly distributed from 2020 to 2024.

Second, we analyze memory system bugs with patches, focusing on their types and impacts. We categorize these bugs into 11 types, including non-contiguity in memory, segment fault, indexing, etc. Our analysis highlights that non-contiguity bugs are the most impactful, leading to crashes and silent correctness failures that are difficult to detect. Segment faults are associated with severe system failures, while indexing bugs contribute to runtime errors.

Finally, we observe that most of the non-contiguity bugs were resolved by memory layout transformation. Although the patches ensure the code correctness, they ignore the impact of these patches on the performance (e.g., latency) of data paths. In this paper, we conducted two experiments to assess the impact of memory layout on tensor operations. The first experiment compared tensor augmentation performance between a memory-layout-aware approach, which ensures tensor contiguity throughout the augmentation process, and the default PyTorch approach, which permits alternating tensor contiguity. The second experiment measured the time required to convert non-contiguous tensors to contiguous ones, focusing on how the number of dimensions affect conversion time. We have the following observations from the results. (1) Maintaining contiguous tensors during augmentation resulted in a 73.9% performance improvement on CIFAR-10 dataset. (2) Converting a permuted non-contiguous tensor to contiguous ones took minimal time for two-dimensional permutations but up to 53 seconds for five-dimensional permutations.

The rest of this paper is organized as follows. Section II gives a brief description of related work. In Section III we describe the methodology of our bug study. Section IV describes the overall patterns in data-path bugs. Section V

studies the causes and impact of memory bugs. Section VI experimentally studies the performance impact of memory layout transformation widely used for addressing memory bugs. Finally, in Section VII, we share our suggestions, followed by conclusions in Section VIII.

II. Related Work

Bug analysis and distribution: Recent related works [2]–[6] have adapted examining their relationships with symptoms and root causes from various perspectives. Ho et al. [2] analyzed 194 bugs in TensorFlow and PyTorch to discern performance issues. Chen et al. [3] expanded the scope by incorporating 1000 bugs from 4 popular frameworks - TensorFlow, PyTorch, MXNet, and DL4J - characterizing bug distribution across the five layers of framework architecture. Makkouk et al. [4] scrutinized 17,893 and 16,284 bug reports from TensorFlow and PyTorch, respectively, shedding light on performance and non-performance bug complexity, fix time, and fix size. Jia et al. [5] conducted a meticulous examination of 202 manually selected TensorFlow bugs to explore bug locations. Yang et al. [6] analyzed 1,127 bug reports from eight DL frameworks. Another study [7] segregated pulled bugs into machine learning (ML) and non-ML bugs, analyzing 109 ML bugs to investigate their fixing time. Du et al. [8] initially classified 3,555 TensorFlow, MXNet, and PaddlePaddle bugs into Bohrbugs and Mandelbugs types, later extending into bug classification, correlation among types, and fixing time. Tambon et al. [9] focused on a specific bug subcategory - silent bugs - collecting 1,168 closed issues from Keras and TensorFlow. Different from these, our work focuses on analyzing memory bugs that occurred in data paths.

Tools derived from different bug study works: Various approaches have been adapted following bug analysis in DL frameworks. Ho et al. [2] identified patterns for 84 issues, leading to the discovery of eight repair patterns. Chen et al. [3] proposed TENFUZZ, a tool primarily focusing on tensor operations, which tests frameworks by mutating tensor type, shape, structure, and parameter values. Jia et al. [5] prioritized bug location/component, categorizing repair patterns and proposing new ones for TensorFlow. Yang et al. [6] discussed 15 fixing patterns identified from 143 bug reports. Tambon et al. [9] showed the distribution of silent bugs with threat levels and verified them through developer surveys.

These [10], [11] two works proposed developing a testing tool for DL frameworks. Audee was able to identify [10] 26 unknown bugs where COMET found 32 new bugs. Both of the ideas mutated model test cases for multiple layers in DL frameworks. However, they differed in the way of searching test cases. Audee adopted genetic algorithms and COMET used parameter analysis and a random sampling-based approach.

Optimizations for non-contiguous data in memory: Several studies highlight the performance challenges and solutions related to non-contiguous data in GPU computing. [12] emphasizes that tensor gradient computation benefits from contiguous memory storage, as non-contiguous storage can significantly hinder the unfolding process and computational efficiency. [13] identifies that repeatedly launching GPU kernels for packing and unpacking operations introduces overhead, degrading performance across successive operations. [14] also points out the additional computational resources required for rearranging non-contiguous data into contiguous blocks, which increases latency. Furthermore, [15] discusses how typical PyTorch implementations of non-contiguous pooling operations result in high GPU memory usage and slower performance due to inefficient memory handling. This issue was mitigated by developing custom CUDA kernels that directly handle non-contiguous memory, thereby improving efficiency and reducing memory usage.

Research highlights several challenges associated with non-contiguous data in GPU computing. [16] notes that traditional data transfer methods struggle with non-contiguous data due to the need for extra memory copies and rearrangement, leading to increased latency and resource usage. [17] discusses inefficiencies such as irregular memory access and complex workload distribution, which can slow down processing and necessitate advanced load-balancing strategies. According to [18], the CUDA graph programming model also suffers from kernel call overhead and multiple memory accesses when handling non-contiguous memory. Additionally, [19] points out that traditional matrix multiplication routines and tensor operations, optimized for contiguous data, face performance issues with non-contiguous tensors due to costly reshaping and temporary arrays. Lastly, [20] highlights that non-contiguous memory allocations lead to fragmentation, which hampers memory management efficiency and increases allocation overhead in deep learning frameworks.

III. Methodology

We focus on PyTorch because of its prominent role in deep learning and its evolving capabilities for high-performance computing. Our study was conducted in three phases. First, we categorized PyTorch's code into **data path** and **computation** components, tracking code growth from version 1.3.0 to the latest release to pinpoint significant changes and potential bug areas. Second, we analyzed 3,089 closed issues from 2017 to 2024, identifying 101 critical bugs, with 73 reviewed in detail by experts to validate fixes and reveal patterns. Finally, we manually examined 29 non-contiguous data bugs from 2020 to 2024, with expert analysis to understand root causes and performance impacts. This approach provided a thorough understanding of key issues in PyTorch's memory management and tensor operations.

