PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation

Jason Ansel
Meta

Edward Yang
Meta

Horace He
Meta

Natalia Gimelshein
OpenAI

Animesh Jain
Meta

Michael Voznesensky
Meta

Bin Bao
Meta

Peter Bell
Quansight

David Berard
Meta

Evgeni Burovski
Quansight

Geeta Chauhan
Meta

Anjali Chourdia
Meta

Will Constable
Meta

Alban Desmaison
Meta

Zachary DeVito
Meta

Elias Ellison
Meta

Will Feng
Meta

Jiong Gong
Intel

Michael Gschwind
Meta

Brian Hirsh
Meta

Sherlock Huang
Meta

Kshiteej Kalambarkar
Quansight

Laurent Kirsch
Meta

Michael Lazos
Meta

Mario Lezcano
Quansight

Yanbo Liang
Meta

Jason Liang
Meta

Yinghai Lu
Meta

CK Luk
Meta

Bert Maher
Meta

Yunjie Pan
University of Michigan

Christian Puhrsch
Meta

Matthias Reso
Meta

Mark Saroufim
Meta

Marcos Yukio Siraichi
Quansight

Helen Suk
Meta

Michael Suo
Meta

Phil Tillet
OpenAI

Eikan Wang
Intel

Xiaodong Wang
Meta

William Wen
Meta

Shunting Zhang
Meta

Xu Zhao
Meta

Keren Zhou
OpenAI
George Mason University

Richard Zou
Meta

Ajit Mathews
Meta

Gregory Chanan
Meta

Peng Wu
Meta

Soumith Chintala
Meta

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0385-0/24/04
https://doi.org/10.1145/3620665.3640366

Abstract

This paper introduces two extensions to the popular PyTorch machine learning framework, TorchDynamo and TorchInductor, which implement the `torch.compile` feature released in PyTorch 2. TorchDynamo is a Python-level just-in-time (JIT) compiler that enables graph compilation in PyTorch programs without sacrificing the flexibility of Python. It achieves this by dynamically modifying Python bytecode

before execution and extracting sequences of PyTorch operations into an FX graph, which is then JIT compiled using one of many extensible backends. TorchInductor is the default compiler backend for TorchDynamo, which translates PyTorch programs into OpenAI's Triton for GPUs and C++ for CPUs. Results show that TorchDynamo is able to capture graphs more robustly than prior approaches while adding minimal overhead, and TorchInductor is able to provide a 2.27× inference and 1.41× training geometric mean speedup on an NVIDIA A100 GPU across 180+ real-world models, which outperforms six other compilers. These extensions provide a new way to apply optimizations through compilers in eager mode frameworks like PyTorch.

ACM Reference Format:
Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, CK Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Michael Suo, Phil Tillet, Eikan Wang, Xiaodong Wang, William Wen, Shunting Zhang, Xu Zhao, Keren Zhou, Richard Zou, Ajit Mathews, Gregory Chanan, Peng Wu, and Soumith Chintala. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3620665.3640366

1 Introduction

Modern machine learning frameworks can be divided into *eager mode* frameworks, such as PyTorch [32] and JAX [8], and *graph mode* frameworks, such as TensorFlow [1], Caffe [25], Theano [5], and CNTK [37]. Eager mode frameworks use an imperative *define-by-run* [47] approach where a machine learning model is represented as code that is executed each time one wants to run the model. Graph mode frameworks take a more declarative *define-and-run* [47] approach, where they expose a graph building API that requires users to first construct a graph and then later execute that graph.

Users of machine learning frameworks, and especially researchers, have shown an overwhelming preference for the eager programming model [22]. The eager mode model is easier to understand and can be debugged using standard tools such as `print` and `pdb` in Python [23]. This user preference towards eager mode has caused traditionally graph mode frameworks to switch to eager mode programming models [4].

