Mind the Gap: Revealing Inconsistencies Across Heterogeneous AI Accelerators

Elliott Wen*, Sean Ma*, Evan Tempero*, Bruce Sham*, Yousong Sun*, Hong Jia*, Daniel Luo†, Jiayi Hua†, Jens Dietrich‡, Kaiqi Zhao§ Jiaxing Shen¶
, *The University of Auckland, New Zealand
†Hong Kong Polytechnic University, Hong Kong
‡Victoria University of Wellington, New Zealand
§Harbin Institute of Technology, China
¶Lingnan University, Hong Kong
Email: elliott.wen, sean.ma, e.temper, bruce.sham, y.song, jia.hong@auckland.ac.nz,
daniel.luo, jiayi.hua@polyu.edu.hk,
jens.dietrich@vuw.ac.nz, zhaokaiqi@hit.edu.cn, jiaxingshen@ln.edu.hk

***Abstract*—While NVIDIA remains the dominant provider of AI accelerators within cloud data center, emerging vendors such as AMD, Intel, Mac, and Huawei offer cost-effective alternatives with claims of compatibility and performance. This paper presents the first empirical study investigating divergence in machine learning model across heterogeneous AI accelerators. Utilizing an automated pipeline, we synthesize over 100,000 variant models derived from 4,000 real-world models and execute them across five different enterprise-grade accelerators. Our findings suggest that newer AI platforms from Mac and Huawei support at least 17% fewer operators than NVIDIA. These platforms also exhibit a higher rate of output discrepancies (exceeding 5%), which stem from differences in operator implementations, handling of exceptional numerical values, and instruction scheduling. They are also more susceptible to failures during model compilation-based acceleration, and in some cases, the compiled models produce outputs that differ noticeably from those generated using the standard execution mode. In addition, we identify 7 implementation flaws in PyTorch and 40 platform-specific issues across vendors. These results underscore the challenges of achieving consistent machine learning behavior in an increasingly diverse hardware ecosystem.**

I. Introduction

Machine learning (ML) is a rapidly expanding field with growing influence across various disciplines. To meet the computational demands of ML models training and inference, many cloud service providers now offer access to hardware such as graphics processing units and specialized AI accelerators. Although NVIDIA-based solutions are most widely adopted, other platforms such as AMD, Intel, Mac and Huawei are gaining traction. They claim to offer sufficient computational performance for general machine learning tasks, while at a more competitive price point. In addition, they emphasize support for existing ML frameworks, particularly PyTorch, one of the most widely used frameworks in academia and industry [8].

As hardware platforms become increasingly diverse, concerns about their consistency and reliability are growing, particularly for developers seeking to avoid vendor lock-in and adopt more cost-effective cloud AI hardware services. In this paper, we present, to the best of our knowledge, the first empirical study assessing divergence in ML model execution across heterogeneous AI accelerator. We aim to answer the following research question: *How consistent are the outputs of ML models across different hardware platforms, and what are the potential sources of inconsistency?* To address this, we employ a differential testing approach; we execute 100,000 synthesized models with the same randomly generated inputs across various hardware platforms. We then investigate significant discrepancies to uncover potential issues in the underlying hardware or software stack.

To support this investigation, we implement an automated pipeline, as illustrated in Figure 1. The pipeline begins with corpus curation, where we crawl approximately 4,000 real-world models from various machine learning domains. These curated models are then converted into computational graphs, which serve as input to a model synthesizer. The synthesizer generates a large number of variant models by iteratively merging random subgraphs from the corpus and applying mutations to individual nodes, inspired by prior work [5]. The variant models are executed across different hardware platforms, and any output discrepancies are recorded. We then analyze these discrepancies to pinpoint potential sources of divergence.

We conduct our experiments on five enterprise-grade AI accelerator hardware commonly accessible through cloud, including the `NVIDIA H200`, `AMD MI300X`, `Intel Max 1100`, `Huawei Ascend 910B`, and `Mac M4 Pro`. We unveil the following key research findings:

1) Relatively new platforms, such as Mac and Huawei, exhibit lower model compatibility due to unsupported data types or unimplemented operations. Our analysis reveals that Mac and Huawei support 34% and 17% fewer operators than NVIDIA, respectively. As such, code modifications are often necessary. Most missing operators on Huawei and Mac involve quantized/sparse operations, attention, NLP embeddings, and network training.