First, we identify the source codes that have been frequently modified over the years. For this purpose, we divide the PyTorch source code into two primary components: **data path** and **computation**. The data-path component encompasses the management of data storage, allocation, and loading, involving PyTorch modules such as `c10/core`, `torch/utils`, `torch/backends`, and `torch/sparse`. The computation component includes the code responsible for neural network training and optimization, with key modules like

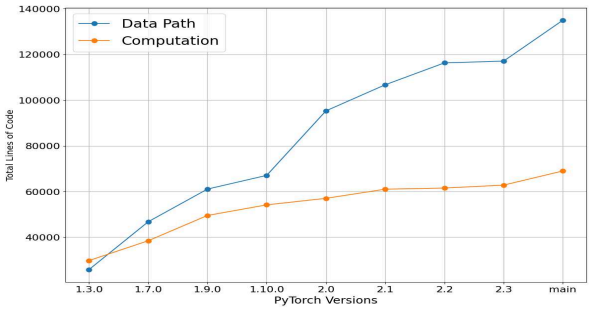

Fig. 1: The change in the Pytorch code in terms of LOC across different PyTorch versions

`torch/autograd`, `torch/nn`, and `torch/optim`. By analyzing the growth in lines of code for these components from PyTorch version 1.3.0 to the most recent version on the 'main' branch, we aim to identify which areas have experienced the most significant changes over time. This approach helps us pinpoint where the codebase has evolved the most, allowing us to focus on finding and studying bugs related to the more frequently modified parts of the source code.

Second, we analyze the major causes and consequences of memory bugs because they account for almost 50% of the bugs in the data-path component. Our second phase of analysis began by extracting data from the PyTorch issue repository, focusing on closed issues between 2017 and 2024. From this, we identified 3,089 closed issues related to the data path and filtered for those issues as bugs, silent bugs, crashes, or issues that resulted in NaNs or Infinity during computation. This filtering process resulted in 101 distinct bugs. We further narrowed our focus to 73 bugs that had at least one patch applied, ensuring that only issues with confirmed fixes were selected for detailed examination. This phase involved a thorough investigation by experts, who meticulously reviewed each bug and its corresponding patch to validate the fix and uncover any underlying patterns. Their expertise was crucial in identifying the root causes and ensuring that the fixes effectively addressed the issues without introducing new problems.

In the final phase of our study, we focused on a particular type of memory bug: non-contiguous bugs because they are unique in PyTorch. They cause many silent bugs and have never been comprehensively studied in the literature. The non-contiguous data bugs often stem from inefficient memory layouts and incorrect assumptions during tensor manipulations. To gain deeper insights, we manually examined 29 issues from the PyTorch repository, covering the period from 2020 to 2024. This manual analysis was conducted with the involvement of expert contributors who possess in-depth knowledge of PyTorch's internals. Together, we systematically reviewed each bug, beginning with the initial report and tracing the issue through the applied patches and resolution. This hands-on approach allowed us to carefully identify the root causes, such as improper handling of non-contiguous tensors in memory, and to understand the wider consequences, including silent errors, crashes, or performance bottlenecks. The expert input was invaluable in ensuring an accurate interpretation of complex bugs and their underlying mechanisms, providing a clear picture of how these issues arose and were subsequently addressed. This methodical examination also allowed us to assess the impact on performance, revealing patterns that could guide improvements in PyTorch's memory management and tensor operations.

IV. Evolution of Data-Path Bugs

In this section, we conduct an evolution study of PyTorch bugs. We first analyze the lines of code (LOC) in **Data Path** and **Computation** components of PyTorch. As shown in Figure 1, the analysis of LOC reveals a substantial growth disparity between the two components. The data-path component has expanded by approximately 833%, from 15,000 LOC to 140,000 LOC, reflecting a major increase in data handling and management capabilities. In comparison, the computation component has grown by around 250%, from 20,000 LOC to 70,000 LOC. This indicates that while both components have seen significant growth, the data-path code has experienced a much more pronounced increase, highlighting its enhanced focus on complex data management in memory and storage.

Bug type	**Definition**
Memory	Errors related to memory management, such as illegal memory access, memory corruption, segmentation faults, improper handling of memory layouts, memory leaks, or buffer overflows.
Data Pipe	A composable component in PyTorch that represents a sequence of data processing operations.
Device Management	Controlling and communicating with hardware devices, like CPUs and GPUs, to perform operations.
Logical	Bugs that stem from improper implementation in the program's logic rather than syntax or runtime errors.
Data Type	Errors caused by mismatches or improper handling of different data types in PyTorch.
File Descriptor	An identifier used to manage open files or communication channels like sockets and shared memory between processes.
Indexing	The process of accessing and manipulating specific elements or slices of tensors using indices, allowing for selective data retrieval and modification.
Initialization	Errors that occur due to improper or incomplete initialization of elements.
ONNX	Errors occurring during or after the conversion of PyTorch models to ONNX format due to issues or limitations in the conversion process or resulting ONNX representation.
Resource Limit	Issues that occur when the program requests more system resources than are available, leading to failures or issues due to improper handling of resource constraints.
Thread Management	Issues related to the handling and synchronization of multiple threads, which can lead to warnings, crashes, or unpredictable behavior when threading resources are not properly managed or configured.

TABLE I: Definitions of bug types in the data-path component in PyTorch.

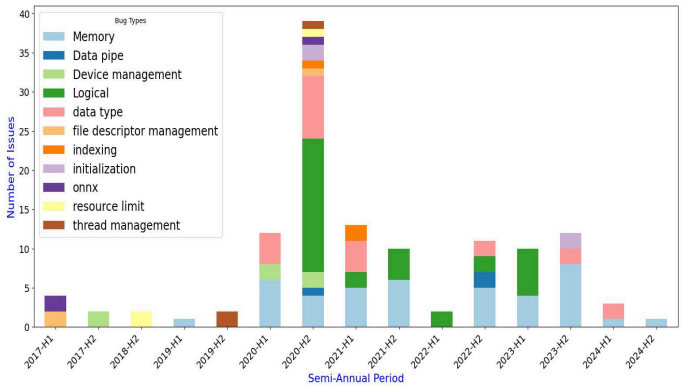

Fig. 2: Bug distribution in the source code of the data path module over time.

