MagPy: Compiling Eager Mode DNN Programs by Monitoring Execution States

Chen Zhang, Rongchao Dong, Haojie Wang, Runxin Zhong, Jike Chen, and Jidong Zhai, *Tsinghua University*

https://www.usenix.org/conference/atc24/presentation/zhang-chen

This paper is included in the Proceedings of the 2024 USENIX Annual Technical Conference.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-41-0

MagPy: Compiling Eager Mode DNN Programs by Monitoring Execution States

Chen Zhang Rongchao Dong Haojie Wang Runxin Zhong Jike Chen Jidong Zhai

Tsinghua University

Abstract

Real-world deep learning programs are often developed with dynamic programming languages like Python, which usually have complex features, such as built-in functions and dynamic typing. These programs typically execute in eager mode, where tensor operators run without compilation, resulting in poor performance. Conversely, deep learning compilers rely on operator-based computation graphs to optimize program execution. However, complexities in dynamic languages often prevent the conversion of these programs into complete operator graphs, leading to sub-optimal performance.

To address this challenge, we introduce MagPy[1] to optimize the generation of operator graphs from deep learning programs. MagPy generates more complete operator graphs by collecting key runtime information through monitoring program execution. MagPy provides a reference graph to record program execution states and leverages reference relationships to identify state changes that can impact program outputs. This approach significantly reduces analysis complexity, leading to more complete operator graphs. Experimental results demonstrate that MagPy accelerates complex deep learning programs by up to 2.88× (1.55× on average), and successfully instantiates 93.40% of 1191 real user programs into complete operator graphs.

1 Introduction

In recent years, deep learning models have shown their power in various domains like biological science, weather forecasting, and recommendations. For ease of programming, users usually program deep learning models in Python and call tensor libraries like PyTorch when a tensor operation is needed. Then, the program is eagerly executed by running tensor operations one by one, which usually leads to poor performance.

Conversely, to boost the speed of deep learning models, deep learning compilers usually convert deep learning models into operator graphs to perform graph-level optimizations, such as graph substitution [22, 38, 43] and operator fusion [27, 29, 32, 41, 46]. Once operator graphs are available, TorchInductor [1] and XLA [3], two representative deep learning compilers, can accelerate DNN models by 1.65× and 1.27× on average over eager execution. Detailed results are shown in Figure 1, denoted as `Fullgraph-Inductor` and `Fullgraph-XLA`. However, achieving such improvements requires manually converting user programs into operator graphs, which is difficult for many model developers. Especially given the current extensive use of deep learning, an increasing number of models are developed by non-professional programmers, like scientists in chemistry, biology, and astronomy. Consequently, there is a growing demand for automatically converting users' easy-to-write programs with Python to compiler-friendly graphs to accelerate programs, which we call *operator graph instantiation* in this paper.

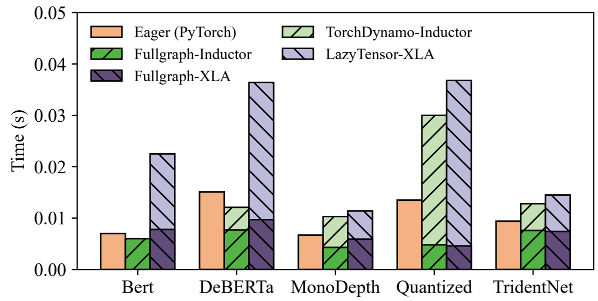


Figure 1: Deep learning compilers can significantly accelerate DNN models when operator graphs are available (`Fullgraph-Inductor` and `Fullgraph-XLA`), but current performance cannot be guaranteed due to existing graph instantiation techniques (`TorchDynamo-Inductor` and `LazyTensor-XLA`). Tested with batch size 1 on A100.

Existing techniques for graph instantiation can be classified into analysis-based and trace-based approaches. Analysis-based approaches, including Janus [20], `torch.jit.script` [2], and TorchDynamo [1], generate operator graphs by trying to analyze the exact behavior of the source code. Trace-based approaches,

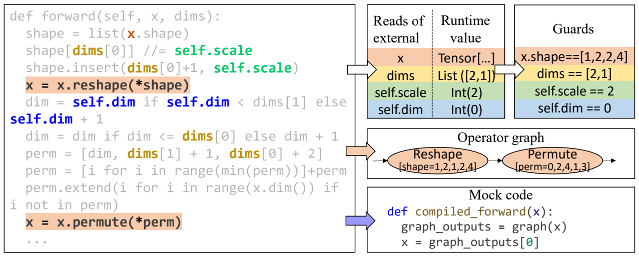
[1] MagPy is available at https://github.com/heheda12345/MagPy

including AutoGraph [28], torch.jit.trace [2], torch.fx.symbolic_trace [31], Terra [23], LazyTensor [33], and Torchy [26], generate operator graphs by executing user programs and recording all tensor operations via overloading the tensor operations. Neither approach can achieve the optimal instantiation result when encountering complex models. As shown in Figure 1, TorchDynamo causes 2.08× overhead (TorchDynamo-Inductor vs. Fullgraph-Inductor) and LazyTensor causes 2.85× overhead (LazyTensor-XLA vs. Fullgraph-XLA) on average compared with directly compiling the graph with the same deep learning compiler.

Analysis-based approaches fail to generate graphs when programs use flexible Python features not yet supported by the compiler, like advanced built-in functions and complex inheritance. As a result, these approaches can only extract multiple small graph fragments from user programs. Significant runtime overhead will be introduced by executing unsupported parts in a slow Python Interpreter and frequent switches between Python and operator graph execution. Global graph optimizations by deep learning compilers are also impeded. Supporting all Python features is impractical because it is even harder than re-implementing a Python interpreter. Moreover, the Python interpreter is under active development with a large community; significant human efforts are required to align the modifications and support new Python features. Trace-based approaches can only collect an operator graph for one execution, but they fail to know whether the graph will change for another execution. As a solution, they either rely on users to ensure the graph remains unchanged, potentially introducing silent bugs, or retrace the operator graph for each execution, which results in a large execution overhead.

The key challenge in graph instantiation is to understand dynamic and flexible user programs. Analysis-based approaches try to achieve this by re-implementing a small subset of Python features in their compilers. In contrast, instead of understanding user programs, trace-based approaches execute programs repeatedly. Both approaches fail to provide enough information for compilers to understand user programs and extract operator graphs precisely. Fortunately, Python interpreter, being a mature software that accurately supports all Python features, should be able to provide valuable information for graph instantiation. Effectively leveraging the information provided by the interpreter, we can gain a more comprehensive understanding of a program's behavior, thus facilitating graph instantiation. Therefore, in this paper, we propose MAGPY[2] to enhance operator graph instantiation by *monitoring the execution states of Python interpreter*. MAGPY is designed based on the following insights.

First, most deep-learning programs only have limited dynamics. Though programs are written in Python, with potential dynamics like data type, control flow logic, and runtime function dispatch, the operator graph structures, (e.g., operator orders, operator attributes, and operator shapes), usually remain the same across different batches. ParityBench[3] is a benchmark that crawls deep learning programs written in PyTorch and with over 100 stars from Github. 83% of that benchmark's 1421 deep learning programs satisfy limited dynamics. Their operator graphs can be obtained by monitoring tensor operations during program execution. Thus, instead of analyzing "*what the graph is*" like analysis-based approaches, which require a full simulation of a given user code, we only need to know "*when the graph will change*".



```
def forward(self, x, dims):
  shape = list(x.shape)
  shape[dims[0]] //= self.scale
  shape.insert(dims[0]+1, self.scale)
  x = x.reshape(*shape)
  dim = self.dim if self.dim < dims[1] else
self.dim + 1
  dim = dim if dim <= dims[0] else dim + 1
  perm = [dim, dims[1] + 1, dims[0] + 2]
  perm = [i for i in range(min(perm))]+perm
  perm.extend(i for i in range(x.dim()) if
i not in perm)
  x = x.permute(*perm)
  ...
```

Reads of external	Runtime value	Guards
x	Tensor[...]	x.shape==[1,2,2,4]
dims	List ([2,1])	dims == [2,1]
self.scale	Int(2)	self.scale == 2
self.dim	Int(0)	self.dim == 0

```
def compiled_forward(x):
  graph_outputs = graph(x)
  x = graph_outputs[0]
```

Figure 2: The program logic remains the same as long as all values reading from external remain unchanged.

Second, only *external values* can affect program behavior. By leveraging this feature, we can significantly reduce the complexity of monitoring dynamics during graph instantiation. Program behaviors affecting program outputs include the operator graph and all side effects (writing to external). They will not change as long as all values that read from external, like input arguments and global variables, remain unchanged, and all operations are ensured to produce fixed results when feeding into unchanged inputs. Therefore, MAGPY only needs to validate the operations and generate guards to check whether external reads are unchanged to determine whether the graph is changed. For example, though the program in Figure 2 uses plenty of complex Python features, the operator graph will remain unchanged as long as all reads of external values (x, dims, self.scale, and self.dim, marked with bold), match the previous run. When the *guards* on these values pass, MAGPY can safely run a *mock* function for calling the accelerated operator graph and simulating the program behavior (writing x to external) of the previous run.