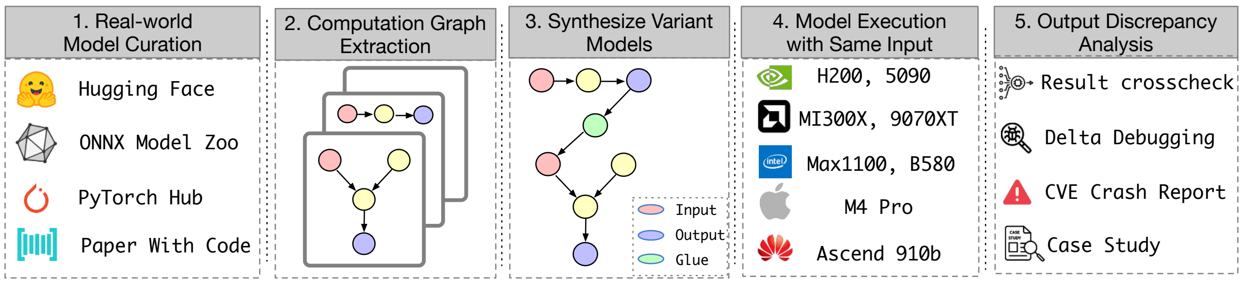
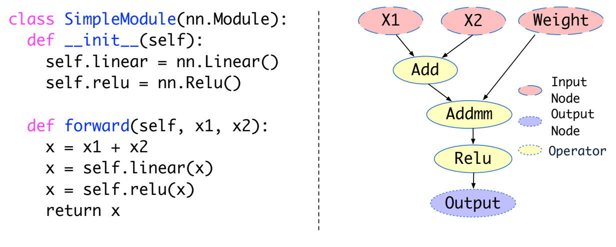


Fig. 1: Our pipeline to uncover the behavior inconsistency across different hardware platforms



```
class SimpleModule(nn.Module):
  def __init__(self):
    self.linear = nn.Linear()
    self.relu = nn.Relu()

  def forward(self, x1, x2):
    x = x1 + x2
    x = self.linear(x)
    x = self.relu(x)
    return x
```

(a) High-Level Python Code (b) Computation Graph

Fig. 2: PyTorch Model Representation: From High-Level Python Code to Computation Graph

2) Using NVIDIA as the baseline, Intel and AMD generally produce more consistent results, with output agreement rates of 99.6% and 99.8%, respectively. In contrast, the Ascend and Mac platforms show significantly lower consistency, with agreement rates dropping to 95% and 86%. These discrepancies primarily stem from flawed operator implementations, differences in handling exceptional values, variations in instruction scheduling, and instances of undefined behavior.
3) Some platforms exhibit reduced reliability when PyTorch's compilation-based acceleration features are enabled. On Mac and Huawei devices, approximately 2% of models fail to compile, and about 1.1% yield results that differ from those produced by the default execution mode.

II. Background and Related Work

We open-source our testing pipeline together with our model corpus. Using this pipeline, we uncover 7 implementation flaws in the PyTorch framework, such as memory access violations and program hangs caused by infinite optimization loops in the just-in-time compiler. We also identified behavioral inconsistencies or implementation flaws across different platforms: 1 on NVIDIA, 4 on AMD, 3 on Intel, 11 on Mac, and 13 on Huawei.

ML Model and Computation Graph: To aid reader comprehension of the rest of this paper, we first provide a brief overview of ML frameworks. We focus on PyTorch due to its broad adoption in both academic research and industry applications. Moreover, Pytorch is the only framework that consistently receives active development and optimization support from all five hardware vendors discussed in this paper.

In Pytorch, models are defined in Python by subclassing the `nn.Module` class. The model's layers are specified within the `init` method, while the forward computation logic is implemented in the `forward` method. Once defined, input data can be passed through the model to generate predictions. Figure 2(a) illustrates a simple model that takes two input tensors, $x1$ and $x2$, adds them together, and passes the result through a linear layer followed by a ReLU activation function. Under the hood, these high-level Python tensor operations are dispatched to PyTorch's low-level C++ backend, known as `ATen`. Each operation is mapped to a corresponding ATen operator. In our example, the tensor addition, linear transformation, and ReLU activation are translated into `aten::add`, `aten::addmm`, and `aten::relu`, respectively. These ATen operators form a directed acyclic computation graph as shown in Figure 2(b), where each operation corresponds to a node. An edge is created between two nodes if the output of one operation is used as the input to another. The input and output tensors of the graph can be treated as special types of nodes, representing the entry and exit points of the computation. Internally, ATen operators are further redispatched to hardware-specific implementations, such as CUDA kernels `relu_cuda` and `addmm_cuda` to leverage hardware acceleration.

Framework Bug Detection: ML frameworks can exhibit various types of bugs, such as incorrect operator implementations or memory safety violations [12]. Existing approaches for bug detection typically involve feeding data into curated or synthesized models to identify framework-level errors or inconsistencies. For instance, CRADLE [13] targets cross-framework inconsistencies by comparing outputs from a small set of real-world models across multiple frameworks. Rather than relying on real-world models, subsequent works such as AUDEE [7] and Luo et al. [11] generate models and inputs using genetic algorithms and graph-theoretic techniques. Building on this direction, LEMON [1], Muffin [6], and EAGLE [15] introduce various model mutation strategies to expand the pool of test models. NNSmith [10] proposes advanced abstract operator modeling to further enhance model diversity.

More recently, researchers have begun exploring differential testing at the operator level instead of the model level. For example, FreeFuzz [17], SkipFuzz [9] and IvySyn [2] apply random mutations, such as changes to data types and values, to individual operator parameters to uncover potential bugs. DeepREL [3] and TENSORSCOPE [4] identify related APIs within a library and corresponding APIs across different libraries. They then leverage test inputs from other APIs to generate inputs for the target APIs. DocTer [18] and ACETest [14] extract and analyze input constraints from API documentation and C++ implementation to generate more valid and targeted parameter inputs.