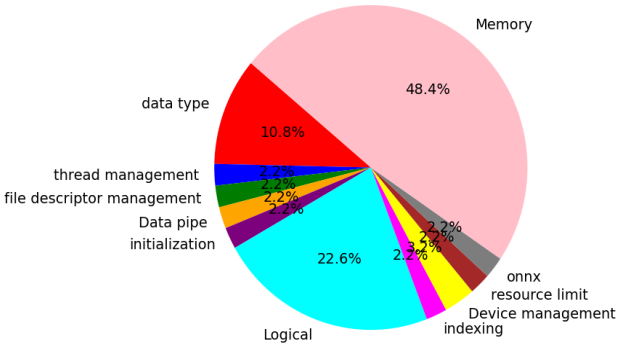

Fig. 3: Bug type distribution on Data Path.

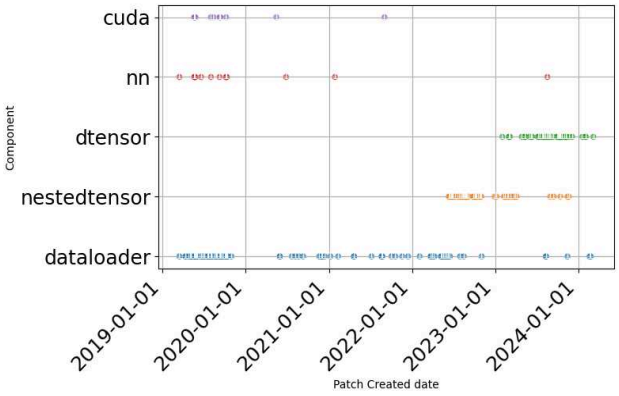

Fig. 4: Data-path patches submitted from 2019 to 2024. The patches are grouped based on components.

The analysis of the data-path component in PyTorch reveals substantial growth in managing complex data structures. The `torch/nested` module expanded from 149 lines to 3,537 lines, and the `torch/sparse` module grew dramatically from 136 lines to 11,677 lines, with both increases occurring primarily from version 2.0.0 to the main branch. This significant growth reflects a heightened focus on supporting hierarchical and sparse data structures. Additionally, advancements in `torch/distributed/tensor` and `torch/distributed/pipelining` highlight progress in distributed data management. These developments emphasize PyTorch's ongoing efforts to enhance its capabilities for handling and processing diverse data types.

The data-path component in PyTorch is frequently updated, much like other subsystems in the software. The source code in the data-path component has more changes than computation as shown in Figure 1. However, few comprehensive studies have focused on bugs in this critical area. In our research, we examine bugs related to code in the data path between 2017 and 2024 by analyzing closed issues from the PyTorch issue repository. From a total of 3,089 closed issues related to the data path, we identified 101 that were labeled as bugs, silent bugs, crashes, or those returning NaNs or Infinity. We further investigated only those issues that had at least one patch applied to fix the problem, resulting in a detailed analysis of 73 such bugs. We found 12 types of bugs as shown in Figure 2. The definition of each bug type is described in Table I.

1) How is the Data Path Evolving?: Figure 2 illustrates the distribution of patches across various components within the data path, addressing issues that could otherwise result in crashes, NaNs, silent bugs, or infinite values. Early in the development cycle (2017-2018), bugs were more concentrated around foundational issues like file descriptor management, device handling, and ONNX framework integration. However, as the system matured, more complex issues began to surface. By 2020-2022, there was a notable increase in logical, data type, and memory-related bugs, indicating that while the core system had stabilized, challenges were shifting toward efficient data handling and performance optimization. Since 2020, memory-related bugs have been consistently occurring in every semi-annual period, except the first half of 2022. This steady presence underscores the ongoing challenges and critical nature of memory management in Pytroch software development.

2) Where did the Data Path Change?: Figure 3 illustrates the distribution of bug types within the data-path category. The most prominent bug type is `Memory`, accounting for 48.4% of the total. This is followed by `Logical` and `Data type`, which constitute 22.6% and 10.8%, respectively.

In PyTorch, `Memory` issues have been a significant contributor to crashes, correctness (silent) failures, and unexpected outcomes with a marked prevalence during the period from 2020 to 2024. Given their substantial impact, it is crucial to conduct further research into `Memory` bugs to improve the stability and performance of the PyTorch software.

Type	Subtype	Description
Non-contiguous	CUDA limitation	CUDA software constraint on specific memory structure.
	Information loss	Stride information loss.
	Input tensor management	Issue while managing non-contiguous data as input tensor.
	Channel management	Issue with non-contiguous data on convolution layer channel layout.
Segment fault	Memory Access violation	Accessing incorrect memory address.
	Invalid memory address	Accessing memory that does not exist anymore.
	Memory corruption	A program unintentionally modifies memory, leading to unpredictable behavior, crashes, or security vulnerabilities.
	Programming	Failure due to developer's implementation.
Indexing	Overflow	Indexing boundary overflow.
	Data type	Issues due to use of wrong data types as element index in tensors.
	Invalid class type	Indexing data type is of invalid class type.
	Invalid memory access	Accessing memory that does not exist anymore while indexing.
Memory format	Channel management	Convolution layer fails to correctly format the output
	Stride calculation manually	PyTorch fails to calculate output memory format, thus needs manual memory format
Logic	Order	Code implementation order.
Pin memory	Input tensor management	Pinning page table failure for tensor.
	Programming	Failure due to developer's implementation.
Concurrency	Miss lock	Expected lock issue.
Layout of nested tensor	Storage format mismatch	Storage format mismatch for nested tensor.
Memory overflow	Programming	Failure due to developer's implementation.
Checking	Input checking	Input data type or class, subclass checking.
Initialization	Order	Code implementation order.

TABLE II: Table of Types, Subtypes, and Descriptions

Besides, we can see from Figure 4 that the number of patches applied to *distributed tensors* (*dtensor*) and *nested tensor* is higher than any other components in PyTorch since mid of 2022. This shows the shift of development efforts from model computation to the data path.

Summary: (1) PyTorch has been actively improving its data path. (2) Recently, the code changes in the data path are highly concentrated on supporting its training using nested and distributed tensors in large-scale distributed systems. (3) 48.4% of the patches of data path are related to addressing memory bugs.

V. Memory Bugs in Data Path of PyTorch

A memory bug refers to any issues in memory allocation, access, or management that can lead to crashes, data corruption, or performance problems. In this section, we examine the memory bugs in the **Data Path** of PyTorch in detail to understand their patterns and consequences. We focus on the memory bugs because (1) they account for nearly 50% of the bugs in the data path and (2) non-memory bugs have been well-studied in the previous work, e.g., [2]–[4].