Third, both the *guard* and *mock* can be determined by analyzing program *execution states*. The guard validates whether the input state of a new call matches the previous run, and the mock reproduces the final state of the previous run. Both parts are only based on runtime states instead of the logic of the user program. The guards and mocks are performed on values explicitly read from or written to the external, together with the values they hold a reference. Therefore, MAGPY proposes to maintain reference relationships to record and analyze the

[2] Can be pronounced as "magpie", which is a kind of bird that builds a bridge between a separated couple Niulang and Zhinv every year in a Chinese romantic mythology.

[3] github.com/jansel/pytorch-jit-paritybench, commit b219ee7

program behavior.

Based on the above observation, MAGPY proposes *RefGraph* (Reference Graph) for recording program states during program execution. MAGPY defines execution state interface for gathering runtime information during program execution and uses annotation-based graph update rules to maintain the RefGraph. MAGPY also provides a searching algorithm on the RefGraph to generate graph instantiation results.

MAGPY is implemented on top of Python and PyTorch. MAGPY instantiates the operator graph in the format of Torch.fx [31], a widely used graph format that is already supported by mainstream deep learning compilers like TorchInductor [1] and XLA [3]. The graph instantiation by MAGPY can be enabled by adding only one line of code. Evaluation of 8 typical models shows that MAGPY achieves 1.55× speedup on average (up to 2.88×) compared to existing graph instantiation techniques when using the same deep learning compiler to compile the operator graphs. MAGPY also passes the correctness check of ParityBench, and successfully instantiates 93.40% of the 1191 limited dynamic models in the benchmark into complete operator graphs, reducing the rate of unsupported Python features by 3.44× compared with TorchDynamo, the state-of-the-art analysis-based framework.

We summarize the main contributions below.

1. We identify the limited dynamics of deep learning programs, a property that makes effective graph instantiation practical.
2. We propose RefGraph to record program states during execution and monitor potential dynamics during graph instantiation. This significantly reduces the complexity of dynamic analysis, facilitating the implementation of better graph instantiation.
3. We propose an annotation-based approach to handle the complexity of programming languages, including builtin functions, user-defined functions, etc., for the better generation of RefGraph.
4. We design and implement MAGPY, an effective operator graph instantiation system for deep learning programs in Python. Evaluation shows that MAGPY achieves up to 2.88× (1.55× on average) speedup over state-of-the-art systems and successfully instantiates 93.40% of the 1191 limited-dynamic real user models in ParityBench to complete operator graphs.

2 Background and Challenges

2.1 Challenges

Operator graph instantiation of deep learning programs is challenging because both Python and PyTorch are designed for programming easily and thus provide abundant flexible features. The compiler has to handle all these features correctly to generate a correct operator graph. We manually analyze 272 real user programs in ParityBench that TorchDynamo fails to export complete operator graphs though these operator graphs are actually static. The main obstacles for graph instantiation are listed below:

```python
#1:LoSealL/VideoSuperResolution
perm = [2, 1]
perm.extend(i for i in range(x.dim()) if i not in perm)
x = x.permute(*perm)
#2:rawmarshmellows/pytorch-unet-resnet-50-encoder
for i, block in enumerate(self.down_blocks, 2):
    ...
#3:bayesiains/nflows, numel:number of elements
x * torch.Size(x.shape).numel()
```

(a) Builtin functions

```python
# pabloppp/pytorch-tools
class EqualLeakyReLU(nn.LeakyReLU):
  def __init__(self, scale):
    super().__init__()
    self.scale = scale
  def forward(self, x):
    return super().forward(x) * self.scale
```

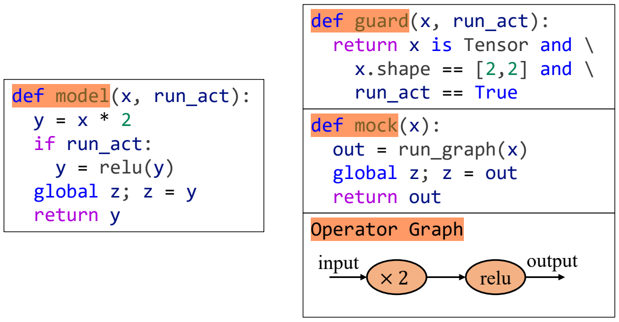
(b) OOP Programming

```python
# Deberta model in huggingface/transformers
def forward(self, inp, mask, dim):
  self.dim = dim
  output = inp.masked_fill(mask, ...)
  output = torch.softmax(output, self.dim)
  return output
```

(c) Side effect

Figure 3: Real DNN programs with complex features highlighted. `xxx/yyy` means the program is from `github.com/xxx/yyy`.

- **Dynamic data types.** Python is dynamically typed, meaning that the type of a Python expression can only be known during program execution. The example programs in Figure 3 have no type annotations, so the compiler cannot know which operations are operated on tensors and should be inserted into the generated operator graph. Almost all user programs do not provide the exact data type of variables.
- **Numerous builtin APIs.** Python and PyTorch provide various useful APIs to operate on objects, including but not limited to the ones used in Figure 3a: advanced container operations (#1), flexible iterators (#2), and Tensor metadata processing (#3). The compiler needs to know the exact result of these APIs to construct operator graph. However, the number of builtin APIs is too many to be implemented one by one. 47.2% of Dynamo failures come from unsupported builtin APIs.
- **Complex object-oriented programming (OOP).** Users tend to define new classes in OOP style to encapsulate the repetitive parts of their models, as shown in Figure 3b. As Python is a dynamically interpreted language, it is difficult for the compiler to find actual objects and functions. 34.7% of Dynamo failures result from the limited ability to handle user-defined classes.
- **Side effects.** Variable mutations also widely exist in Python



```
def model(x, run_act):
  y = x * 2
  if run_act:
    y = relu(y)
  global z; z = y
  return y
```

```
def guard(x, run_act):
  return x is Tensor and \
    x.shape == [2,2] and \
    run_act == True
```

```
def mock(x):
  out = run_graph(x)
  global z; z = out
  return out
```

(a) The user program (b) Guard, mock, and operator graph

Figure 4: Example of compiled record

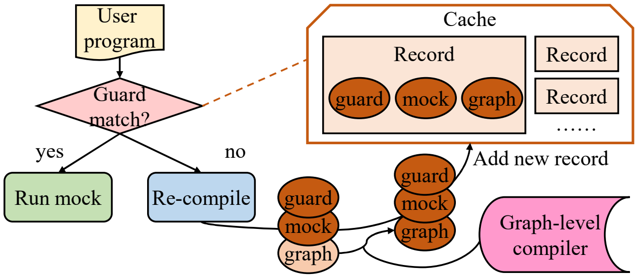


Figure 5: Workflow of guard-based just-in-time compilation

programs. In Figure 3c, the function `forward` writes to and reads from the `dim` attribute at lines 2 and 4. These modifications can change the behavior of subsequent function calls, so they should be handled correctly. 6.3% of Dynamo failures are caused by this.

Moreover, there is a large semantic gap between the Python programming language and operator graph representation. For better performance, graph-level compilers prefer a simple and static graph representation. Therefore, the above complex Python features cannot be represented in an operator graph and have to be eliminated during graph instantiation.

Despite the above challenges, the exact behavior of user programs is usually the same across different batches. Therefore, as long as it can be verified that the program behavior remains unchanged, MagPy can safely reuse the operator graph in the previous run without understanding the precise semantics of these programs.

2.2 Guard-based Just-in-time Compilation

Given that many variables remain constant across different batches, deep learning programs can be specialized with these unchanged variables by just-in-time (JIT) compilation. For example, when calling the `model` function in Figure 4(a) with a 2×2 Tensor `x` and `run_act=True`, the JIT compiler specializes the program into a *mock* in Figure 4(b). The specialized mock function executes a static *operator graph* without a branch on `run_act` and reproduces the side effect in `model` function (updating the global variable `z`). To ensure the correctness of specialization, the JIT compiler also generates a *guard* function to validate the assumptions of the specialized

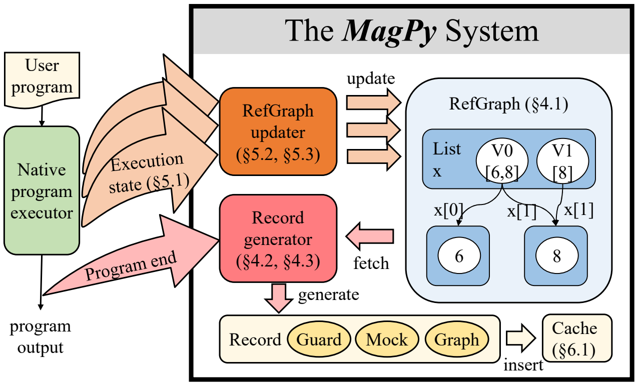


Figure 6: MagPy Overview

program. When the guard passes for new inputs, the specialized `mock` function will produce the same result as the original user program. We call the collection of a guard function, a mock function, and an operator graph as a *record* in this paper.