Our testing pipeline draw inspiration from prior research and introduces a practical extension by combining graph synthesizing techniques with operator-level input variation methods to explore a broader diversity of execution behaviors. **Our Contributions:** Unlike prior work that primarily investigates software-level bugs and inconsistencies within ML frameworks, our study explores a complementary dimension: behavioral divergence across heterogeneous AI accelerators. Our findings on platform compatibility and reliability have practical implications for developers aiming to avoid vendor lock-in and adopt more cost-effective hardware alternatives.

III. Pipeline Implementation

In this section, we provide implementation details for our testing pipeline.

A. Dataset Curation

To synthesize variant models for testing, we first require a model corpus. We construct this corpus from a large and diverse collection of real-world machine learning models from multiple domains.

We implement a web crawler to collect models from popular machine learning distribution platforms such as Hugging Face[1] and TorchHub[2]. These platforms offer convenient access to a wide range of pre-trained models in various domains and enable deployment with minimal Python code. Our crawler successfully collected approximately 170,000 PyTorch models from 50 tasks domains along with their associated metadata. While the total number of models is substantial, many share the same underlying network architecture. The primary differences often lie in their learned weights, as they are just fine-tuned on different datasets or downstream tasks. To eliminate duplicate architectures, we examine the accompanying `config.json` files, which specify key architectural parameters such as the model class name and the number of layers. We also analyze each model's architecture-specific base class (e.g., `BertModel` or `GPT2Model`) to assess structural equivalence. Eventually, we retain a final set of 3,711 models with unique architectures.

In addition, we manually curate models from the *Papers with Code* platform[3], which connects ML publications with their corresponding code implementations and datasets. Unlike the previously mentioned sources, Papers with Code is academically oriented and exhibits greater variation in code structure and optimization techniques. We retrieve the platform's database as of May 2025 and identify 71,232 PyTorch models across 45 categories. From these, we apply random sampling to select a total of 300 repositories.

[1]https://huggingface.co/

[2]https://pytorch.org/hub/

[3]https://github.com/paperswithcode/paperswithcode-data

B. Computation Graph Extraction

The next step is capture the computational graph of each model in the corpus. We achieve this by instrumenting the Pytorch's ATen dispatcher functions, such as `kernel.call` and `kernel.callBoxed`, to capture detailed operator invocation records. For each invocation, we record its input parameters and return values. Most parameters are simple constants, such as scalar numbers or boolean values. If a parameter is a tensor, we additionally record its identifier along with metadata such as shape, data type, and storage device. The recorded traces reveal how tensors are produced and consumed across operator invocations. Using this information, we can reconstruct the corresponding computational graph through a straightforward one-to-one mapping. This approach records computation graph transparently without altering the original model and supports both forward and backward computation graph. In contrast, PyTorch's in-built graph tracing frameworks, such as GraphFX or JIT tracing, require modifying model code and do not capture backward computation graph.

C. Variant Model Generation

We adopt the graph merging technique introduced in GraphFuzz [5] to generate variant models. This method has demonstrated high network structural diversity and code coverage. Algorithm 1 outlines the procedure. The process begins with an empty graph G, and iteratively integrates subgraphs extracted from our computational graph corpus until a predefined node threshold T is reached. As T increases, the architectural diversity expands, which also leads to increased model execution and compilation time. Based on extensive preliminary experiments, we set $T = 1{,}000$ nodes to achieve a trade-off between them.

A key implementation detail involves resolving mismatches in tensor dimensions between the output nodes of the existing graph and the input nodes of the new components. This is achieved by inserting additional glue nodes. If the output tensor contains more elements than required, we first apply the `aten::flatten` operator to convert the output tensor into a one-dimensional form, followed by `aten::slice` to remove the excess elements, and finally use `aten::reshape` to match the desired input shape. Conversely, if the output tensor contains fewer elements than needed, we employ the `aten::pad` operator to insert additional constants before reshaping the tensor.

To further enhance the diversity of our models, we also employ a well-tested operator mutation strategy from prior work [17]. Specifically, with a predefined probability (e.g., 0.25), we replace an operator with a syntactically similar alternative. For instance, an out-of-place operator such as `aten::add` may be substituted with its in-place counterpart `aten::add_`, which stores the result directly in the input tensor rather than creating a new one. Activation functions

Algorithm 1 Variant Model Generation Workflow

```
 1: Initialize an empty graph $G$
 2: while number of nodes in $G \leq$ threshold $T$ do
 3:     Sample a computational graph $C$ from the corpus
 4:     Extract a random connected subgraph $S$ from $C$
 5:     for all incoming edges in $S$ without a predecessor node do
 6:         Insert a new input node as its predecessor
 7:     end for
 8:     for all outgoing edges in $S$ without a subsequent node do
 9:         Insert a new output node as its successor
10:     end for
11:     Merge $S$ into $G$ by randomly connecting output nodes of $G$ to input nodes of $S$
12: end while
```

can also be replaced with alternatives, such as substituting `aten::relu` with `aten::hardtanh`. Additionally, functionally equivalent operations with different formulations may be exchanged, for example, replacing `aten::add` with `aten::addcdiv` with appropriate scaling and divisor tensors.