A. What are the Memory Bugs in PyTorch?

We categorize these bugs into 11 types: non-contiguous memory, segment fault, indexing, memory layout, pin memory, memory overflow, concurrency, logic, initialization, layout of nested tensor, and checking. Each type addresses specific memory-related challenges. Within each type, various subtypes further detail the nature of the bugs. The complete breakdown of these subtypes and their descriptions can be found in Table II.

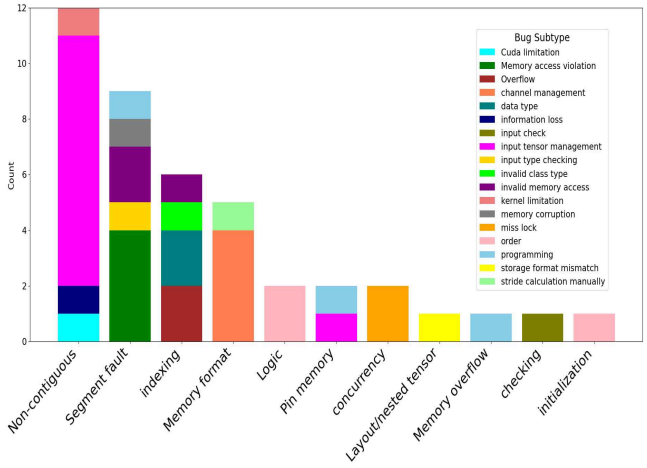

Fig. 5: The distribution of memory bugs and sub-type breakdown.

B. How are Memory Bugs Distributed?

Figure 5 shows the distribution of memory bugs in PyTorch. A majority of these bugs lie in 5 sub-types: *input tensor management*, *channel layout management*, *invalid memory access*, *order* and *programming limitations*. More specifically, we identify several interesting findings in the memory bugs.

1. Non-contiguity bugs (30.2%): Non-contiguous data refers to data elements that are not stored in adjacent memory locations. The non-contiguous data layout can occur in various PyTorch data structures where tensor elements are spread out across different memory locations rather than being laid out

in a single, continuous memory block. For example, non-contiguous tensor slices may need to be transferred between GPUs, which is a common scenario in ML applications. We find that *input tensor management* (e.g., [21]–[23]) are the most common bugs in this category because the existing PyTorch implementation is not flexible enough for handling non-contiguous data as input. We also find that the CUDA driver is also not flexible enough to tackle such non-contiguous data. Thus, computation like backward or gradient computation using CUDA ends up crashing [24], [25] when processing non-contiguous data. Moreover, wrong convolution layer channel management, information loss, and performance degradation can be caused by non-contiguous data input [26], [27]. As a result, PyTorch and CUDA drivers cannot always assume that tensor elements in memory are contiguous. In fact, because nested tensors and distributed tensors are widely adopted for training large machine learning models, non-contiguous data becomes the norm. They need to provide the flexibility and capability of processing non-contiguous data in-situ without the manual process (e.g., code patching) of transforming data layout in memory explicitly by users or the library.

2. Segmentation fault (20.9%): There are five types of bugs that commonly lead to segmentation faults: *invalid memory access*, *input type checking*, *data type issues*, *programming implementation errors*, and *memory corruption*. `Invalid memory access` occurs when the callback functions registered for tensors improperly handle garbage collection in memory [28], [29]. `Programming` and `Input type checking` can also lead to segmentation fault [30], [31].

3. Indexing (14%): In PyTorch, *memory overflow* occurs due to mismanagement of indexing like padding in the convolution layer or creation of pointers [32], [33]. Moreover, when `data type` is misused and `invalid class` is initiated, the wrong index can also lead to memory bugs [34] [35].

4. Memory format (11.6%): Memory format tells the operator how to organize the output in memory, ensuring efficient access and computation. Convolution layer computation fails to correctly format the output on memory resulting in *Memory format* bugs [36]–[38]. Changing the weight tensor format to `channels_last` in `ConvTranspose2d` can corrupt output, leading to random or NaN values. Additionally, when using `channels_last` format for weights, convolutions with a contiguous input tensor may yield incorrect results if the input has only one channel.

5. Logical (4.7%): Wrong ordering of code implementation leads to logical bugs in memory [39].

6. Pin Memory (4.7%): In PyTorch, "pin memory" refers to the process of using pinned (or page-locked) memory for tensors on the CPU. This can be beneficial when transferring data from CPUs to GPUs because using the pinned memory can accelerate the data transfer process [40]. We found errors related to pin memory in `input tensor management` and `programming implementation` [41], [42].

7. Concurrency (4.7%): Concurrently accessing shared memory by multiple processes needs to assign proper locking on the memory regions. Memory bugs related to concurrency happen when there is any kind of `lock misses`.

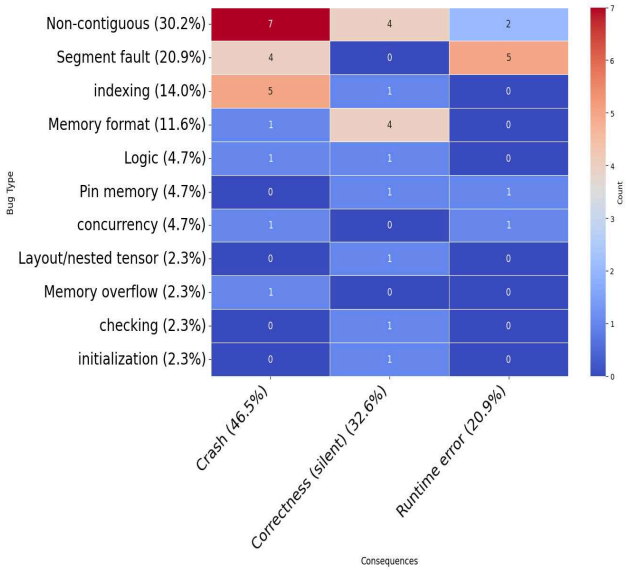

Fig. 6: Heatmap of memory bugs and its bug distribution regarding the consequences.

8. Other (9.2%): A Few other types of memory bugs (i.e., *memory overflow*, *checking*,*initialization*, *nested tensor layout*) also occurred leading to memory bugs.