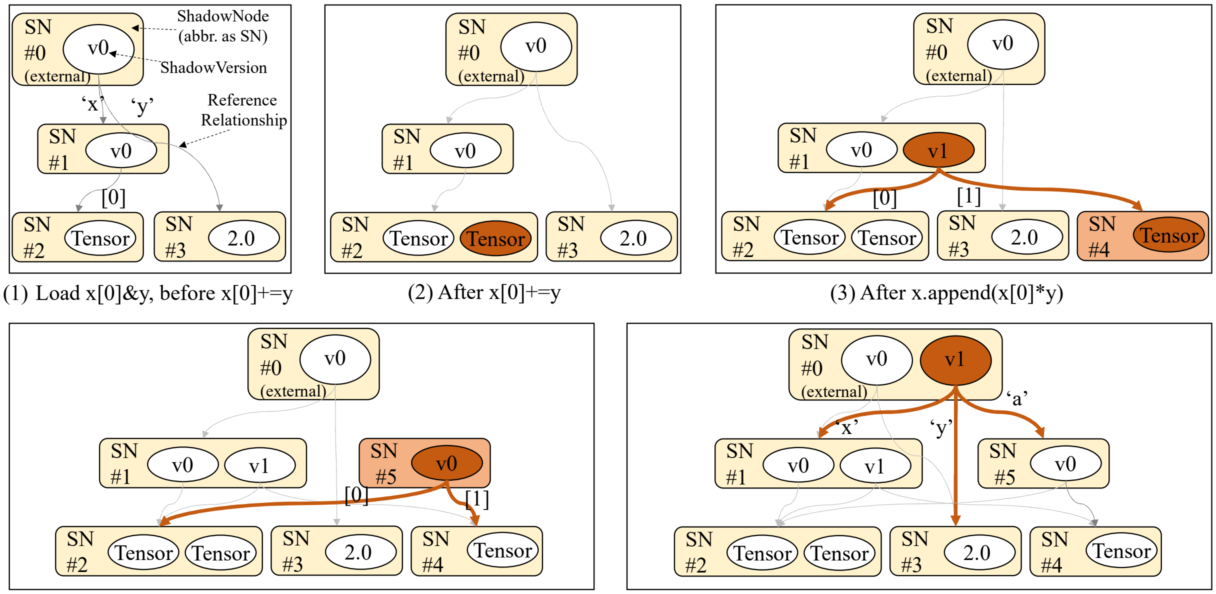
As shown in Figure 5, the compiled records are saved in a cache of the JIT system. When the compiled function is called, the JIT system first searches the cache for a record whose guard can be successfully passed. If a matching record is found, the JIT system will replace the call by calling a `mock` associated with that record. Otherwise, the JIT system will recompile the user program to generate a new record and call the graph-level deep learning compiler to optimize the operator graph in the record. Then, the record is saved into the cache for future use.

3 Overview

The overview of MagPy is shown in Figure 6. When not finding a matched record, MagPy recompiles the user program by executing the program with the native language executor (e.g., Interpreter of Python) and monitors execution. We call the process a *monitor run* of the user program. Different from traditional compilers that analyze program logic based on structures like control flow graphs, MagPy only cares about program states. MagPy proposes a *RefGraph* (Reference Graph) to save runtime state information, mainly the reference relationship between runtime variables of the user program. MagPy captures the *Execution State* of each instruction from the native executor during program execution and updates the RefGraph based on captured execution states. When the program is finished, MagPy generates a new record by analyzing the RefGraph and saves the new record into the record cache.

4 Design

In this section, we will introduce the definition of RefGraph (§4.1), plus the algorithms to generate guard, mock, and operator graph based on the RefGraph (§4.2-§4.3).



(1) Load x[0]&y, before x[0]+=y

(2) After x[0]+=y

(3) After x.append(x[0]*y)

(4) After building tuple(x), before assigning to a

(5) After a=tuple(x)

Figure 7: Update of reference graph when monitoring program in Figure 9. The changes in different steps are highlighted.

```
class RefGraph:
 sn: List[ShadowNode]
 def get_node_by_var(variable: Any)->int
 def get_node_by_id(node_id: int)->ShadowNode
```

(a) Reference Graph

```
class ShadowNode:
 node_id: int
 versions:
 ↪  List[ShadowVersion]
```

(b) ShadowNode

```
class ShadowVersion:
 value: Any
 refs: List[Pair<Relation,
 ↪  dest_id>]
```

(c) ShadowVersion

Figure 8: Definition of reference graph

```
def f(x, y, z):
  x[0] += y # inplace update x[0] to x[0]+y
  x.append(x[0]*y) # add x[0]*y to end of list x
  a = tuple(x) # build tuple a with elements in x
  ......

tensor=torch.rand((3,3))
f([tensor], 2.0, 3.0) # list of tensor, float, float
```

Figure 9: An example deep learning program

4.1 Definition of RefGraph

The definition of RefGraph is shown in Figure 8. It contains the following elements.

RefGraph RefGraph saves the runtime information, with *ShadowNodes* (correlation each runtime variable) as graph nodes and *reference relationships* of the runtime variables as graph edges. MagPy can retrieve the ShadowNode using either the runtime variable or the ShadowNode id. Figure 7 shows the example updating process of RefGraph when MagPy monitoring the execution of deep learning program in Figure 9. The RefGraph is updated during the execution of programs, and the details of this process will be explained in the following part.

ShadowNode Each runtime variable (including both Tensor and non-Tensor variables) is represented by a ShadowNode in the RefGraph. A special ShadowNode `SN #0` is introduced for saving the state of external runtime, e.g., global variables and arguments in Python. ShadowNodes will be only created lazily when the interpreter explicitly accesses the variables in the runtime. By doing that, MagPy can eliminate the overhead of creating ShadowNodes for all variables and avoid over-specialization of unused variables during guard generation. Figure 7(1) shows the RefGraph after the runtime reading `x[0]` and `y` from external and before performing `x[0] += y`. MagPy only creates ShadowNodes for three used variables `x` (`SN #1`), `x[0]`(`SN #2`), and `y` (`SN #3`) respectively, while ignores unused argument `z`.

ShadowVersion MagPy introduces ShadowVersion to hold the variables' update history. The update history helps recover initial input values for guard generation and find in-place updates for mock generation. Though MagPy only needs the input and final states of runtime variables, we introduce MagPy as recording all versions of the variables in this paper to make the explanation fluent. A new version is created when an in-place operation happens in the program runtime. For each version, MagPy records the value and variables that it holds a reference in that version. For example, the in-place add `x[0] += y` in step (2) creates a new version for `x[0]` (`SN #2`), and `x.append()` in step (3) creates a new version for `x` (`SN #1`).

Reference relationship Many composite variables hold references to other variables. For example, a list in Python holds the reference of elements in it. The reference relation-

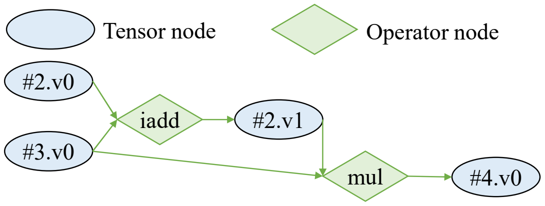


Figure 10: The result operator graph

ship affects the set of variables to be guarded unchanged and to be reproduced during mocking. Therefore, MAGPY saves such information explicitly, with edges from a specific ShadowVersion of a composite variable to the whole ShadowNode of the variable whose reference is held. The edges are saved in the `ref` field of ShadowVersion. The detailed reference relationship is saved as an attribute of the edge. For simplicity, Figure 7 only depicts the detailed relationship of newly-created edges in each step. MAGPY only needs the state of reference, which can be easily obtained by inspecting runtime variables, and does not care about why such a state is formulated. For example, in step (3) of Figure 7, MAGPY can know list `x` contains `SN #2` as `x[0]` and `SN #4` as `x[1]` by observing the value of `x` in the runtime without knowing `append` in Python is for adding a new element into the list; and in step (4), though the tuple (`SN #5`) is created from list `x` (`SN #1`), these two ShadowNodes are independent without an edge like in a traditional data flow graph. The reference relationship from a variable `x` to the target `y` that `x` holds a reference can be generated lazily if the target can only be explicitly visited, e.g., the attributes of user-defined variables. However, the relationship should be generated when visiting `x` if the target variable may be implicitly visited, like the contained variables of a container.

Operator graph In addition to the generated RefGraph, MAGPY also records the operator graph of the executed program, which contains all Tensor operations that visit the tensor data, as shown in Figure 10. The ShadowNode id and version id in RefGraph are also recorded for each Tensor. By using RefGraph, MAGPY can know where to load the input Tensors and where to store the output Tensors.

The RefGraph records all information needed to generate the guard function for validation and the mock function for calling the recorded operator graph properly. As RefGraph mainly saves the exact reference relationship of runtime variables, MAGPY can build the graph by examining the runtime state of the executor while not needing much knowledge of the complex language semantics.

4.2 Guard and Mock Generation

Due to the lazy creation of ShadowNode and reference relationship, the RefGraph in MAGPY naturally filters the set of *key variables* that have real influence on the program output. MAGPY then uses Algorithm 1 to collect the key variables for guard and mock via searching on the RefGraph.