D. Model Execution

We evaluate each synthesized model across three execution modes in PyTorch.

1) **Default Mode (i.e., Eager Mode)**: In this mode, computation operations are executed immediately as they are encountered in the Python code (i.e., without optimization). Eager mode is the most widely used execution mode in PyTorch. Our analysis of all GitHub repositories listed on the *Papers with Code* platform reveals that approximately 97% of them solely use Eager mode. To execute in this mode, we first deserialize the synthesized graphs using the `torch.jit.load` API, which reconstructs the corresponding high-level Python statements. These statements are then executed directly using the `eval` API.
2) **Just-In-Time (JIT) Compilation**: In JIT mode, the model initially runs in eager mode for several iterations to gather runtime information such as input tensor shapes and data types. This collected information is then used to transform the graph through a series of simple-yet-effective optimization passes, such as constant folding and dead code elimination, to enhance execution efficiency. We can enable JIT profiling and compilation using the `torch._C._jit_set_profiling_mode` API.
3) **TorchDynamo Compilation**: PyTorch recently introduces a new execution mode called *TorchDynamo*. In this mode, PyTorch hands off the computation graph to vendor-specific backends (e.g., IPEX for Intel) that compile it into highly optimized machine code. To run in this mode, we pass the deserialized Python module directly to the `torch.compile` API.

We generate a set of random inputs to evaluate each model. We adjust the value ranges according to the required input types as guided by prior work [16]. For example, floating-point tensors (e.g., `torch.double`) are populated with values sampled uniformly from the range $[0, 1)$. Special consideration is given to the `torch.int64` data type, which is frequently used as an index parameter in many aten operators. Excessively large indices can easily lead to out-of-bounds accesses. To suppress the error, we adopt a heuristic range of $[0, 4]$, informed by our preliminary experiments. Following previous work [18], we also adopt a set of input constraints to ensure semantic correctness; for example, the offset array in `aten::embedding_bags` must begin with 0. We also disable the autocast feature to ensure all the test cases to run in the same precision mode.

E. Output Comparison and Analysis Discrepancy

Following prior work [15], we adopt the inconsistency detection formula in the TensorFlow and PyTorch test suites. Specifically, two outputs O_{p1} and O_{p2} from different platforms are considered equivalent between outputs if the condition $|O_{p1} - O_{p2}| \leq \text{atol} + \text{rtol} \cdot |O_{p2}|$ holds element-wise, where $\text{atol} = 5 \times 10^{-4}$ is the absolute tolerance and $\text{rtol} = 1 \times 10^{-4}$ is the relative tolerance. Both thresholds are used jointly to account for absolute and relative differences in the outputs.

Once output discrepancies are detected across platforms, we aim to trace them back to the specific locations in the model that triggered the differences to facilitate manual analysis. To achieve this, we construct a data flow graph of the model to capture the dependency chain between output tensors and the operations that produce them. Beginning with the earliest operator in the dependency chain, we perform a layer-by-layer comparison of the outputs generated on each platform. This iterative process continues until we isolate the first operator whose output diverges. As part of this process, we adopt the comparison metrics from CRADLE [13]. Specifically, we compute the Mean Absolute Distance (MAD) between each pair of corresponding outputs in the dependency chain across platforms. We then calculate the rate of change for each output and cluster inconsistencies based on the operator that exhibits the highest rate of change.

IV. Evaluation

We conduct our evaluations on a bare-metal machine equipped with a 240-core 5th Gen Intel Xeon Platinum 8580 processor and 1.97 TiB of DDR5 memory. It is installed with four enterprise-grade AI accelerator hardware platforms, including the `NVIDIA H200`, `AMD MI300X`, `Intel MAX 1100`, and `Huawei Ascend 910B`. To include the Mac platform in our study, we additionally use a `Mac M4` Pro desktop machine. We use the NVIDIA H200 as our baseline, given its widespread deployment in AI production environments.

A. Discrepency in Model Execution across Platforms

In our experiment, we synthesized 100,000 variant models from our corpus. In our setup, the model generation process takes approximately 17 minutes. The execution times

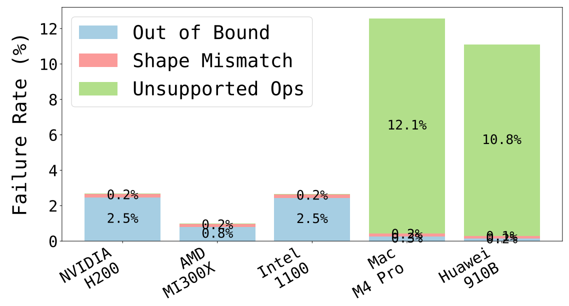


Fig. 3: Model Execution Failures in Default Execution Mode

are approximately 289 hours on the `NVIDIA H200`, 297 hours on the `Intel MAX 1100`, 320 hours on the `AMD MI300X`, 277 hours on the `Mac M4 Pro`, and 412 hours on the `Huawei Ascend 910B`. Huawei exhibits slightly longer execution times, likely due to its unoptimized PyTorch extension. Specifically, it introduces an initialization overhead (about 15 seconds in our setup) each run to query the available hardware operators.