The downside of eager mode frameworks is that they make it harder to apply graph-level optimizations through compilers. The framework only has visibility of a single operator at a time, and thus cannot automatically perform optimizations, like fusion or scheduling, that cross operator boundaries. To address this, there have been attempts to allow graph capture in PyTorch through record/replay [17, 34], Python parsing [17], and lazy evaluation [39]. Unfortunately, these approaches have sacrificed much of the usability that draws users to PyTorch. Record/replay is unsound and can produce incorrect behavior [17]. Python parsing works for simple programs, but has not been able to replicate the complex semantics of all of Python, so results will show it fails on over half of real-world models. Lazy evaluation incurs high run-time overheads and adds latency to kernel launches. Additionally, an exclusively graph mode backend for PyTorch is intractable for some models. Due to the flexibility provided by PyTorch, many model authors take advantage of features that do not easily map to graphs, such as: dictionaries, lists, custom classes, third party libraries (numpy, logging, etc), disk/network, multiprocessing, exceptions, and handwritten kernels.

This paper presents two open source extensions to PyTorch: *TorchDynamo* and *TorchInductor*. These extensions are behind the `torch.compile` feature introduced in PyTorch 2 and officially released in March 2023. TorchDynamo is a Python-level JIT compiler designed to allow graph compilation in PyTorch programs while retaining the full flexibility of Python. TorchDynamo hooks into the Python frame evaluation API [9] in CPython to dynamically modify Python bytecode right before it is executed. It rewrites Python bytecode in order to extract sequences of PyTorch operations into an FX graph [34] which is then just-in-time compiled with many extensible backends. It creates this FX graph through bytecode analysis and is designed to generate smaller graph fragments that can be mixed with Python execution to get the best of both worlds: usability and performance.

TorchInductor is a new compiler backend for TorchDynamo. It translates PyTorch programs into OpenAI's Triton [46] for GPUs and C++/OpenMP [15] for CPUs. TorchInductor is able to support the flexibility and dynamism of PyTorch by using similar abstractions to PyTorch eager mode. It introduces a new define-by-run loop-level intermediate representation (IR) to make it easy to add new operator lowerings. Additionally, it is implemented in Python, so it is easy for PyTorch users to extend and modify to meet their needs.

Experimental results show that TorchDynamo is able to capture graphs more robustly than prior approaches while adding minimal overhead. TorchDynamo is able to capture a single whole-program graph for most models and can gracefully fall back to partial graphs when needed. Measurements show TorchInductor produces faster code on average than six other PyTorch compiler backends. Performance comparisons include both training and inference, CPU and GPU, float32 and float16, and three large benchmark suites containing 180+ full-sized models taken from real-world applications.

2 Prior Attempts at PyTorch Graph Capture

Graph capture in PyTorch presents unique challenges when compared to graph mode frameworks [1, 25, 5, 37], where the user is restricted to only using constructs that are representable in the graph. With PyTorch and other eager mode frameworks, the user is free to embed arbitrary code, including non-PyTorch libraries, inside their models. This results in frequent conversion from PyTorch Tensors to Python types (via `.item()`, `.tolist()`, etc), usage of external libraries (numpy, logging, etc), and usage of Python constructs (classes, closures, exceptions, control flow, etc) that do not map well to a fixed graph abstraction. Due to this mismatch between the flexibility provided by Python/PyTorch, and the inflexibility of graph representations, prior attempts at graph capture in PyTorch have needed to place restrictions on the user experience. While this tension between flexibility and representation is solved by TorchDynamo, we examine prior art in the space to provide context and background.

2.1 torch.jit.trace

`torch.jit.trace` uses record/replay with example inputs to produce a TorchScript [17] graph. The recording is done at the PyTorch dispatcher level, which is inside the C++ portion of PyTorch and used to dispatch operators to device-specific kernels and for autograd. Because the recording is done in C++, `torch.jit.trace` does not capture any control flow in Python. Consider this example:

```
def example1(x):
    if len(torch.nonzero(x)) > 1:
       return x + 1
    return x - 1
```

With example input `torch.tensor([0, 0])`, `torch.jit.trace` would capture a graph equivalent to:

```
def example1_incorrect_capture(x):
    torch.nonzero(x)
    return x - 1
```

Since the path through the program is specialized on the example input, a different input (such as `torch.tensor([1,1])`) will give incorrect results. Additionally, any non-PyTorch operators (such as external libraries, prints, logging, side effects, etc.) will be omitted from the captured graph.