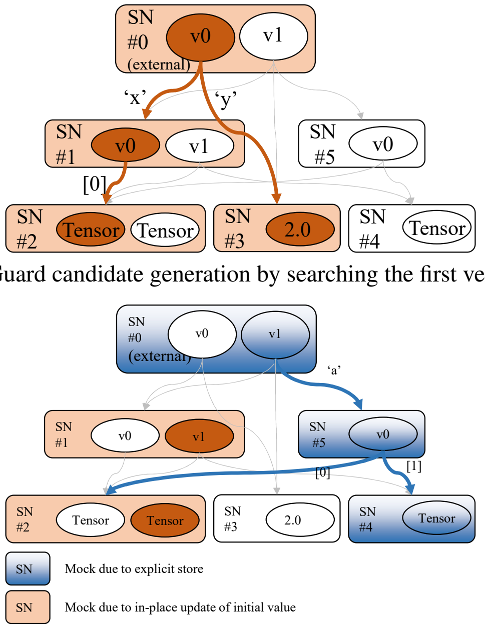
Algorithm 1: Guard and mock generation via searching reference graph

```
    Data: Reference graph $G$
    Result: List of ShadowNodes $result$
1   Function Search(G, src_sn, version):
2       $result \leftarrow$ empty list, $queue \leftarrow$ empty queue;
3       enqueue $src\_sn$ into $queue$;
4       add $src\_sn$ to $result$;
5       while $queue$ is not empty do
6           $cur\_sn \leftarrow$ dequeue from $queue$;
7           foreach $\_, id \in cur\_sn.v[version].refs$ do
8               $ref\_sn \leftarrow G.get\_node\_by\_id(id)$ ;
9               if $ref\_sn \notin result$ then
10                  enqueue $ref\_sn$ into $queue$;
11                  add $ref\_sn$ to $result$;
12      return $result$;
13  Function GetGuardNodes(G):
14      return Search($G$, $G.sn[0]$, 0)
15  Function GetMockNodes(G):
16      $final\_nodes \leftarrow$ Search($G$, $G.sn[0]$, -1) ;
17      $initial\_nodes \leftarrow$ Search($G$, $G.sn[0]$, 0) ;
18      $result \leftarrow$ empty list ;
19      foreach $sn \in final\_nodes$ do
20          if $sn \notin initial\_nodes$ then
21              add $sn$ to $result$ ;
22      foreach $sn \in initial\_nodes$ do
23          if $len(sn.version) \neq 1$ then
24              add $sn$ to $result$ ;
25      return $result$;
```

Guard generation The goal of the guard is to verify whether the initial state is not changed when the program starts. It needs to check the variables that the runtime explicitly read from the external state during the monitor run. All these variables exist at the function entry, and can be reached in RefGraph by walking from the external state (`SN #0`) only via the reference relationship of version 0. MAGPY collects these nodes by `GetGuardNodes` in Algorithm 1. The variables to be guarded and the guard function for the example code in Figure 9 are shown in Figure 11a and Figure 11c.

Mock generation The goal of mock is to reproduce the final program state of the monitor run without executing the user program. The mock needs to reproduce all variables that may be used outside the compile region. These variables can be divided into two types, both of which can be collected from RefGraph by searching from the external state `SN #0`, as shown in `GetMockNodes` of Algorithm 1.

1. Variables that are created during program execution and explicitly stored to the external, e.g., `a` in the example program. These variables can be collected by searching from the external node `SN #0` with reference relationship in the last version as edges (line 16, 19- 21 in Algorithm 1). However, if one variable exists at the function entry and is not in place updated (with only one version in its ShadowNode), MAGPY can skip reproducing it and use its initial value



(a) Guard candidate generation by searching the first version

(b) Mock candidate generation by searching first and last version

```
def guard_f(x, y, z):

  return sn[1].match(x) \
    and sn[2].match(x[0]) \
    and sn[3].match(y)
```

```
def mock_f(x, y, z):
  sn2,sn4 = run_graph(x[0],y)
  sn[2].inplace_upd(x[0],sn2)
  sn[1].inplace_upd(x,sn2,sn4)
  a = sn[5].mock(sn2, sn4)
```

(c) Generated guard (d) Generated mock

Figure 11: Nodes inside the reference graph

instead.

2. Variables that exist at the function entry and are in-place modified, e.g., `x` and `x[0]` in the example program. These variables may be visited outside the compile region, so even if there is no path from the external node `SN #0` with the reference relationship in the last version, these variables still need to be updated in mock. These variables are collected by line 17, 22- 24 in Algorithm 1.

The mock will first call the operator graph accelerated by graph-level compilers to get the output Tensors (with details in §4.3). Then, the mock will reproduce the variables as in the monitor run. The variables of type 1 can be created from scratch, and variables of type 2 need to be in-place updated. Figure 11b and Figure 11d show the collected nodes and generated mock code of the example code respectively.

Pointer alias analysis is a challenge in compiler design. However, in MagPy, the pointer alias relationship is naturally represented in RefGraph, with different paths to reach the specific node from `SN #0`. In addition to the value guards mentioned above, MagPy also verifies the pointer alias relationship matches the monitor run. It can be achieved by checking all paths to the same node in RefGraph reach the same variable. The mock also needs to reproduce the reference relationship, which can be achieved by simply reproducing the reference relationship in the RefGraph.

4.3 Operator Graph Generation

For the recorded operator graph, MagPy will determine the input and output nodes, together with where these tensors should be loaded from or stored to the runtime during mocking. The input nodes are the Tensor nodes with no in-edge in the operator graph. These Tensor variables are not created by tensor operations, so they should exist at the function entry. Therefore, there are reference paths using edges of version 0 from `SN #0` to the corresponding nodes in RefGraph. The output nodes are the Tensor variables that need to be mocked and are determined during mock generation. The output nodes are also available by searching from `SN #0` in RefGraph, and the path represents the stored position of the output Tensors. MagPy can ensure that the intermediate nodes will not be used in the future and pass this information to the graph-level compiler. So the graph-level compiler can safely perform optimizations like dead code elimination or operator fusion. However, trace-based frameworks fail to achieve this because they cannot know whether a Tensor will be used in the future.

5 RefGraph Generation

MagPy generates the RefGraph by analyzing the intermediate runtime state of the monitor run. Specifically, MagPy captures the *execution state* (§5.1) of each instruction during monitor run, gets the pre-defined *operation property* (§5.2) of the instruction, and updates the RefGraph with different *RefGraph update rules* (§5.3) based on the property.

5.1 Interface of Execution State Capture

Dynamic languages usually save the high-level information of runtime variables for dynamic interpretation, e.g., types and variable structures. Such information is valuable for MagPy to analyze the runtime state. Therefore, MagPy collects the execution state of each instruction from the runtime, which includes all relevant variables of the instruction. The execution state contains the following elements:

- `OP`: the operation that will be performed, like loading a global variable or adding two variables.
- `input[x]`: the input to the operation indexed by *x*. For example, the `input` of a function call is its argument list, and the `input` of an `assign` operation is the value that will be stored.
- `output[x]`: the output of the operation indexed by *x*. E.g., the `output` of a function call is its return value.

5.2 Operation Property Annotation

For performance concerns, dynamic languages use native machine code to implement some operations. For example, `Python` implements some functions in `C`. These low-level machine codes do not preserve high-level variable information and are difficult to analyze. To collect all operators and run MAGPY correctly, the *property annotation* of these functions are needed. Note that they are just annotations, and are much easier than re-implementing the functions as in analysis-based approaches. MAGPY also encourages annotation on frequently-used functions implemented by the dynamic language, to allow MAGPY to skip diving into the implementation detail of the function and make the monitor run faster. The required annotations are listed below.

Function Properties are annotated for functions:

- `AsOpNode`: a function to convert the operation to an operator node of the operator graph. Returns None if the operation cannot be represented by an operator node.
- `PureClosure`: a boolean attribute that can be annotated as True if the return values are identical for identical input arguments, and no external effect happens like implicitly updating the global variables. In-place updates of input arguments are allowed when `PureClosure` is `True`.

`AsOpNode` is defined for generating the operator graph. `PureClosure` identifies dynamic behaviors in Python like system calls and random number generators, whose annotations will be `False`.

Input Properties are annotated for each input variable of the function:

- `ValueRead`: a boolean property that is `True` when the function reads the value of that input variable.
- `InPlace`: a boolean property indicating the variable may be in-place modified by the function.

For example, all arguments of mathematical operations like `add` or `sub` will be annotated as `ValueRead=True`. For function `list.append` that adds a variable to the list, the first argument (the list) is annotated as `ValueRead=True` as its internal structure is accessed by the function. In contrast, the second argument (the variable to be added) is annotated as `ValueRead=False` because the function only adds its reference to the list and does not access its value. Moreover, the first argument is annotated as `InPlace=True` because the list is in-place modified with the specified value, while the second argument is annotated as `InPlace=False`.

Output Properties are annotated for each output variable:

- `InPlace`: a boolean property indicating the variable may be in-place modified by the function.
- `Relation`: marks the relationship that needs to be created between the input and output node, together with the detailed relationship. Need to specify further whether it is a `read` through existing reference (e.g., reading `v.x`) or a `write` that creates a new reference (e.g., assigning `v.x = 1.0`).