C. What are the Consequences of Memory Bugs?

From the heatmap in Figure 6, we observe that crashes are the most frequent consequence (46.5%) across different memory bug types, followed by silent correctness issues (32.6%) and runtime errors (20.9%). The most prominent bug types, contributing to the most severe consequences, are non-contiguity (30.2%) and segment fault (20.9%), while indexing (14%) and memory layout (11.6%) bugs are also significant contributors.

Further, the type of non-contiguity bugs stands out prominently in the heatmap, contributing to 7 crashes, 4 silent correctness issues, and 2 runtime errors. These numbers underscore how non-contiguous memory handling can result in some of the most severe system breakdowns, such as crashes and silent correctness failures. The association with silent correctness is especially concerning because these bugs do not immediately cause visible system failure but can lead to incorrect results, making them more difficult to detect and resolve. This highlights the need for specialized techniques to handle non-contiguous memory to ensure that systems remain robust and accurate during execution.

Other significant memory bug types include segment fault, which accounts for 4 crashes and 5 runtime errors, demonstrating its association with severe system failures. Indexing bugs, while contributing to fewer crashes, are still associated with 5 runtime errors. Additionally, the memory layout bugs contribute to both crashes and silent correctness issues. Lastly,

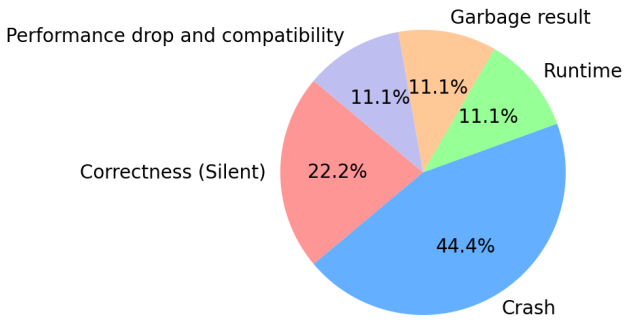

Fig. 7: Consequence distribution of Tensor memory layout bugs for Nested tensor and dtensor

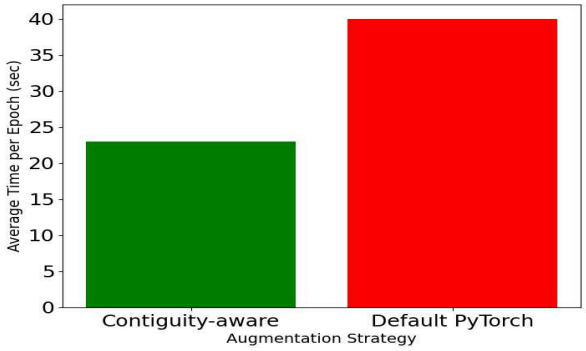

Fig. 8: The comparison of the augmentation time with tensor layout transformation using the layout-aware approach and the time with the default PyTorch approach

Concurrency issues are linked to both crashes and runtime errors, reflecting the complexity of managing parallel processes.

D. Non-Contiguity Bugs

We further analyzed 29 non-contiguity bugs. Among the identified issues, 15 were specifically tied to the implementation of memory contiguity in CUDA computations, highlighting a recurring problem that affects multiple aspects of the system. One example of such bugs is [43]. It arises because transposing the key tensor (`k`) within a self-attention module results in a non-contiguous memory layout for the nested tensor. During the backward pass, PyTorch expects the gradient tensor to be contiguous in memory. However, due to the transpose operation, the tensor becomes non-contiguous, leading to the `RuntimeError` error. The bug is challenging to detect for two reasons. First, the transposed non-contiguous tensor (`k`) can be produced after passing a contiguous tensor (`x`) through one or multiple linear layers. Second, since the nested tensors can contain the tensors with irregular structures, transposing the nested tensor is not straightforward.

Additionally, we examined various components such as torch serialization, nested tensor backward operations, device mesh, and sharding. Of particular concern were the challenges posed by maintaining tensor layout, especially during CUDA-based computations, and the resulting impact on gradient computation, backward propagation, tensor indexing, performance, and compatibility across different components.

In Figure 7, the distribution of consequences reveals that 44% of non-contiguity bugs caused crashes, indicating significant reliability concerns. 22% of issues cause incorrect output but are silent, meaning they do not immediately cause visible failures but may lead to subtle errors. 11% of the bugs pertain to runtime errors, garbage results, and performance drops or compatibility complexities, highlighting areas where operations either fail, produce incorrect results, or experience performance inefficiencies.

Summary: (1) Non-contiguity bugs account for 30% of memory bugs in PyTorch. The consequences of these bugs are system crashes, incorrect results, and runtime errors. (2) Memory bugs are hard to detect because they may cause incorrect outputs without system failures. We call them silent bugs. It is hard to detect such bugs using existing tools designed based on static and dynamic analysis. (3) PyTorch and CUDA drivers may assume that tensor elements are contiguous in memory for the convenience of programming. However, as distributed tensors and nest tensors are widely adopted, many components in PyTorch did not work correctly for non-contiguous tensors.

VI. Performance Study of Tensor Layout Transformation

After extensively studying the patches for 29 non-contiguity bugs, we found that 9 bugs (31%) were fixed by converting the non-contiguous layout to contiguous layout for tensors [44]–[52]. In some cases, the conversion was hard-coded into the PyTorch source code by the developers [44], [45], [49]. In other cases, users were asked to manually convert tensors to contiguous format and pass them to PyTorch [49], [52]. Additionally, there are bugs associated with the lack of support for non-contiguous tensors, which necessitate converting them to a contiguous format to function properly [53]. Therefore, it is crucial to study the performance characteristics of non-contiguous tensors, particularly focusing on the conversion process to contiguous tensors.

In this section, we conducted two experiments to study the performance of tensor layout conversion and its impact on the performance of data paths. All the experiments were conducted on a machine equipped with one Intel Xeon Silver 4208 processor featuring 8 cores and 16 threads, running at 2.10 GHz. It has 64 GB of RAM and two NVIDIA GeForce RTX 3090 GPUs, each with 24 GB of VRAM, supported by CUDA 12.3. The system operates on Ubuntu 22.04 LTS. The primary storage is a 916 GB SSD.

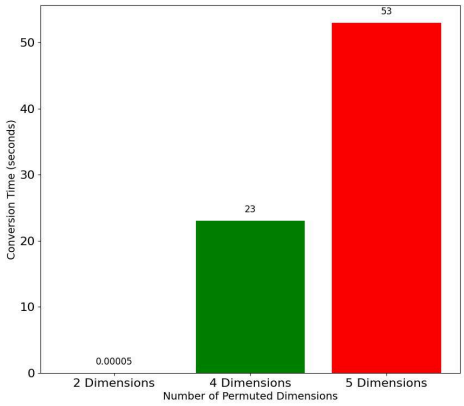

Fig. 9: Conversion time for non-contiguous tensors to contiguous tensors with varying numbers of transposed dimensions.