2.2 torch.jit.script

`torch.jit.script` also constructs a TorchScript [17] graph, but does so by parsing the Python AST and performing static analysis. It is able to capture `example1` above correctly and, unlike `torch.jit.trace`, it is a sound approach that should not produce incorrect results.

The major challenge `torch.jit.script` faces is that it is trying to reimplement all of Python as a static language. This approach is all or nothing: encountering an unimplemented component of Python makes the entire program unfit for capture. Emulating all of Python statically is a daunting task and, in practice, `torch.jit.script` only supports a subset of Python. Experimental results show that `torch.jit.script` works only about half the time on real-world models in the TorchBench benchmark suite, and anecdotally we have heard stories of it taking weeks or months to "torchscript" large models, which leads to a frustrating user experience.

2.3 Lazy Tensors

Lazy Tensors were introduced in the PyTorch/XLA [42, 39] project, which is primarily focused on supporting Google TPUs [26] with PyTorch. Lazy Tensors is a C++ level graph capture technology. Every iteration, it defers execution of operations to accumulate a graph and then sends the accumulated graph to the XLA [45] compiler. By hashing this graph, Lazy Tensors can avoid recompilation when the captured graph is identical across iterations. While this approach is effective and sound, it has a few major downsides:

- *Higher overheads*: Lazy Tensors incurs additional work when compared to PyTorch eager. Besides running the same Python code and PyTorch dispatcher stack that eager does, it must maintain additional graph data structures that incur added runtime costs.
- *Introduced delays*: PyTorch eager issues the first kernel on the first operation of the model, after which point host-side code is run in parallel with kernels on the GPU or accelerator thus hiding overheads. In contrast, Lazy Tensors doesn't issue the first kernel until the model's code has finished executing, resulting in added delays before the first kernel is issued and after any operation that requires a round trip to the CPU (which are common in real-world models). Thus, Lazy Tensors often serializes host execution with GPU/accelerator utilization, which amplifies host side overheads. Models, loss logging, and optimizers need to be modified to work around this issue.
- *Recompilation*: Whenever the captured graph has a new hash, Lazy Tensors must recompile. This can lead to some pathological cases where recompilation happens frequently.

The PyTorch/XLA project has built [10] an integration with TorchDynamo which uses a hybrid of both Lazy Tensors and TorchDynamo. This integration hides the overheads of Lazy Tensors by only running Lazy Tensors once, rather than every iteration, and using TorchDynamo to figure out when recapture is needed. The PyTorch/XLA results later in the paper use this integration.

2.4 torch.fx.symbolic_trace

`torch.fx.symbolic_trace` [34] is the newest of these systems and introduced the FX graph format that is shared by TorchDynamo. It takes a similar record/replay-based approach to `torch.jit.trace`, but does its tracing at the Python level as opposed to at the PyTorch C++ dispatcher level. It runs the

user code using a *Proxy* Python object to record its behavior and uses the *torch_function* [3] extension point in PyTorch.

By recording higher up at the Python level, `symbolic_trace` is able to capture many operations that `torch.jit.trace` cannot. Since it records using Proxy objects instead of real tensors, it is able to detect many cases where `torch.jit.trace` would be incorrect, e.g., when trying to read sizes or values from Proxy tensors or when using them in control flow, such as `example1` above. It also suffers from the all-or-nothing limitation of many solutions described above. For example, in the control flow case above, the user is still forced to rewrite the code they want to trace.