The user effort of making such annotations is affordable. Despise 6 properties in total, only `PureClosure` for operations plus `InPlace` for input and output variables are frequently annotated. The `AsOpNode` only needs to be annotated for the limited number of operations supported by the graph-level compiler. And `ValueRead` is necessary only when the function can read a Tensor because of the graph update rule in §5.3. `Relation` is only used for special lazily-create reference relationships like explicitly loading an attribute. Moreover, many operations have the same annotation, like the $+-\times\div$ operations, allowing for mass production of annotations.

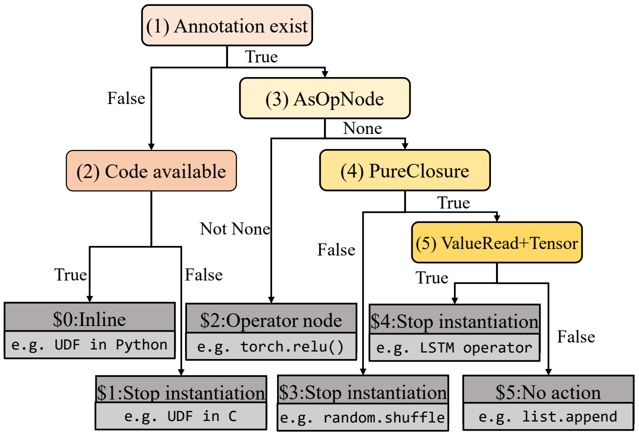


Figure 12: Operator graph update and dynamics detection

5.3 Graph Update Rule

MAGPY first generates ShadowNodes for input and output variables that do not have corresponding nodes in the RefGraph. MAGPY then retrieves the property annotation of the operation and updates the RefGraph and operator graph based on the annotations and captured execution state.

5.3.1 Operator Graph Update and Dynamics Detection

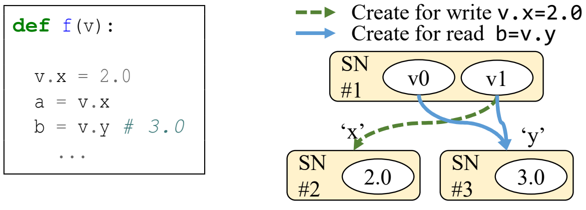
The `AsOpNode`, `PureClosure`, and `ValueRead` annotations are used for updating the operator graph and detecting the dynamics. The workflow is shown in Figure 12.

(1) Annotation Retrieval. MAGPY first attempts to retrieve the annotation of the function. Although annotations are provided for most built-in and commonly used library functions, some functions, like user-defined ones, may lack annotations. MAGPY goes to steps (2) or (3) based on whether the annotation exists.

(2) Handling Functions with No Annotations. If no annotation is found, MAGPY treats the function call as an unconditional jump into the function and tries to inline the function ($0). If the function code is unavailable, e.g., when implemented in native language, MAGPY will treat the function as dynamic and stop the graph instantiation ($1).

(3) Operator Graph Generation with `AsOpNode`. If `AsOpNode` successfully generates an operator node for the function (e.g., `torch.relu`), MAGPY will add the node to the operator graph ($2). Jump to (4) otherwise.

(4) Dynamic Function Detection with `PureClosure`. If `False`, MAGPY will regard the function as a dynamic func-



```
def f(v):

  v.x = 2.0
  a = v.x
  b = v.y # 3.0
    ...
```

(a) Example code (b) Reference relationship

Figure 13: Update of reference relationship

tion and stop graph instantiation ($3) [4]. If `True`, MAGPY will assume the function's behavior is unchanged in different batches, and jump to (5).

(5) Dynamic Tensor Computation Detection with `ValueRead`. MAGPY scans input arguments to find whether an input position is marked as `ValueRead=True` and its runtime value is a Tensor. If such an argument is found, MAGPY will treat the function as dynamic and stop graph instantiation ($4). The reason is that MAGPY allows the change of Tensor data in different batches, so these operations will generate different results as they read the data of the Tensor. Typical cases are the tensor operations not supported by the graph-level compiler like the `nn.LSTM` for `Inductor`. These operations are not converted to operator nodes by `AsOpNode` and reach this branch. Otherwise, the operations only read the value of non-tensor variables, like `list.append` only reads the `list`, and can be ensured to provide fixed results ($5). In such cases, no change is needed.

When detecting dynamic functions and stopping graph instantiation, MAGPY will use dynamic function calls as split points to cut the user program into subprograms. MAGPY can compile each subprogram separately by monitoring a single run of the user program, and generate the compiled code that runs the dynamic functions eagerly.

5.3.2 Update of RefGraph

The `Relation` and `InPlace` annotations are used for updating the RefGraph.

`Relation` for creating reference edges. If a reference relationship is annotated between two variables, MAGPY will add reference edges between the corresponding ShadowNodes in the RefGraph. For a `write` operation like `v.x=2.0` in Figure 13a, MAGPY creates a new version in the source ShadowNode, and adds only one reference edge from this version. For a `read` operation, MAGPY will create no edge if the relationship already exists in the last version of the source ShadowNode (e.g., `a = v.x` in Figure 13a), and create edges from all versions of the source ShadowNode to the target ShadowNode if the relationship does not exist (e.g., `b = v.y` in Figure 13a) because the reference exists right from the beginning of program execution.

`InPlace` for version management. If `InPlace=True`, MAGPY will add a new version to the corresponding ShadowNode and copy all reference relationships of that version if they still exist.

6 Detect and Handle Dynamic Behaviors

Though most deep learning programs satisfy limited dynamics, some programs still have dynamic behaviors like dynamic-value scalars, dynamic-shape Tensors, and dynamic control flow. MAGPY can detect them automatically and handle them with best effort.

6.1 Dynamic Detection

The handling of dynamic functions is discussed in §5.3. Another source of dynamics comes from the dynamics of input arguments, which will cause the guard failure.

Cache for different inputs. When guards fail, MAGPY will re-generate a compiled record and save it to the cache. When a guard matches, MAGPY will update the cache to try the record earlier in subsequent calls.

Guard fail reason detection. When the number of records exceeds a threshold, MAGPY will assume the records are over-specialized and try to make weaker assumptions. MAGPY will locate the non-const variables that cause the guard failures and try to generate weaker guards for these variables (§6.2.1).

6.2 Dynamic Handling

6.2.1 Lift up Non-const Variables

When MAGPY detects a non-const input variable whose datatype is supported by the underlying graph-level compiler, MAGPY will regard the variable as a "Tensor node" in the operator graph, and handle the variable like a Tensor. MAGPY will record all operations of the variable into the operator graph and recompute all variables that depend on it, so that the value of the variable is allowed to change. Typical cases include feeding dynamic scalars and dynamic shapes to the user program. If the underlying graph-level compiler does not support the variable type, MAGPY will cut the user program to appropriate pieces that do not visit the variable.

6.2.2 Dynamic Control Flow

The user code is mixed with static and dynamic control flow.

Static control flow is usually used to try different architectures or hyperparameters, but the model structure is fixed across different batches. These control flow operations usually determine the jump target with static scalars. MAGPY will add proper guards for these scalars as discussed in §4 and remove the control flow operation in the generated graph and mock.

[4] MAGPY handles some specific random API (e.g., randint) by generating new random numbers in mock, so these APIs will not stop graph instantiation.

```
import MagPy
model = ResNet()
compiled_model = MagPy.compile(model)
compiled_model(torch.rand((1,3,224,224))
```

Figure 14: Example of using MAGPY

Dynamic control flow is typically used for getting the model architecture during runtime, and usually determines the jump target with dynamic Tensor value. MAGPY can convert these dynamic jumps to control flow operator in the operator graph when the target graph-level compiler supports dynamic control flow.

For loop operations, MAGPY monitors the execution of all iterations. If both the value of external reads and the operator graph are the same in different iterations, MAGPY can safely convert the recorded operator graph to the body of the `Loop` node.

For branch operations, MAGPY will force the runtime to execute both branches by automatically modifying the source code like in AutoGraph [28]. MAGPY can guarantee the correctness by monitoring the execution of two branches. Specifically, MAGPY first executes the branch that the program will not jump to, records all effects made by the branch, and ensures that all these effects are recoverable. If MAGPY detects a non-recoverable effect, MAGPY will cut the user program before the dynamic jump and run this jump eagerly. Then, MAGPY recovers all effects, runs the branch that the program will jump to, generates the `If` node by merging the operator graph of two branches, and merges the RefGraph of two branches.

7 Implementation

MAGPY is implemented by about 4000 lines of code on top of Python and PyTorch. The operator graph in MAGPY is exported to `torch.fx` format that is compatible with mainstream graph compilers. To enable MAGPY, only one line of `MagPy.compile` is needed, as shown in Figure 14. Then, during each call to the compiled model, MAGPY tries to match a compiled record or performs a monitoring-based recompilation if no match is found.

MAGPY monitors the interpreter state of executing each Python Bytecode using the per-bytecode callback of `sys.settrace` in Python. The capture is achieved by analyzing the frame passed to MAGPY during the callback of `sys.settrace`. The calling of guard match and mock function is implemented by modifying the ByteCode with the Frame Evaluation API of Python.