A. Performance Impact on Augmentation Operations

In the first experiment, we use tensor augmentation operations as benchmarks. The benchmark executes a sequence of augmentations including tensor conversion, crop, vertical flip, horizontal flip, rotation, color jitter, crop, and grayscale. It concludes with normalization. It executes these augmentation operations on 60,000 images repeatedly. We use CIFAR-10 datasets in the experiment. We compare the execution time of this benchmark between a contiguity-aware approach, which maintains tensor contiguity throughout the process after each augmentation operation, and the default PyTorch approach, which allows tensors to alternate between contiguous and non-contiguous states. Figure 8 shows that the execution times of the contiguity-aware approach and the default PyTorch are 23 sec and 40 sec, respectively. This indicates that maintaining a contiguous memory layout can reduce data augmentation time by 42.5% on average.

B. Performance Impact on Permutation Operations

In the second experiment, we use tensor permutation operations as benchmarks. The benchmark executes one permute() function and one contiguous() function in order on a 6-dimension tensor whose shape is (1, 101, 101, 101, 101, 101). In PyTorch, after permute(), a tensor will always be converted to a non-contiguous tensor in memory. After that, the contiguous() function will convert the output of permute() to a contiguous tensor. We measure the execution time of the benchmark given a different number of dimensions being permuted. Figure 9 shows the results, which show a significant increase in conversion time as the number of permuted dimensions increases. For instance, permuting only two dimensions resulted in a minimal conversion time of 0.00005 sec, whereas permuting four dimensions caused the conversion time to spike to 23 sec. This effect is even more pronounced when five dimensions are permuted, leading to a conversion time of 53 sec. These results underscore the challenges of working with non-contiguous tensors, particularly when operations demand contiguous memory layouts.

Our experimental results also show that even with the same number of dimensions, different permutation orders can lead to significantly different conversion times. This result underscores the complexity of tensor memory layout and suggests that the efficiency of tensor operations is influenced not only by the number of dimensions permuted but also by the specific order of these permutations.

Summary: The substantial performance drop observed with default PyTorch handling tensors highlights how deviations from an optimal memory layout can lead to significant inefficiencies. This finding emphasizes the need for memory layout awareness in tensor operations to ensure optimal performance and avoid unnecessary delays.

VII. Discussions and Suggestions

A. Tensor Contiguity Requirements

Assumptions about tensor contiguity. Developers often assume [54] that tensors are always contiguous during backward and gradient computations, which can result in silent correctness issues. Errors occur when users do not verify whether the input tensor is non-contiguous [55], [56], leading to unexpected outcomes.

Intermediate tensor requirements. The intermediate tensor received from upstream operations during backpropagation must maintain a contiguous memory layout to ensure efficient computation and correctness in gradient propagation [43]. Applying operations such as narrow(), view(), expand(), transpose(), and permute() [57]–[62] can alter a tensor's memory layout, potentially resulting in errors if users are unaware of these changes.

Suggestion: Ensure that all tensors involved in gradient and backward computations, including input and intermediate tensors, are in a contiguous memory layout to avoid silent correctness issues and ensure efficient computation.

B. Indexing Errors in Large Tensors

When a tensor has a very large number of elements, the data type for implementing element index in tensors may not have a sufficient range to index all elements correctly. This can lead to indexing errors, where indices exceed the allowable range, causing operations on the tensor to crash or produce silent, incorrect results. This issue is particularly problematic in scenarios where the tensor's size impacts the ability to access and manipulate its elements accurately [63]–[65].

Suggestion: (1) To prevent potential indexing errors and ensure compatibility with the declared ScalarType, all tensor indexing operations should use index data type (e.g., index_t). (2) It is crucial to implement checks or

assertions to verify that indexing operations do not exceed the bounds of the data type, especially in large tensor operations.

C. Programming Language and Hardware Constraints

Some programming languages need special attention when used with GPUs. For example, Fortran supports a feature called contained subroutines. However, when non-contiguous arrays are passed to Fortran subroutines, the runtime needs to create temporary contiguous arrays as a temporary data buffer. This leads to inefficiency and increased memory usage, especially in the CUDA systems [66]. Therefore, users should avoid non-contiguous pointer slices and contained subroutines to mitigate these issues.

A limitation in CUDA is that efficient data transfers require contiguous physical memory addresses, which is best achieved with pinned host memory [67]. Transfers from pageable memory involve extra DMA overhead and lower performance, particularly for small transfers. Managing transfer sizes and using pinned memory is crucial for optimizing performance [68]–[71].

Another limitation arises from compatibility issues where even identical graphics cards can expose different device numbers to host operating systems. For example, assume that we have three GPU cards. On one host, it may be visible as device 0, device 1, and device 3. On another host, it may be visible as device 0, device 2, and device 3. This discrepancy may lead to the failure of PyTorch [72]. Finally, CUDA involves incorrect bounds during device-to-host memory copies, which can cause undefined behavior, crashes, or incorrect results. Such problems often occurs when copying more data than allocated, incorrect strides for multi-dimensional arrays are used, and failure to account for padding [73]–[75] happens.

Suggestion: (1) To ensure effective CUDA programming, it is crucial to understand both software and hardware restrictions and compatibility with the device memory architecture. (2) The programmers need to make sure that copy sizes match the allocated memory to avoid errors. And (3) they need to employ proper APIs for handling various data layouts and utilize debugging tools like cuda-memcheck to identify and resolve issues.

VIII. Conclusion

In this paper, we present a comprehensive study of bugs related to memory management in PyTorch, examining changes from version 1.3.0 to the latest release over the past seven years. These patches and bugs reflect critical aspects and challenges within PyTorch's memory management. Our experiments highlighted the importance of memory-layout awareness in optimizing performance. Additionally, our detailed analysis of non-contiguity bugs provided insights into their root causes and the performance impacts of tensor contiguity. We anticipate that our findings will enhance the development of current and future PyTorch memory systems and improve bug detection and debugging tools. Furthermore, our study offers valuable perspectives for the advancement of fast computational kernels, such as CUDA, which are integral to high-performance computing tasks within the PyTorch ecosystem.