Unfortunately, `torch.fx.symbolic_trace` is still unsound and can produce incorrect results. Consider this example, which increments a global variable and calls a function not dependent on the function input:

```
def example3(x):
    global call_count
    call_count += 1
    return torch.rand(10) + x
```

If one runs `torch.fx.symbolic_trace` on this example it produces a graph equivalent to:

```
def example3_incorrect_capture(x):
    return _tensor_constant0 + x
```

The call to `torch.rand` got removed and the result of it got burned into the graph as a fixed constant. Subsequent uses of the graph will not get new randomness, but instead reuse whatever value was generated during capture. This type of incorrect capture can be difficult to debug and may go unnoticed by users.

The `call_count` operations are completely lost because they did not interact with the Proxy object `x`. Instead, the `call_count` got incremented to 1 during tracing and will not be incremented when the graph is called. This is also an example of something that is not supported by any of the graph representations. Nearly all graphs formats for machine learning have no concept of a Python global, so even if this could be captured, it is not supported by downstream backend compilers.

2.5 torch.onnx.export

ONNX [31] export is not actually a graph capture mechanism, but some people confuse it for one, so we include it here for completeness. Internally, ONNX export uses `torch.jit.trace` and `torch.jit.script` (Section 2.1 and 2.2), so it faces all the same limitations imposed by those systems. Additionally, the conversion from TorchScript to the ONNX format can fail due to ONNX not supporting all PyTorch operators. Thus, the set of models supported by ONNX is a subset of those supported by TorchScript.

The ONNX team is working on an integration with TorchDynamo that will replace TorchScript with a direct TorchDynamo integration. Once finished, this will increase the number of models ONNX works on.

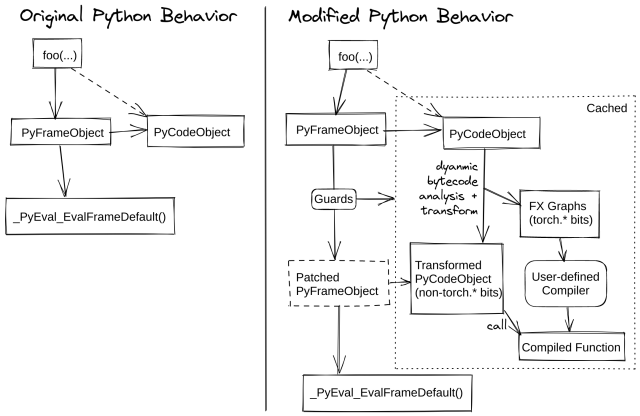

Figure 1. Overview of how TorchDynamo modifies the CPython interpreter to capture FX graphs

2.6 Comparison To Graph Capture In JAX

JAX [8] largely doesn't face the same challenges being solved by TorchDynamo. The initial design of JAX was heavily coupled to the design of XLA [45], and JAX has been backed by XLA from its inception. This has the effect of forcing JAX programs to conform to the constraints built into the design coming up the stack from XLA. Thus, JAX uses a simpler capture mechanism, and expects users to write their programs to meet the constraints of that capture mechanism. As an example, `jax.jit` does not support data-dependent Python control flow and requires user code to be functionally pure.

In contrast, PyTorch started as an eager-only framework without any compiler-minded constraints built into its design. A large corpus of models has grown on top of PyTorch, most of which were written without any regard to how hard to capture and compile they would be.

On an implementation level, the capture mechanism in JAX is similar to `torch.fx.symbolic_trace` (Section 2.4), but somewhat simpler, because JAX programs are purely functional and thus do not need to worry about state. The Torch FX paper [34] contains a more detailed comparison with JAX.

3 TorchDynamo Design and Implementation

TorchDynamo takes a fundamentally different approach from prior graph capture systems in PyTorch. Rather than trying to remove or replace Python, TorchDynamo tries to work with CPython by just-in-time (JIT) compiling Python bytecode. TorchDynamo is a Python bytecode to Python bytecode translator, where it extracts PyTorch operations from the original bytecode and replaces them with calls to compiled artifacts that fuse many PyTorch operations together. Figure 1 provides an overview of how TorchDynamo

operates and will be explained in the remainder of this section.