MAGPY regards each object in Python as a variable with its own ShadowNode. And the `get_node_by_obj` interface in RefGraph (Figure 8b) is implemented by a map between the object's `id` and the ShadowNode. To make the `id` of objects unique, MAGPY holds a reference of all objects to avoid garbage collection. MAGPY has implemented `ShadowNode.match` and `ShadowNode.mock` in Figure 11 for 29 commonly-used Python datatypes, and allows the existence of other data types during monitor run as long as they do not need to be guarded and mocked. The `match` returns true when both type and value match the monitor run. For scalar, the value refers to the exact value. For containers, MAGPY checks that all objects in the container equal the value at the same position during monitor run. For Tensors, MAGPY verifies the metadata is not changed, but does not check the value of Tensor. If MAGPY finds these guards are over-specialized, MAGPY will try to make weaker assumptions as in §6.2.1.

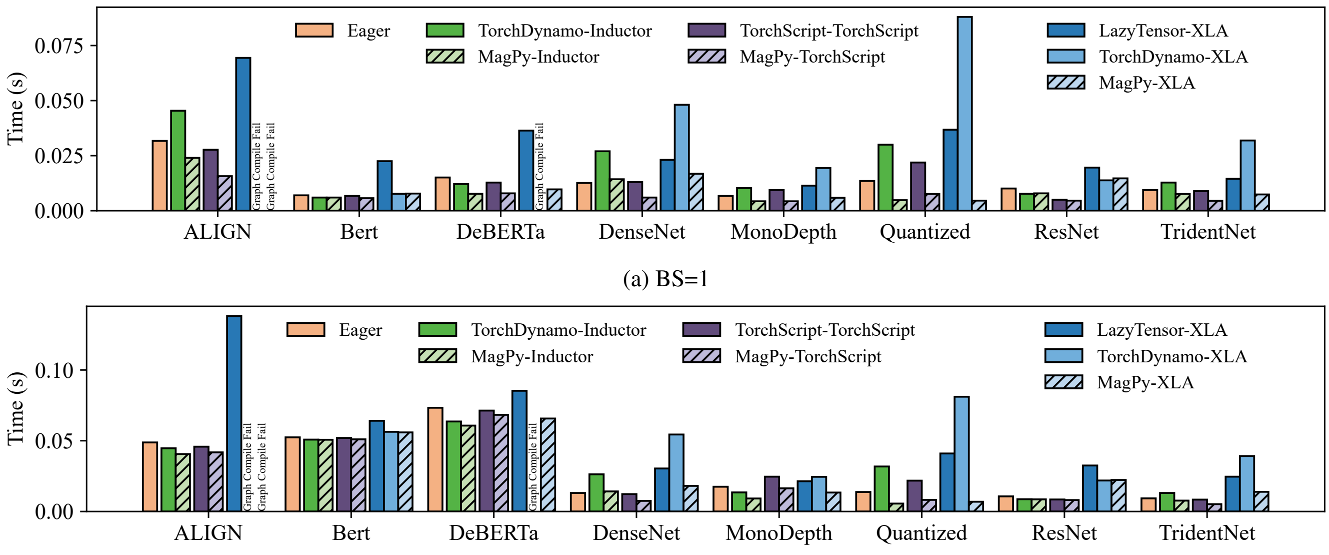
8 Evaluation

8.1 Evaluation Setup

Platform The evaluation is performed on one NVIDIA A100-PCIE-40GB GPU with two AMD EPYC 7742 CPUs. The software versions are Python 3.9, CUDA 11.8, and GCC 11.4.

Baselines We compare MAGPY with state-of-the-art graph instantiation techniques, including analysis-based frameworks TorchDynamo [1] and TorchScript [2] with PyTorch v2.0.1, and trace-based framework LazyTensor with torch.xla v2.0. TorchScript fails to compile some models, and we manually annotate the largest possible supported regions of the models. Some trace-based techniques, like `torch.jit.trace` [2] and AutoGraph [28], may cause silent errors theoretically, so they are not compared. Janus [20] and Terra [23] are not included as they are not open-sourced.

Benchmarks Our evaluation uses two sets of models. The first is ParityBench, a benchmark containing 1418 PyTorch models with over 100 stars on Github, to evaluate the coverage of different Python features in real user code. However, since the model input shape cannot be crawled, the shapes are automatically generated by the benchmark. The generated input shape is far from the real scenario, so we use 8 representative DNN models for time measurement. These models cover typical DNN architectures like CNN and transformers and include popular scenarios like multi-model and quantization. These models contain both classical models like Bert and ResNet that have been manually simplified to accommodate the currently limited graph instantiation ability, and models that are a bit less popular so not yet been simplified but still gain a lot of attention (e.g., with a considerable citation count). ALIGN [21] is a multi-modal that combines the Bert [9] language model and EfficientNet [35] vision model. Bert [9] is a classical Transformer model and DeBERTa [16] is another Transformer model with disentangled attention. DenseNet [18], MonoDepth [14], ResNet [15], and TridentNet [25] are four vision models with different CNN architectures. We use densenet-121, Monodepth-18, ResNet-101, and TridentNet based on ResNet-101, respectively. The Quantize model uses the algorithm [4] proposed by Banner et al. to quantize a ResNet model. The configurations, citations,



(a) BS=1

(b) BS=16

Figure 15: End-to-end inference on NVIDIA A100. `xxx-yyy` means extracting graphs via `xxx` and compiling graphs via `yyy`.

Table 1: Model information. BS denotes "batch size". The citation is based on Google Scholar as of January 8, 2024. `xxx/yyy` in source refers to `github.com/xxx/yyy`.

Model	Input shape	Citation	Source
ALIGN	text length 64 image [BS, 3, 289, 289]	2029	huggingface/transformers v4.29.1
Bert	text length 256	88345	huggingface/transformers v4.29.1
DeBERTa	text length 256	1595	huggingface/transformers v4.29.1
DenseNet	image [BS, 3, 224, 224]	41359	pytorch/vision v0.4.1
MonoDepth	image [BS, 3, 256, 256]	3151	OniroAI/MonoDepth-PyTorch b76bee4
Quantized	image [BS, 3, 224, 224]	330	eladhoffer/quantized.pytorch e09c447
ResNet	image [BS, 3, 224, 224]	195511	pytorch/vision v0.4.1
TridentNet	image [BS, 3, 224, 224]	932	open-mmlab/{mmdetection v2.28.2, mmcv v1.7.1}

and code source are listed in Table 1.

We use the original user code without modification except for fixing version compatibility issues of Python and PyTorch. Time is measured by averaging 100 runs after 100 warm-ups.

8.2 End-to-end Evaluation

Figure 15 compares the end-to-end inference performance of different graph instantiation approaches. As some approaches only support specific graph-level compilers, we compare the performance of different graph instantiation approaches using their default graph compiler with MagPy using the same graph compiler. MagPy achieves up to 2.93× speedup (1.73× on average) over eager execution, which is directly executed without compiling. When using TorchInductor, TorchScript, and XLA as graph compilers, MagPy achieves up to 6.25×(1.68× on average), 2.88×(1.56× on average), and 8× (1.78× on average) speedup respectively compared with the best implementation of each model using that compiler. The missing values in ALIGN model and DeBERTa model compiled by Dynamo-XLA and MagPy-XLA are caused by the bug in compiling FX graph with XLA backend.

The speedup of MagPy mainly comes from the reduced Python overhead. MagPy only needs to run a fast guard and a fast mock in Python. Though compiling some program fragments, TorchDynamo and TorchScript still need the Python interpreter to execute the Python code between the fragments. LazyTensor re-traces the graph in every run and it also suffers from the significant Python overhead. Moreover, GPU kernels are more efficient in some models because the larger operator graph generated by MagPy enables more optimizations by graph-level compilers. The kernel optimizations will be more significant in the near future because current graph-level compilers are exploring optimizing the operator graph in a larger scope. When both convert the model to one graph, MagPy-TorchScript is faster than TorchScript-TorchScript because MagPy can safely pass more static information to the TorchScript graph compiler.

The number of graphs exported by different technologies are listed in Table 2. The graph count of TorchDynamo is computed by counting the times it calls the graph compiler, the number of graphs exported by TorchScript is based on the times it enters the region marked with `torch.jit.script`, and the graph count of LazyTensor is based on the `CachedCompile` metric. These estimated graph counts are lower than actual values because some tensor operators may execute eager without compilation. MagPy can convert all evaluated models to full operator graphs, while TorchDynamo, TorchScript, and LazyTensor generate 24.88, 20.54, and 1.75 graphs on average. Though LazyTensor only generates one graph on most models, it is still slower than MagPy-XLA, proving its large runtime overhead.