References

[1] "PyTorch," https://github.com/pytorch/pytorch.

[2] S. C. Yin Ho, V. Majdinasab, M. Islam, D. E. Costa, E. Shihab, F. Khomh, S. Nadi, and M. Raza, "An empirical study on bugs inside pytorch: A replication study," in 2023 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2023, pp. 220–231.

[3] J. Chen, Y. Liang, Q. Shen, J. Jiang, and S. Li, "Toward understanding deep learning framework bugs," 2023.

[4] T. Makkouk, D. J. Kim, and T.-H. P. Chen, "An empirical study on performance bugs in deep learning frameworks," in 2022 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2022, pp. 35–46.

[5] L. Jia, H. Zhong, X. Wang, L. Huang, and X. Lu, "The symptoms, causes, and repairs of bugs inside a deep learning library," Journal of Systems and Software, vol. 177, p. 110935, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121221000327

[6] Y. Yang, T. He, Z. Xia, and Y. Feng, "A comprehensive empirical study on bug characteristics of deep learning frameworks," Information and Software Technology, vol. 151, p. 107004, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584922001306

[7] M. M. Morovati, A. Nikanjam, F. Tambon, F. Khomh, and Z. M. J. Jiang, "Bug characterization in machine learning-based systems," Empirical Software Engineering, vol. 29, no. 1, p. 14, Dec 2023. [Online]. Available: https://doi.org/10.1007/s10664-023-10400-0

[8] X. Du, Y. Sui, Z. Liu, and J. Ai, "An empirical study of fault triggers in deep learning frameworks," IEEE Transactions on Dependable and Secure Computing, vol. 20, no. 4, pp. 2696–2712, 2023.

[9] F. Tambon, A. Nikanjam, L. An, F. Khomh, and G. Antoniol, "Silent bugs in deep learning frameworks: an empirical study of keras and tensorflow," Empirical Software Engineering, vol. 29, no. 1, p. 10, Nov 2023. [Online]. Available: https://doi.org/10.1007/s10664-023-10389-6

[10] Q. Guo, X. Xie, Y. Li, X. Zhang, Y. Liu, X. Li, and C. Shen, "Audee: automated testing for deep learning frameworks," in Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, ser. ASE '20. New York, NY, USA: Association for Computing Machinery, 2021, p. 486–498. [Online]. Available: https://doi.org/10.1145/3324884.3416571

[11] M. Li, J. Cao, Y. Tian, T. O. Li, M. Wen, and S.-C. Cheung, "Comet: Coverage-guided model generation for deep learning library testing," 2023.

[12] A. H. Phan, P. Tichavský, and A. Cichocki, "On fast computation of gradients for candecomp/parafac algorithms," 2012. [Online]. Available: https://arxiv.org/abs/1204.1586

[13] C.-H. Chu, K. S. Khorassani, Q. Zhou, H. Subramoni, and D. K. Panda, "Dynamic kernel fusion for bulk non-contiguous data transfer on gpu clusters," in 2020 IEEE International Conference on Cluster Computing (CLUSTER). IEEE, 2020, pp. 130–141.

[14] K. K. Suresh, K. S. Khorassani, C. C. Chen, B. Ramesh, M. Abduljabbar, A. Shafi, H. Subramoni, and D. K. Panda, "Network assisted non-contiguous transfers for gpu-aware mpi libraries," in 2022 IEEE Symposium on High-Performance Interconnects (HOTI). IEEE, 2022, pp. 13–20.

[15] D. K. Pal and M. Savvides, "Learning non-parametric invariances from data with permanent random connectomes," CoRR, vol. abs/1911.05266, 2019. [Online]. Available: http://arxiv.org/abs/1911.05266

[16] W. Wu, G. Bosilca, R. Vandevaart, S. Jeaugey, and J. Dongarra, "Gpu-aware non-contiguous data movement in open mpi," in Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing, 2016, pp. 231–242.

[17] M. Osama, S. D. Porumbescu, and J. D. Owens, "A programming model for gpu load balancing," in Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, 2023, pp. 79–91.

[18] D.-L. Lin and T.-W. Huang, "Efficient gpu computation using task graph parallelism." Springer, Cham, 2021, pp. 435–450.

[19] D. A. Matthews, "High-performance tensor contraction without transposition," 2016.

[20] C. Guo, R. Zhang, J. Xu, J. Leng, Z. Liu, Z. Huang, M. Guo, H. Wu, S. Zhao, J. Zhao, and K. 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, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 450–466. [Online]. Available: https://doi.org/10.1145/3620665.3640423

[21] "linalg.householder product is incorrect when given non-contiguous inputs 67513," https://github.com/pytorch/pytorch/issues/67513.

[22] "Incorrect gradient for masked select when inputs are non-contiguous ," https://github.com/pytorch/pytorch/issues/99638.

[23] "kthvalue incorrect with strided GPU tensor ," https://github.com/pytorch/pytorch/issues/45721.

[24] "Custom Autograd Functions Don't Work If Forward Pass Outputs a List of Tensors ," https://github.com/pytorch/pytorch/issues/87713.

[25] "Investigate from padded implementations correctness," https://github.com/pytorch/pytorch/issues/84082.

[26] "Investigate from padded implementations correctness," https://github.com/pytorch/pytorch/issues/84082.

[27] "Calling nested tensor.transpose(-1, -2) causes autograd error," ://github.com/pytorch/pytorch/issues/94303.

[28] "Calling saved tensors hooks. exit inside unpack hook can lead to segfault," https://github.com/pytorch/pytorch/issues/130734.

[29] "Segmentation fault when a Tensor backward hook removes itself," https://github.com/pytorch/pytorch/issues/58354.

[30] "Segmentation fault in dataloader after upgrading to pytorch v1.8.0 53894," https://github.com/pytorch/pytorch/issues/53894.

[31] "SSegfault on setting gradient value to instance of user-defined class 64813," https://github.com/pytorch/pytorch/issues/64813.

[32] "CUDA error: an illegal memory access was encountered when using output padding in nn.ConvTranspose3d 32866," https://github.com/pytorch/pytorch/issues/32866.

[33] "Illegal Memory Access was encountered in AvgPool2d CUDA kernel 84018," https://github.com/pytorch/pytorch/issues/84018.