3.1 Usage API

The primary API introduced in this paper is `torch.compile`. It can be used either by calling it on a PyTorch Module or as a function decorator. It has the following keyword options:

- **backend**: allows the user to provide a custom compile function which takes a `torch.fx.Graph` and a list of example inputs and returns a Python callable. This defaults to TorchInductor, but can also be set to one of many builtin backends or a user-defined backend.
- **options**: An optional dictionary of backend-specific configuration flags.
- **mode**: Shorthand strings for a predefined set of options: "default", "reduce-overhead", or "max-autotune".

When you run a module with `torch.compile`, the module is executed with the modified CPython behavior shown in Figure 1. Specifically, a custom CPython frame evaluation hook will rewrite the bytecode of of each Python function being executed in order to extract and compile sequences of PyTorch operations. This bytecode rewriting process is cached, but the analysis relies on certain dynamic properties of the program that we use guards to check on subsequent calls.

3.2 CPython Frame Evaluation Hook

PEP 523 [9] introduced the frame evaluation API into the CPython interpreter. A frame is the data structure in CPython used to represent a function call. This is the main extension point used by TorchDynamo, and it was designed to facilitate just in time (JIT) compilers and debuggers in Python. PEP 523 added an `eval_frame` function pointer to `PyInterpreterState`, which allows overriding the core function used to interpret a single function call in CPython. Whenever CPython calls a function, it first creates a `PyFrameObject`, then it calls this user defined `eval_frame` hook. By default, `eval_frame` points to `_PyEval_EvalFrameDefault`, which contains the main interpreter loop for CPython. TorchDynamo modifies `eval_frame` to replace this standard CPython interpreter loop with one that performs JIT compilation of Python frames.

The custom eval frame function installed by TorchDynamo performs the following operations:

- Check if the frame should be skipped due to filename exclusion, previous failures in analysis (which mark the frame to be skipped), or exceeded cache size limits. Filename exclusions are used for common libraries, like Python standard libraries and numpy, which will not contain PyTorch operations. For skipped files, call `_PyEval_EvalFrameDefault` on the original bytecode and return.
- Check if the frame has previously been compiled and is cached; if so, execute the generated guard function (Section 3.3) for each entry in the cache. If a guard function returns `True`, run the matching cached compiled bytecode with `_PyEval_EvalFrameDefault` and return.
- Perform symbolic analysis (instruction by instruction) of the function bytecode to extract an FX graph [34], guards, and side effects. This analysis can stop partway through the function if it encounters an unsupported operation.
- Compile the FX graph with a user-defined compiler function specified by the `backend=` argument provided to `torch.compile`.
- Generate and compile a single Python function that checks all of the guards. It returns `True` if the guards pass and the existing compiled artifact can be reused.
- If the analysis did not reach the end of the function, generate `resume_at_XX` continuation functions. Continuation functions run the remainder of the function in a new frame and are described in Section 3.8.
- Generate new Python bytecode. This new bytecode will: 1) call the compiled FX graph; 2) store and reconstruct the local/stack state; 3) perform side effects the original function should have had, see Section 3.7; 4) either return or implement a graph break by falling back to the original bytecode and calling the generated continuation function(s).
- Install the generated Python bytecode and guard function in the cache, run the generated bytecode with `_PyEval_EvalFrameDefault`, and return.

3.3 Guards

Guards are the mechanism TorchDynamo uses to recheck dynamic properties used by JIT compilation to determine is a cached compilation can be reused. TorchDynamo generates a guard function for each transformed `PyCodeObject` that returns `True` if it is safe to reuse a compiled artifact. Both the guards and the transformed code are stored using the `_PyCode_SetExtra` extension point introduced in PEP 523 [9].

Guards are accumulated during analysis and can point to variables originating from globals/locals or nested within Python data structures. At the time of writing there were 30 different types of guards. Guards include: checking many `torch.Tensor` properties, Python types, constant specialization, attributes, dicts/lists/tuples, `nn.Module` instances, and global PyTorch state. The guard system spans across TorchDynamo, AOTAutograd, and TorchInductor. Any layer can introduce guards to protect specializations. Guards are all independent checks and do not interact with each other beyond deduplication.