Table 2: Number of exported operator graphs

Model	ALIGN	Bert	DeBERTa	DenseNet
TorchDynamo	168	1	53	59
TorchScript	204	36	37	3
LazyTensor	2	1	1	1
MAGPY	1	1	1	1
Model	**MonoDepth**	**Quantized**	**ResNet**	**TridentNet**
TorchDynamo	54	85	1	61
TorchScript	31	22	1	57
LazyTensor	1	44	1	1
MAGPY	1	1	1	1

Table 3: Result on ParityBench

Cases	Static cases	Compiler	Failed cases	Fail rate
1421	1191	TorchScript	769	64.5%
		TorchDynamo	272	22.8%
		MAGPY	79	6.6%

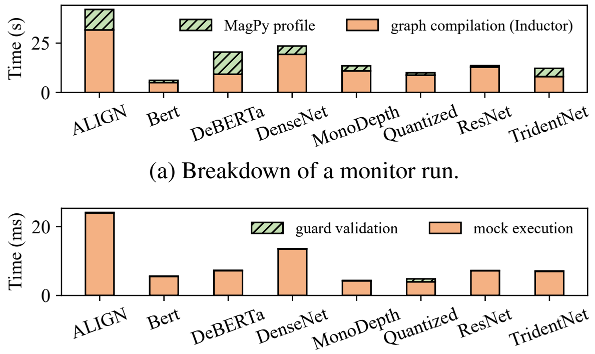
8.3 Coverage of Python Features

We use the ParityBench benchmark to evaluate the coverage of Python features. The benchmark initially contains 2000 real deep learning programs crawled from Github. We remove 490 cases that the benchmark fails to generate the model inputs. Though these models may run under specific input configurations, the input generator of ParityBench fails to find such configurations. We also remove 89 cases that all tests crash in eager mode. There are 1421 cases remained. MAGPY works correctly on all these models. By manually examining the cases, we find 230 (16.2%) models with dynamic behaviors, and MAGPY can detect them automatically.

The coverage test is performed on the remaining 1191 models that can be represented by one static operator graph theoretically. TorchScript (marking the model with `torch.jit.script`) and TorchDynamo (passing `full_graph=True`) fail to export 64.5% and 22.8% models to a full graph. In contrast, MAGPY does not cover only 6.6% models now and can successfully convert 93.4% models to a full graph. The result proves that MAGPY is more extensible for the complex Python features. The main reasons for uncovery are (1) The bool scalars generated by different expressions have the same object id in Python, so MAGPY cannot lift dynamic-value bool scalars to different nodes in the operator graph. Note that MAGPY works well on bool Tensors and static bool scalars; (2) Some models contain output that is not supported in the mock generation yet, such as `deque`, `Conuter`, and `UserDict`.

8.4 Overhead Analysis

The time breakdown of a monitor run for generating a new compile record and a matched run that runs a compiled record is shown in Figure 16. The graph-level compiler in the experiment is TorchInductor.



(a) Breakdown of a monitor run.

(b) Breakdown of a matched run.

Figure 16: Time breakdown of execution (BS=1)

During monitor run, MAGPY analyzes the interpreter state of executing each bytecode. The analyzation time takes 2.91 seconds on average, which accounts for 23% of the total time. The remaining 77% of time is spent on TorchInductor compiling the operator graph. The graph compilation time of TorchInductor is typical for graph-level compilers, and many graph compilers will take longer time, such as hundreds-of-seconds reported by recent works GraphTurbo [41], Welder [32], and EinNet [43].

For a matched run, MAGPY runs the guard to find the matched compile record and runs the compiled mock of the record. The time for guard and mock are 2% and 98% on average respectively.

8.5 Dynamic Scenarios

This section evaluates MAGPY on the two common dynamic scenarios including dynamic shape and dynamic control flow. Though not satisfying the default limited dynamics assumption, MAGPY can also generate a proper operator graph with technologies in §6.2.

Figure 17 shows the performance of models with fixed operator graph structures but fed with dynamic-shape inputs. The `model-bs` tests use the input shape in Table 1 but vary the batch size with a uniform distribution between $[2,16]$. The input texts of `model-seqlen` tests for language models are of batch size 8 and dynamic sequence length between 32 to 256. The operator graph is compiled by TorchInductor with dynamic shape enabled. Bert and ResNet are two simple models that TorchDynamo can also export full graphs. MAGPY can achieve similar performance with TorchDynamo on the two models, proving that MAGPY does not lose the hidden runtime information that analysis-based frameworks can collect. For complex models that are beyond the analysis ability of TorchDynamo, MAGPY can achieve 1.06×, 1.16×, and 2.05× speedup over TorchDynamo on DeBERTa with dynamic batch size, DeBERTa with dynamic sequence length, and DenseNet with dynamic batch size respectively.

Figure 18 shows the result of compiling models with dynamic control flow. LSTM [17] is a classical RNN model

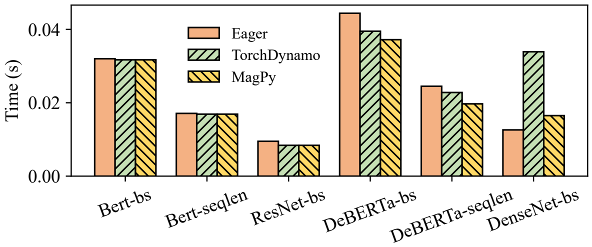


Figure 17: Inference of models with dynamic shape

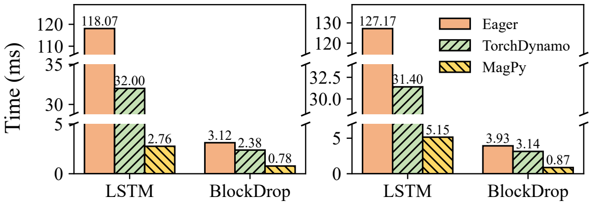


Figure 18: Inference of models with dynamic control flow

for NLP and contains a dynamic loop over static LSTM cells. We use a 10-layer LSTM with a hidden size of 256. The code is from PyTorch's official repository on Github. BlockDrop [39] is a convolutional neural network that dynamically skips some layers with a dynamic `if` statement. The source code is from `Tushar-N/blockdrop` on Github. As TorchInductor cannot compile operator graphs with dynamic control flow, we compile the exported operator graphs with Cocktailer [40], the state-of-the-art graph compiler for dynamic control flow. TorchDynamo chooses to unroll the whole LSTM model, which is over-specialized for the dynamic workload. To make a fair comparison, we manually mark the LSTM Cell as the compile region of TorchDynamo and run the loop in Python. MagPy achieves up to 11.59× speedup (5.96× on average) over TorchDynamo.

9 Related Work

Several approaches have been proposed to export operator graphs from user code. Janus [20], TorchScript [2], and TorchDynamo [1] get the operator graph by directly analyzing the user code. They only support a small subset of Python features and cannot export a complete operator graph when the program contains complex Python features. Other works get the operator graph by executing the program with special Tensors and tracing all operators on it. `torch.jit.trace` [2] and `torch.fx` [31] create a stand-alone operator graph that is never recompiled. AutoGraph [28] and `JAX.jit` [12] can automatically recompile when the input arguments to the function changes but cannot detect the change of other external values like global variables, so they still produce wrong results silently. LazyTensor [33], Terra [23], and Torchy [26] ensure the correctness by re-trace the operator graph in every run which will introduce large runtime overhead.

The operator graph exported by the above systems and MagPy will be fed to graph-level deep learning compilers. The graph level compilers then convert the operator graph to a faster implementation by technologies like graph substitution (e.g., TASO [22], PET [38], and EinNet [43]) and kernel fusion (e.g., Rammer [27], DNNFusion [29], AStitch [46], GraphTurbo [41], Welder [32]). Other works including TVM [8], FlexTensor [45], Ansor [42], AMOS [44], Roller [47], TensorIR [11], and Hidet [10] optimize the deep learning operators, and can accelerate both eager execution and compile-based execution.

Many domain-specific languages, including Triton [37] and FreeTensor [36], use Python as their frontend language and need to convert users' Python programs to their intermediate representations for further compilation. Graph instantiation technologies can be used for the conversion, helping to avoid silent errors (e.g., caused by global-parameter mutations), and to support more flexible grammars.

The just-in-time compilation based on runtime information is widely used in compiling general-purpose programs. Examples include PyPy [5] and Numba [24] for Python, java HotSpot [30], trace-JIT [19] for JAVA, TraceMonkey [13] for JavaScript. These tools convert a dynamic program into low-level machine code while MagPy creates high-level operator graphs. The process of instantiating the graph and compiling the graph is similar to multistage programming like MetaML [34], MetaOCAML [7], and BuiltIt [6]. However, multi-stage programming languages require users to specify the stage manually, and cannot automatically recompile when the external environment changes.

10 Conclusion

We propose MagPy, which exploits the limited dynamics inherent in deep learning programs to enable effective operator graph instantiation. MagPy introduces RefGraph to record program states and reduces the graph instantiation complexities by monitoring external values that impact program behavior. Evaluation shows that MagPy can accelerate complex deep learning programs by up to 2.88× (1.55× on average), and successfully instantiate 93.40% of 1191 limited-dynamic user programs into complete operator graphs.

Acknowledgements

We thank the anonymous reviewers their extensive suggestions. This work is supported by National Key R&D Program of China under Grant 2021ZD0110202, the Young Student Basic Research Fund of the National Natural Science Foundation of China (623B2061), NSFC for Distinguished Young Scholar (62225206), National Natural Science Foundation of China (U20A20226), the Young Scientists Fund of the National Natural Science Foundation of China (62202259). Jidong Zhai is the corresponding author of this paper (zhaijidong@tsinghua.edu.cn).