We record the execution behavior of each model in default execution mode and visualize the number of failures and their root causes across different platforms in Figure 3. Each model is evaluated using the latest version of its corresponding runtime environment.

Baseline: For our baseline NVIDIA platform, the most common type of failure is out-of-bounds errors, which are expected given that the models are tested with randomly generated inputs, including index tensors. Although these index tensors are generated within a restricted range, out-of-bounds accesses may still occur, for example, when the tensor being indexed is also small. The other type of error is shape mismatch. These errors arise because certain operators, such as `aten::where` or `aten::select`, determine the shape of their output tensors based on the values of the input tensors. Due to input randomness, the resulting shapes of intermediate tensors may differ from the shapes expected by subsequent operations, thus leading to execution failures. Nevertheless, these two categories of errors together account for only about 2.5% of total model executions on our baseline platform. Following prior work [17], we also examine the successfully executed models and analyze the types of operators they contain. Our analysis reveals that these models cover 397 out of the 488 operators available in PyTorch 2.7.1. This indicates that our synthesized models achieve reasonable operator coverage.

Unsupported Operations on Huawei and Mac: Another notable phenomenon is that the Mac and Huawei platforms exhibit a significant number of execution failures due to unsupported operation errors. They account for more than 10% of all models. Our pipeline detects 47 unimplemented operators on Mac and 34 on Huawei. It is important to note that these numbers likely underestimate the count of unsupported operators, as some operators silently fall back to CPU implementations without warning. Such fallbacks often lead to significant performance degradation.

To gain a deeper understanding on the unimplemented operators, Specifically, we examine the code graph to identify operators that ultimately invoke hardware command interfaces (e.g., `EXEC_NPU_CMD` on Huawei). This indicates that these operators can benefit from hardware acceleration. We observe that our NVIDIA baseline provides hardware acceleration for 488 operators, with the AMD platform offering an identical level of support. Intel supports slightly fewer operators, totaling 449. Huawei's platform falls further behind and supports 407 operators, which is 17.5 percent fewer than NVIDIA. The Mac platform offers the least support, with hardware acceleration available for just 332 operators. This is 34.0 percent fewer than NVIDIA. The missing operators on these platforms fall into the following categories:

1. Quantized Inference: Huawei and Mac offer very limited support for quantized inference operators, such as `aten::_weight_int4pack_mm`. These operators enable matrix multiplication using compressed data formats to reduce memory consumption and bandwidth usage.

2. Sparse Operations: Both platforms lack support for sparse tensor computation like `aten::_sparse_semi_structured_linear`. These routines exploit structural sparsity in model weights to decrease memory usage and computation time by skipping zero elements.

3. Attention Mechanism: Huawei and Mac do not provide specialized attention kernels, such as `aten::flash_attention`. These kernels fuse multiple steps of the attention computation into a single optimized operation for higher computation speed.

4. NLP Embeddings: Both platforms lack support for embedding operations like `aten::_embedding_bag_backward`. They are crucial for computing gradients in embedding bag layers commonly found in NLP models.

5. Fused Training Support: Fused implementations of the Adam optimizer update (e.g., `aten::fused_adam`) and dropout steps (e.g., `aten::fused_dropout`) are missing on Huawei and Mac, which potentially degrades the training efficiency.

6. Advanced Linear Algebra: Huawei and Mac lack support for advanced linear algebra routines, such as `aten::triangular_solve` and `aten::inalg_matrix_exp`, which are essential in scientific computing, control theory, and differentiable physics.

What makes matters worse is that most operators on the Mac and Huawei platforms cannot handle 64-bit floating-point input tensors. Additionally, some operators on the Huawei platform do not support 64-bit integers as the data type for index tensors. The Mac backend consistently raises a runtime error when encountering a 64-bit floating-point value. In contrast, the behavior on the Huawei Ascend platform is less predictable. For certain operators (e.g., `aten::to`), the data may be silently cast to a lower-precision type, while some operators (e.g., `aten::huber_loss_backward`) trigger a runtime error. We also observed that operators on Mac and Huawei platforms sometimes require input tensors to be in a contiguous memory format. Otherwise, an exception may be raised.

Missing Bounds Checking on AMD: Another interesting observation is that AMD platforms report significantly fewer out-of-bounds (OOB) errors compared to the baseline and Intel platforms (approximately 1,700 fewer). Upon examining the models that trigger OOB on AMD, we find that all of them are reported by the baseline and Intel platforms. Conversely, many additional models flagged as OOB by the baseline execute without error on AMD and produce outputs that may appear correct to a human observer. Moreover, OOB errors on the baseline and Intel platforms typically raise runtime exceptions. These exceptions include clear stack traces, which make it easier to identify the offending operators. In contrast, the AMD platform often returns a generic error message such as 'memory access violation' and crashes the PyTorch program. This provides little diagnostic insight to users.

Finding 1: Mac and Huawei platforms lack support for certain operators and data types. As a result, developers may need to refactor their codebases, potentially at the cost of reduced performance and precision.

Finding 2: Most missing operators on Huawei and Mac involve quantized/sparse operations, attention, NLP embeddings, and Adam training, indicating reduced efficiency for common deep learning workloads.

Finding 3: AMD lacks a robust bounds-checking mechanism, which hinders the early detection of model bugs and introduces potential security vulnerabilities.