3.4 Symbolic Evaluation

A fundamental part of TorchDynamo is the symbolic Python bytecode evaluator which is responsible for analyzing Python bytecode and modeling effects of each instruction. Symbolic evaluation contains data structures that keep track of: 1)

stack state; 2) local variables; 3) exception contexts; 4) accumulated FX graph [34]; 5) accumulated guards; and 6) side effects. The algorithm operates one Python bytecode at a time and contains a function corresponding to every Python bytecode instruction type.

At the start of symbolic evaluation, function arguments are examined and converted to a symbolic representation, `VariableTracker`. If bytecodes access data structures such as class attributes or global variables, new symbolic representations for these constructs are added lazily. This representation is discussed more in Section 3.5. The symbolic evaluator starts at the first bytecode instruction of the function, and continues processing the function one bytecode at a time. The soundness of this analysis can be shown via induction: as long each individual bytecode is processed correctly, the overall algorithm will be correct.

As an example, suppose the first instruction was `LOAD_FAST`, a Python bytecode that pushes a local variable on to the stack. The handler for `LOAD_FAST` will take the representation variable from the symbolic local variables and push it on to the symbolic stack data structure. The handler for `BINARY_ADD`, will pop two symbolic variables off the stack then push their result on to the stack. The result is computed depending on the types of those variables and the dispatch will vary based on those types. If the value represents a PyTorch tensor, then a new `add` node will be added to the FX graph [34], and a new symbolic tensor pointing to the result node will be created.

3.5 Modeling Python Data Structures

Many semantics of Python are in libraries and data structures, so any Python analysis must model the behavior of these different types. To analyze the behavior of each variable or stack entry, TorchDynamo has a class hierarchy that models common behaviors of different data types. Each of these data structures is a subclass of `VariableTracker`. Notable types of variable trackers include:

- `TensorVariable` represents a `torch.Tensor`. It does not store an underlying tensor value, but instead stores a `fx.Proxy` which points into the partially constructed FX graph [34] as well as a "fake" tensor (see Section 5) that represents the metadata of a tensor without its actual data.
- `ConstDictVariable` and `DataClassVariable` are used to represent key/value pairs where the keys are constant strings and the values can be anything, including nested dicts/lists.
- `ListVariable` and `TupleVariable` represent list/tuple and can contain any other type of symbolic variable.
- `UserFunctionVariable` and `UserMethodVariable` represent user defined functions that can be inlined. They also support functions constructed dynamically containing closures.
- `UserDefinedClassVariable` represent user-defined classes and `UserDefinedObjectVariable` represents instances. We lazily specialize on these as their attributes are accessed, and track mutation on them (Section 3.7).

There are many other variable tracker types that represent other situations. In addition to type-specific data, every `VariableTracker` instance also contains a set of guards, which are initialized when they are created and propagated through operations via union. Additionally, each instance also tracks where it came from so that it can be loaded or mutated in output bytecode.

3.6 Inlining, Control Flow, and Closures

Function calls can either happen directly from user code, or implicitly through *magic methods* such as `__getitem__`. To collect bigger graphs, TorchDynamo will attempt to inline function calls and flatten programs. When a function call is encountered, TorchDynamo first creates a checkpoint of the current symbolic state. Next, it recursively tries to symbolically evaluate the called functions, passing in any input symbolic state and recording any changes that are made. If this recursive analysis hits a case that would cause a graph break (Section 3.8) or other errors, TorchDynamo rolls back to the symbolic state before the function call and generates a graph break on that function call. Otherwise, the recursive analysis returns and the analysis of the parent function continues.