References

[1] torch.compiler. https://pytorch.org/docs/stable/torch.compiler.html.

[2] TorchScript. https://pytorch.org/docs/stable/jit.html.

[3] XLA | TensorFlow. https://www.tensorflow.org/xla.

[4] Ron Banner, Itay Hubara, Elad Hoffer, and Daniel Soudry. Scalable methods for 8-bit training of neural networks. *Advances in neural information processing systems*, 31, 2018.

[5] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: Pypy's tracing jit compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25, 2009.

[6] Ajay Brahmakshatriya and Saman Amarasinghe. Buildit: A type-based multi-stage programming framework for code generation in c++. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 39–51. IEEE, 2021.

[7] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. Implementing multi-stage languages using asts, gensym, and reflection. In *International Conference on Generative Programming and Component Engineering*, pages 57–76. Springer, 2003.

[8] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, 2018. USENIX Association.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[10] Yaoyao Ding, Cody Hao Yu, Bojian Zheng, Yizhi Liu, Yida Wang, and Gennady Pekhimenko. Hidet: Task-mapping programming paradigm for deep learning tensor programs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 370–384, 2023.

[11] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, et al. Tensorir: An abstraction for automatic tensorized program optimization. pages 804–817, 2023.

[12] Roy Frostig, Matthew James Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning*, 2018.

[13] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, et al. Trace-based just-in-time type specialization for dynamic languages. *ACM Sigplan Notices*, 44(6):465–478, 2009.

[14] Clément Godard, Oisin Mac Aodha, and Gabriel J Brostow. Unsupervised monocular depth estimation with left-right consistency. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 270–279, 2017.

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[16] Pengcheng He, Xiaodong Liu, Jianfeng Gao, and Weizhu Chen. Deberta: Decoding-enhanced bert with disentangled attention. *arXiv preprint arXiv:2006.03654*, 2020.

[17] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[18] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[19] Hiroshi Inoue, Hiroshige Hayashizaki, Peng Wu, and Toshio Nakatani. Adaptive multi-level compilation in a trace-based java jit compiler. *ACM SIGPLAN Notices*, 47(10):179–194, 2012.

[20] Eunji Jeong, Sungwoo Cho, Gyeong-In Yu, Joo Seong Jeong, Dong-Jin Shin, and Byung-Gon Chun. JANUS: Fast and flexible deep learning via symbolic graph execution of imperative programs. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 453–468, 2019.

[21] Chao Jia, Yinfei Yang, Ye Xia, Yi-Ting Chen, Zarana Parekh, Hieu Pham, Quoc Le, Yun-Hsuan Sung, Zhen Li, and Tom Duerig. Scaling up visual and vision-language

representation learning with noisy text supervision. In *International conference on machine learning*, pages 4904–4916. PMLR, 2021.

[22] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. TASO: Optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 47–62, New York, NY, USA, 2019. Association for Computing Machinery.

[23] Taebum Kim, Eunji Jeong, Geon-Woo Kim, Yunmo Koo, Sehoon Kim, Gyeong-In Yu, and Byung-Gon Chun. Terra: Imperative-symbolic co-execution of imperative deep learning programs. *Advances in Neural Information Processing Systems*, 34, 2021.

[24] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, pages 1–6, 2015.

[25] Yanghao Li, Yuntao Chen, Naiyan Wang, and Zhaoxiang Zhang. Scale-aware trident networks for object detection. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 6054–6063, 2019.

[26] Nuno P Lopes. Torchy: A tracing jit compiler for pytorch. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*, pages 98–109, 2023.

[27] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 881–897, 2020.

[28] Dan Moldovan, James Decker, Fei Wang, Andrew Johnson, Brian Lee, Zachary Nado, D Sculley, Tiark Rompf, and Alexander B Wiltschko. Autograph: Imperative-style coding with graph-based performance. volume 1, pages 389–405, 2019.

[29] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. DNNFusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 883–898, 2021.

[30] Michael Paleczny, Christopher Vick, and Cliff Click. The java {HotSpot™} server compiler. In *Java (TM) Virtual Machine Research and Technology Symposium (JVM 01)*, 2001.

[31] James Reed, Zachary DeVito, Horace He, Ansley Ussery, and Jason Ansel. Torch. fx: Practical program capture and transformation for deep learning in python. *Proceedings of Machine Learning and Systems*, 4:638–651, 2022.

[32] Yining Shi, Zhi Yang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Ziming Miao, Yuxiao Guo, Fan Yang, and Lidong Zhou. Welder: Scheduling deep learning memory access via tile-graph. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 701–718, 2023.

[33] Alex Suhan, Davide Libenzi, Ailing Zhang, Parker Schuh, Brennan Saeta, Jie Young Sohn, and Denys Shabalin. Lazytensor: combining eager execution with domain-specific compilers. *arXiv preprint arXiv:2102.13267*, 2021.

[34] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 203–217, 1997.

[35] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*, pages 6105–6114. PMLR, 2019.

[36] Shizhi Tang, Jidong Zhai, Haojie Wang, Lin Jiang, Liyan Zheng, Zhenhao Yuan, and Chen Zhang. Freetensor: a free-form dsl with holistic optimizations for irregular tensor programs. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 872–887, 2022.

[37] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 10–19, 2019.

[38] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. PET: Optimizing tensor programs with partially equivalent transformations and automated corrections. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 37–54, 2021.

[39] Zuxuan Wu, Tushar Nagarajan, Abhishek Kumar, Steven Rennie, Larry S Davis, Kristen Grauman, and Rogerio Feris. Blockdrop: Dynamic inference paths in residual networks. In *Proceedings of the IEEE Conference on*

Computer Vision and Pattern Recognition, pages 8817–8826, 2018.

[40] Chen Zhang, Lingxiao Ma, Jilong Xue, Yining Shi, Ziming Miao, Fan Yang, Jidong Zhai, Zhi Yang, and Mao Yang. Cocktailer: Analyzing and optimizing dynamic control flow in deep learning. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 681–699, 2023.

[41] Jie Zhao, Siyuan Feng, Xiaoqiang Dan, Fei Liu, Chengke Wang, Sheng Yuan, Wenyuan Lv, and Qikai Xie. Effectively scheduling computational graphs of deep neural networks toward their {Domain-Specific} accelerators. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 719–737, 2023.

[42] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 863–879, 2020.

[43] Liyan Zheng, Haojie Wang, Jidong Zhai, Muyan Hu, Zixuan Ma, Tuowei Wang, Shuhong Huang, Xupeng Miao, Shizhi Tang, Kezhao Huang, et al. {EINNET}: Optimizing tensor programs with {Derivation-Based} transformations. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 739–755, 2023.

[44] Size Zheng, Renze Chen, Anjiang Wei, Yicheng Jin, Qin Han, Liqiang Lu, Bingyang Wu, Xiuhong Li, Shengen Yan, and Yun Liang. Amos: enabling automatic mapping for tensor computations on spatial accelerators with hardware abstraction. In *ISCA*, pages 874–887, 2022.

[45] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 859–873, New York, NY, USA, 2020. Association for Computing Machinery.

[46] Zhen Zheng, Xuanda Yang, Pengzhan Zhao, Guoping Long, Kai Zhu, Feiwen Zhu, Wenyi Zhao, Xiaoyong Liu, Jun Yang, Jidong Zhai, et al. Astitch: enabling a new multi-dimensional optimization space for memory-intensive ml training and inference on modern simt architectures. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 359–373, 2022.

[47] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, et al. ROLLER: Fast and efficient tensor compilation for deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 233–248, 2022.

A Artifact Appendix

Abstract

This artifact helps to reproduce the results of ATC'24 paper: MAGPY: Compiling Eager Mode DNN Programs by Monitoring Execution States.

Usage

The input of MAGPY is a PyTorch program. MAGPY automatically exports the PyTorch program to `torch.fx` format and adds proper guards with the proposed method described in the paper. Then, MAGPY uses the graph-level compiler provided by the user to compile the exported graph.

Scope

The artifact can be used to reproduce the experiments of the paper, including the introduction (Figure 1), end-to-end comparison (Figure 15), breakdown (Figure 16 and Table 2), dynamic models (Figure 17 and 18), and coverage (Table 3).

Contents

This artifact includes the code of MAGPY, input data of experiments, a guide for setting up the environment of the experiments, and scripts for running the experiments. It helps to reproduce the following Figures:

- Figure 1: Power of graph instantiation
- Figure 15: End-to-end inference on NVIDIA A100
- Figure 16: Time breakdown of execution (BS=1)
- Figure 17: Inference of models with dynamic shape
- Figure 18: Inference of models with dynamic control flow
- Table 2: Number of exported operator graphs
- Table 3: Result on ParityBench

Hosting

MAGPY is host at github.com/heheda12345/MagPy, and the artifact is host at github.com/heheda12345/MagPy-AE.

Requirements

This artifact needs one machine with at least one NVIDIA A100 GPU, with NVIDIA driver properly installed. Users can follow the installation guide to setup the software environment to reproduce the results.