B. *Inconsistency in Model Outputs and Case Studies*

We compare model outputs across different hardware platforms. We use only 87,840 models from the previous section that run successfully on all the platforms. Figure 4 illustrates the observed result discrepancies. Notably, AMD demonstrates the lowest deviation from the baseline, affecting only 0.2% of the evaluated models. Intel shows a slightly higher discrepancy rate at 0.4%. Ascend exhibits a moderate deviation and impacts 5.1% of the models, while Mac has the highest discrepancy rate and affects 14.1% of the models.

To better understand the nature of these inconsistencies, we performed a manual analysis on a sample of inconsistent cases and present a case study. We categorize the cases as follows.

1) Operator Implementation: Output inconsistencies may stem from flawed operator implementations. We identified 13 faulty operators on the Huawei platform that produce incorrect results, data types, or tensor shapes. For example, the operator `max_pool1d_with_indices`, when provided with a 64-bit integer index tensor, unexpectedly returns a tensor with elements of an unsigned 8-bit data type. The operator `grid_sampler_2d_backward` appears to produce incorrect outputs in the zero padding mode, whereas all other platforms yield consistent results. Additionally, the operator `prelu_kernel_backward` is expected to return two tensors with matching dimensions; however, this condition does not hold on the Huawei platform. Most of the identified operators have the suffix 'backward', further suggesting that the Huawei platform may be less suitable for training tasks. We also observe 11 faulty operators on the Mac platform. For instance, the widely used operator `aten::pad`, which adds extra elements around the edges of a tensor, may fail to function correctly; when the padding size exceeds a threshold (e.g., 65,536 for a 1D tensor), the operator begins to overwrite existing elements with corrupted data. This issue is particularly concerning given the operator's widespread usage.

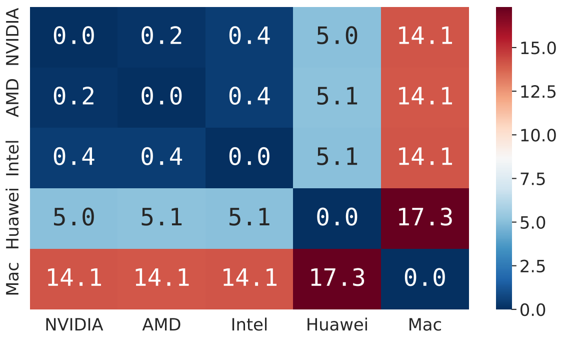


Fig. 4: Model Output Consistency Across Platforms

2) Exceptional Number Handling and Propagation: Operators on different platforms may handle exceptional numerical values differently. For instance, when encountering numerically unstable inputs, the operators `aten::remainder` and `aten::convolution` produce `Inf` on AMD and Intel, whereas on NVIDIA they return `NaN`. We also observe variations in how exceptional values (e.g., `Inf` and `NaN`) propagate through the computation graph. Ideally, such values should be consistently propagated across operators to aid developers in identifying numerical issues. However, AMD and Mac do not always follow this practice. For example, on AMD, the `aten::batch_norm` operator appears to replace `NaN` elements with interpolated values from nearby data. Mac is even less consistent: common operators like `aten::reshape` have been observed to silently convert `NaN` values into zeros.

3) Instruction Scheduling: Some operators can exhibit nondeterministic behavior, in other words, even with identical inputs, their outputs may vary slightly across different platforms or runs. This variability often stems from the parallel execution of these operators; inconsistencies can arise from how the compiler or runtime system schedules hardware instructions. For example, we observe that the operator `aten::max_unpool2d`, when given an index tensor containing repeated elements, behaves deterministically on NVIDIA hardware. Our 10,000 repeated runs all produce identical outputs. In contrast, on AMD hardware, the same operator exhibits nondeterministic behavior under the same conditions. The 10,000 runs yields three different outputs, each occurring at varying ratios

4) Undefined Behavior: Different computing platforms employ distinct strategies for handling undefined operations. One notable example is positive integer division by zero. NVIDIA returns `4,294,967,295`, the maximum value of a 32-bit

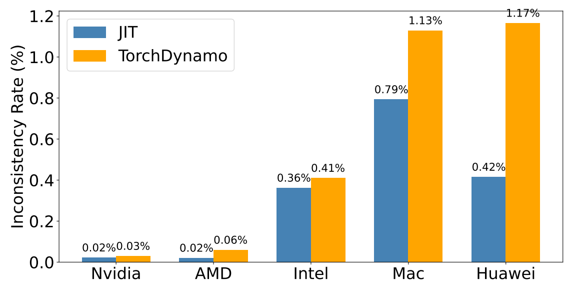


Fig. 5: Result Consistency Between Compilation Mode and Eager Mode

unsigned integer. On macOS, the result is `0`, while AMD interestingly returns the original value incremented by one. Another example involves converting floating-point infinity to a 64-bit integer. On NVIDIA, this conversion yields the largest possible 64-bit integer, whereas AMD returns `-4,294,967,296`.

> Finding 4: Intel and AMD generally produce results more consistent with the baseline than Mac and Huawei.