Most cases of control flow in Python bytecode are optimized away and handled through specialization. For example, when iterating over a list of `torch.nn.Module`, TorchDynamo will guard that the list doesn't change and unroll the loop. For control flow based on the type, size, and shape of tensors, TorchDynamo will guard on those properties and remove the control flow. In less common cases where there is control flow that cannot be removed (for example, branching on the value of a tensor rather than the metadata), TorchDynamo will generate a graph break that will trigger the branch bytecode to run in CPython, and analysis will resume after the jump.

Another challenge is closures. Consider this example:

```
def closure_example(x):
    y = torch.sigmoid(x)
    return lambda z: y + z
```

Here the variable y is in a closure which is represented by what CPython calls a *cell*, which adds a layer of indirection to allow variables in closures to be mutated. There are a number of different cases of closures that TorchDynamo must handle:

- Cell variables created outside the captured region must be accessed differently than other variables. If they are accessed from the top-level function, they can be accessed by generating the `LOAD_DEREF` and `STORE_DEREF` bytecodes. When inlining, this bytecode cannot be

used and instead TorchDynamo generates code to read-/write directly from the inlined function cell, for example `fn.__closure__[0].cell_contents`. If the content of a cell is mutated, TorchDynamo tracks the mutation in the same way as other mutations (Section 3.7).
- Cell variables both created and destroyed within the captured region are the easiest to handle and most common. In this case, TorchDynamo statically optimizes away the closure.
- Cell variables that are created in the captured region, but escape the frame are the most difficult to handle. In this case, TorchDynamo will optimize away all uses of the closure inside the captured region. Then, at the very end in the generated bytecode, it will create any needed cells and Python function objects to return. From the outside, callers will not be able to tell that the returned closure was created differently than the original program.

3.7 Mutation and Side Effects

Python functions sometimes have side effects. TorchDynamo handles side effects by deferring them until after the FX graph [34] has been called, then generating output bytecode that applies all side effects at the end. To do this, TorchDynamo has a side effects data structure that tracks all side effects that the original code would have. If the code tries to read a value that would have been mutated by a pending side effect, it instead reads that pending value. After the graph is generated, a garbage collection pass removes side effects that didn't escape the analysis context, and TorchDynamo generates output code to apply the needed side effects. Handling side effects this way results in multiple writes to the same value being collapsed into a single write. TorchDynamo supports the following types of side effects:

- Writes to global variables result in a `STORE_GLOBAL` bytecode if the target global is in the same file. If it is in a different file (because of inlining), code is generated to mutate the global in the other module.
- Writes to attributes (such as on classes) are handled similarly and mapped to `STORE_ATTR` in output bytecodes. We use the source on the VariableTracker to determine how to load a reference to the object that must be mutated.
- Writes to cells/closures are tracked and handled in a number of ways (see Section 3.6).
- Class construction is handled by creating a placeholder symbolic object, inlining the `__init__` method, and tracking all the attribute mutation on that placeholder object. If the object is live at the end of the function, the output bytecode will create the object (bypassing the constructor) and set the needed attributes.
- Dictionary and list mutation can also cause side effect if the dict/list was passed in as an input or loaded from a global/attribute. The `VariableTracker` representations of dict/lists will guard on the initial symbolic state of these objects, then symbolically track all changes through the entire function. The captured FX graph [34] will have all of these operations optimized away. In the output bytecode, a new dict/list will be created to match the final state and the original list object will be mutated to match that object. This recreation is not needed for lists/dicts that do not escape the captured region because their mutations cannot be observed, and therefore they can be completely removed.

3.8 Graph Breaks and Continuation Functions

When TorchDynamo encounters a Python bytecode it cannot handle, for example a call to an external library, it generates what we call a graph break to split the bytecode being analyzed into multiple pieces. Essentially, TorchDynamo will mix compiled fragments into the original Python code to get a hybrid execution. Any pending partial FX graph [34] is compiled. In the output code when the partial graph will be called, the unsupported bytecode will be executed, and then we will recursively use TorchDynamo to analyze the remainder of the function. To trigger this recursive analysis, TorchDynamo generates one or more continuation functions which take the form:

```
def resume_at_X(... livevars ...):
  ... restore try/except/stack state ...
  JUMP_ABSOLUTE X
  ... original function bytecode ...
```

This continuation function looks very similar to the original function except for a few changes: 1) the arguments are changed to reflect whatever variables are live across the graph break; 2) a prefix is added to restore the stack/exception state, which may also be passed in as an argument; 3) a `JUMP_ABSOLUTE` instruction is created so execution resumes in the middle of the function.