[34] "Cross Entropy doesn't work with the specific batch, but works with each sample from this batch 108345," https://github.com/pytorch/pytorch/issues/108345.

[35] "Incorrect and inconsistent outputs from CrossEntropyLoss(reduction="none") with torch.float16 dtype 111484," https://github.com/pytorch/pytorch/issues/111484.

[36] "Wrong output of single-channel channels last convolution with channels first input 82060," hhttps://github.com/pytorch/pytorch/issues/82060.

[37] "[PT2.0] Channels last for weight for ConvTranpose gives Random output 99519," https://github.com/pytorch/pytorch/issues/99519.

[38] "CUDA native batch norm backward returns non-channels last grad for channels last input 107199," https://github.com/pytorch/pytorch/issues/107199.

[39] "Segment Fault after model inference all images using C++ API," https://github.com/pytorch/pytorch/issues/38385.

[40] "TORCH.UTILS.DATA," https://pytorch.org/docs/master/data.html.

[41] "NotImplementedError: Cannot access storage of SparseCsrTensorImpl 115330," https://github.com/pytorch/pytorch/issues/115330.

[42] "Calling pin memory() fails for nested tensor 102167," https://github.com/pytorch/pytorch/issues/102167.

[43] "contiguous non-contiguous tensors," https://github.com/pytorch/pytorch/issues/94303.

[44] "Incorrect gradient for masked select when inputs are non-contiguous 99638," https://github.com/pytorch/pytorch/issues/99638.

[45] "call contiguous on BMM inputs for NT on CUDA 88108," https://github.com/pytorch/pytorch/pull/88108.

[46] "[Breaking change 2.1] Passing non-contiguous inputs to SDPA on CUDA device with the mem-efficient attention backend returns garbage 112577," https://github.com/pytorch/pytorch/issues/112577.

[47] "Incorrect and inconsistent outputs from CrossEntropyLoss(reduction="none") with torch.float16 dtype 111484," https://github.com/pytorch/pytorch/issues/111484.

[48] "[CUDA] 64-bit indexing fixes for cross-entropy kernels," https://github.com/pytorch/pytorch/pull/112096.

[49] "sparse.mm produces incorrect derivatives 102493," https://github.com/pytorch/pytorch/issues/102493.

[50] "sparse.mm.backward: fix for non-contiguous grad values on CPU 106127," https://github.com/pytorch/pytorch/pull/106127.

[51] "[NestedTensor] Add a contiguous checks to get buffer 86496," https://github.com/pytorch/pytorch/pull/86496.

[52] "Custom Autograd Functions Don't Work If Forward Pass Outputs a List of Tensors 87713," https://github.com/pytorch/pytorch/issues/87713.

[53] "torch.flip not implemented for non-contiguous boolean tensors 52062," https://github.com/pytorch/pytorch/issues/52062.

[54] "sparse.mm.backward: fix for non-contiguous grad values on CPU," https://github.com/pytorch/pytorch/pull/106127.

[55] "[Breaking change 2.1] Passing non-contiguous inputs to SDPA on CUDA device with the mem-efficient attention backend returns garbage," https://github.com/pytorch/pytorch/issues/112577.

[56] "Pull Request #86496: Fix for contiguous tensor handling in PyTorch," https://github.com/pytorch/pytorch/pull/86496.

[57] "Contigious vs non-contigious tensor," https://discuss.pytorch.org/t/contigious-vs-non-contigious-tensor/30107/2.

[58] "Performance of contiguous vs. non-contiguous tensors," https://discuss.pytorch.org/t/performance-of-contiguous-vs-non-contiguous-tensors/107288.

[59] "Different between permute, transpose, view? Which should I use?" https://discuss.pytorch.org/t/different-between-permute-transpose-view-which-should-i-use/32916.

[60] "what makes a tensor have non-contiguous memory?" https://stackoverflow.com/questions/54095351/in-pytorch-what-makes-a-tensor-have-non-contiguous-memory.

[61] "What does .contiguous() do in PyTorch?" https://stackoverflow.com/questions/48915810/what-does-contiguous-do-in-pytorch.

[62] "What's the difference between 'reshape()' and 'view()' in PyTorch?" https://stackoverflow.com/questions/49643225/whats-the-difference-between-reshape-and-view-in-pytorch/49644300#49644300.

[63] "Incorrect and inconsistent outputs from CrossEntropyLoss(reduction="none") with torch.float16 dtype," https://github.com/pytorch/pytorch/issues/111484.

[64] "Cross Entropy doesn't work with the specific batch, but works with each sample from this batch," https://github.com/pytorch/pytorch/issues/108345.

[65] "BF16 Matmul not get same result on cuda and cpu," https://github.com/pytorch/pytorch/issues/111457.

[66] "OpenACC: cuStreamSynchronize crash when using pointers as parameters," https://forums.developer.nvidia.com/t/openacc-custreamsynchronize-crash-when-using-pointers-as-parameters/196944.

[67] M. Bauer, H. Cook, and B. Khailany, "Cudadma: optimizing gpu memory bandwidth via warp specialization," in Proceedings of 2011 international conference for high performance computing, networking, storage and analysis, 2011, pp. 1–11.

[68] "efficiency of copying a strided array," https://forums.developer.nvidia.com/t/efficiency-of-copying-a-strided-array/135518.

[69] "Efficient Host-Device Data Transfer," https://engineering.purdue.edu/~smidkiff/ece563/NVidiaGPUTeachingToolkit/Mod14DataXfer/Mod14DataXfer.pdf.

[70] "How to transfer massive data efficiently?" https://forums.developer.nvidia.com/t/how-to-transfer-massive-data-efficiently/37621.

[71] "Why is the transfer throughput low when transferring small size data from Host to Device (or Device to Host)?" https://forums.developer.nvidia.com/t/why-is-the-transfer-throughput-low-when-transferring-small-size-data-from-host-to-device-or-device-to-host/153962.

[72] "2 same Quadro P1000 cards, but only one can install Ubuntu." https://forums.developer.nvidia.com/t/2-same-quadro-p1000-cards-but-only-one-can-install-ubuntu/64612.

[73] "Memory read error when using csrmv with transpose operation," https://forums.developer.nvidia.com/t/memory-read-error-when-using-csrmv-with-transpose-operation/136019/8.

[74] "CUDA C++ Programming Guide," https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[75] " Memory Management," https://docs.nvidia.com/cuda/cuda-runtime-api/groupCUDARTMEMORY.html.