> Finding 5: Platform inconsistency is mainly attributed to flawed operator implementations, exceptional value handling, instruction scheduling, and undefined behavior.

C. Analysis of Model Compilation Issues

We re-execute the models discussed in the previous section using PyTorch's compilation modes, namely JIT and TorchDynamo. This evaluation includes only $77,291$ models that yield consistent outputs across all platforms under the default execution mode. We expect the output of the compiled models to remain consistent with that produced by the default execution mode, aside from very minor numerical discrepancies. Any substantial deviation may indicate an unreliable implementation within the vendor-specific compiler.

Figure 5 presents the count of inconsistencies between the results obtained using the two compilation modes and those produced by the default execution mode. Overall, JIT compilation appears to be more reliable than TorchDynamo, likely due to its simpler optimization passes. Among all platforms, NVIDIA and AMD demonstrate the fewest inconsistencies. We manually examine these cases and find that the inconsistencies primarily involve transitions between `NaN` and `Inf`. This may be attributed to the reordering of numerically unstable operations. Mac and Huawei exhibit the highest levels of inconsistency, approximately 1.1%. Our analysis indicates that most discrepancies involve significant differences in numerical values. This suggests potential implementation errors within the vendor-specific compiler pipeline.

We encountered a number of failures during the compilation process and a summary of the findings is provided in Table I. One key observation is that platforms generally exhibit better compatibility (i.e., fewer failures) with PyTorch's JIT compilation mode compared to TorchDynamo. In addition, compatibility also varies across platforms, with some experiencing significantly higher rates of compilation failures, for example, 1.9% on Huawei compared to 0.1% on the Baseline. We manually analyze these issues and categorize them as follows:

1) Stalled Compilation: We observed a JIT compiler hanging issue affecting almost every platforms. Our analysis attributes this behavior to a flaw in PyTorch's JIT optimization passes. Specifically, one pass attempts to simplify if/else predicates statements to reduce control flow, In some cases, later passes undo earlier simplifications, causing a loop that prevents the compiler making any progress. The issue appears less severe on Intel platforms, possibly because Intel provides custom optimization passes that avoid some of this problematic behavior. In TorchDynamo mode, we also observed several instances of the compiler hanging, but only on the Huawei platform. Our preliminary analysis suggests that this may also be caused by flawed optimization passes.

2) Unsupported Ops: TorchDynamo compilation encounters failures due to unsupported operations. In our evaluation, the most robust platform support was observed on NVIDIA and AMD GPUs, where all 22 failures were attributed to a small subset of operators lacking support for complex data types. In contrast, Intel platforms exhibited significantly more issues, with 172 model failures primarily caused by the lack of support for 64-bit floating-point data types in certain operators. Mac and Huawei platforms reported approximately 80 times more unsupported operation errors than NVIDIA, suggesting that their compiler infrastructures are still maturing.

3) Heap Corruption: We observed crash instances across nearly all platforms. Manual analysis of the stack traces revealed that these crashes were primarily caused by heap corruption and segmentation faults. We identified that two of the issues originate from the PyTorch's JIT optimization pass implementation, which lacks proper bounds checking. The remaining issues stem from vendor-specific compilers. Most platforms reported fewer than 10 crash instances, whereas Huawei platforms experienced up to ten times as many. This discrepancy may be attributed to less mature compiler implementations on the latter.

4) Code Generation Error: In TorchDynamo mode, the computation graph is first converted into a C++ function and then compiled by vendor-specific backends. NVIDIA uses NVFuser for this purpose. AMD aims to maintain compatibility with CUDA and thus also adopts NVFuser. On these two platforms, we encountered three failures during the code translation process, specifically: `error: duplicate parameter name`. When we examine the generated C++ code, we found that the resulting function contained multiple parameters with the same name. This leads to invalid C++ function prototypes and causes compilation failures.

> Finding 6: Model compilation can reduce output reliability, especially on Mac and Huawei platforms. When compilation is desired to improve performance, JIT mode is recommended over TorchDynamo due to higher reliability.

TABLE I: Summary of model compilation issues across platforms

Issue Type	**Baseline**	**AMD**	**Intel**	**Mac**	**Huawei**
JIT					
Stalled Compilation	9	9	3	7	12
Heap Corruption	2	3	7	3	27
Total	**11 (<0.1%)**	**12 (<0.1%)**	**10 (<0.1%)**	**10 (<0.1%)**	**39 (<0.1%)**
TorchDynamo					
Stalled Compilation	0	0	0	0	16
Unsupported Ops	22	22	79	1719	1484
Heap Corruption	4	3	8	3	45
Codegen Error	3	3	0	0	0
Total	**29 (<0.1%)**	**28 (<0.1%)**	**87 (0.1%)**	**1722 (2.2%)**	**1529 (1.9%)**

Finding 7: Platform compatibility with PyTorch's compilation modes varies considerably. Huawei and macOS show notably higher rates of compilation failures compared to other platforms.

D. Threats to Validity

Our current evaluation focuses on single-device execution. We plan to extend our analysis to distributed settings, where multiple AI accelerator cards operate in parallel. This may reveal additional inconsistencies arising from the complexities of hardware communication and synchronization. Additionally, our experiments are conducted using the latest available versions of runtime environments, compilers, and drivers. While this ensures relevance to current deployments, it may overlook issues present in older or less frequently updated systems. Evaluating a broader range of software versions could reveal version-specific bugs and provide a more comprehensive view of platform stability. Finally, our study focuses solely on the latest generation of hardware. Including consumer-grade or earlier generations could reveal hardware-specific behaviors and inconsistencies even within the same vendor. Additional research efforts are warranted to investigate this further.

V. Conclusion

In this paper, we explore the consistency of machine learning model outputs across five major AI accelerator platforms in cloud data centers. Our findings indicate that newer AI platforms from Mac and Huawei support at least 17% fewer operators compared to NVIDIA, particularly those essential for training. These platforms also show a higher rate of output discrepancies (over 5%), which can be attributed to differences in operator implementations, the handling of exceptional numerical values, and instruction scheduling. In addition, they are more prone to failures during model compilation based acceleration. These results highlight the difficulty of ensuring consistent machine learning behavior across an increasingly diverse range of hardware platforms.

References

[1] AlZamily, J.Y., Naser, S.S.A.: Lemon classification using deep learning (2020)

[2] Christou, N., Jin, D., Atlidakis, V., Ray, B., Kemerlis, V.P.: {IvySyn}: Automated vulnerability discovery in deep learning frameworks. In: 32nd USENIX Security Symposium (USENIX Security 23). pp. 2383–2400 (2023)

[3] Deng, Y., Yang, C., Wei, A., Zhang, L.: Fuzzing deep-learning libraries via automated relational api inference. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 44–56 (2022)

[4] Deng, Z., Meng, G., Chen, K., Liu, T., Xiang, L., Chen, C.: Differential testing of cross deep learning framework {APIs}: Revealing inconsistencies and vulnerabilities. In: 32nd USENIX Security Symposium (USENIX Security 23). pp. 7393–7410 (2023)

[5] Green, H., Avgerinos, T.: Graphfuzz: Library api fuzzing with lifetime-aware dataflow graphs. In: Proceedings of the 44th International Conference on Software Engineering. pp. 1070–1081 (2022)

[6] Gu, J., Luo, X., Zhou, Y., Wang, X.: Muffin: Testing deep learning libraries via neural architecture fuzzing. In: Proceedings of the 44th International Conference on Software Engineering. pp. 1418–1430 (2022)

[7] Guo, Q., Xie, X., Li, Y., Zhang, X., Liu, Y., Li, X., Shen, C.: Audee: Automated testing for deep learning frameworks. In: Proceedings of the 35th IEEE/ACM international conference on automated software engineering. pp. 486–498 (2020)

[8] Imambi, S., Prakash, K.B., Kanagachidambaresan, G.: Pytorch. In: Programming with TensorFlow: solution for edge computing applications, pp. 87–104. Springer (2021)

[9] Kang, H.J., Rattanukul, P., Haryono, S.A., Nguyen, T.G., Ragkhitwetsagul, C., Pasareanu, C., Lo, D.: Skipfuzz: Active learning-based input selection for fuzzing deep learning libraries. arXiv preprint arXiv:2212.04038 (2022)

[10] Liu, J., Lin, J., Ruffy, F., Tan, C., Li, J., Panda, A., Zhang, L.: Nnsmith: Generating diverse and valid test cases for deep learning compilers. In: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2. pp. 530–543 (2023)

[11] Luo, W., Chai, D., Ruan, X., Wang, J., Fang, C., Chen, Z.: Graph-based fuzz testing for deep learning inference engines. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). pp. 288–299. IEEE (2021)

[12] Morovati, M.M., Nikanjam, A., Tambon, F., Khomh, F., Jiang, Z.M.: Bug characterization in machine learning-based systems. Empirical Software Engineering **29**(1), 14 (2024)

[13] Pham, H.V., Lutellier, T., Qi, W., Tan, L.: Cradle: cross-backend validation to detect and localize bugs in deep learning libraries. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). pp. 1027–1038. IEEE (2019)

[14] Shi, J., Xiao, Y., Li, Y., Li, Y., Yu, D., Yu, C., Su, H., Chen, Y., Huo, W.: Acetest: Automated constraint extraction for testing deep learning operators. In: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 690–702 (2023)

[15] Wang, J., Lutellier, T., Qian, S., Pham, H.V., Tan, L.: Eagle: creating equivalent graphs to test deep learning libraries. In: Proceedings of the 44th International Conference on Software Engineering. pp. 798–810 (2022)

[16] Wang, J., Pham, H.V., Li, Q., Tan, L., Guo, Y., Aziz, A., Meijer, E.: D 3: Differential testing of distributed deep learning with model generation. IEEE Transactions on Software Engineering (2024)

[17] Wei, A., Deng, Y., Yang, C., Zhang, L.: Free lunch for testing: Fuzzing deep-learning libraries from open source. In: Proceedings of the 44th International Conference on Software Engineering. pp. 995–1007 (2022)

[18] Xie, D., Li, Y., Kim, M., Pham, H.V., Tan, L., Zhang, X., Godfrey, M.W.: Docter: Documentation-guided fuzzing for testing deep learning api functions. In: Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis. pp. 176–188 (2022)