TorchDynamo will either generate one of these functions, or two of these functions in the case of control flow (all control flow bytecodes have exactly two branches), to continue execution right after the unsupported bytecode. The advantage of structuring continuations as Python functions is that it will recursively trigger TorchDynamo through the frame evaluation API. When TorchDynamo processes a continuation function, it treats it exactly the same as any other Python function.

3.9 AOTAutograd

AOTAutograd is a reusable component in PyTorch that is called by many PyTorch compiler backends to add training support and use shared operator decompositions. TorchDynamo captures the forwards of a model, but, to support training, we also need to generate the backwards pass. In Pytorch eager, the backwards graph is generated dynamically using a tape-based autograd [32]. AOTAutograd turns the forwards

graph into a forwards and backwards graph in a way that supports partial program graphs.

AOTAutograd works by running the PyTorch eager mode autograd engine on fake tensor inputs and recording a joint forwards and backwards graph. Data-dependent operations do not work with fake tensors (since there is no backing data), so we graph break on these operations in TorchDynamo and run them outside the graph. AOTAutograd then uses a min-cut algorithm [55] to split this joint graph into separate forward and backward graphs in a way that optimizes for memory usage. As part of this min-cut algorithm, we apply backend-specific optimizations to rematerialize certain activations that are cheap to recompute in the backwards graph.

As part of AOTAutograd, other dispatcher-level transformations are also applied to the graph. Decompositions are where AOTAutograd maps some PyTorch operators into a smaller set of more primitive operators. AOTAutograd also makes the graph purely functional by removing operations that perform mutation and replacing them with their functional equivalents.

4 TorchInductor Design and Implementation

While TorchDynamo solves the graph capture problem in PyTorch, to be useful it must be paired with a backend compiler that can take the captured FX graph [34] and generate fast code from it. We created TorchInductor as a reference compiler backend. It is designed to be general purpose and can be used both directly by users and as a starting point for other backends.

4.1 Design Principles and Key Technologies

Before diving into the design of TorchInductor, let's first discuss some principles and technologies that motivated its design:

PyTorch Native: PyTorch made many design choices that differ from other frameworks and compilers: Tensors have exposed strides that can be manipulated by users, aliasing views are commonplace, and both data and metadata can be mutated in-place. Any compiler with a dramatically different model will face many challenges in representing PyTorch programs. We wanted TorchInductor to share similar abstractions to PyTorch eager to allow support of all of PyTorch, with a thin translation layer.

Python First: The majority of PyTorch users are most comfortable in Python. The Python parts of PyTorch get far more community contribution than the C++ parts of PyTorch. We chose to implement TorchInductor in Python to make it easy to understand and hackable by PyTorch users.

Breadth First: Rather than focusing on a narrow set of models (e.g. ResNet/BERT) that are already well studied, we intentionally put an early focus on supporting a wide variety of operators, hardware, and optimization. This helped make TorchInductor a general purpose compiler that can scale to many scenarios. This is also why the early focus was on training, since training is a much harder compiler problem than inference.

Reuse State-Of-The-Art Languages: For an output language, we took inspiration from how PyTorch users were writing high performance kernels. We observed rapidly increasing popularity of the OpenAI Triton [46] DSL for writing GPU kernels, and those kernels are often outperforming other compilers and state-of-the-art libraries. High performance CPU kernels are typically written in C++/OpenMP [15]. TorchInductor generates both Triton and C++ as output code, which allows us to leverage the technology of those projects as well as generate output code that is understandable by PyTorch users.

4.2 Decompositions

Rather than implementing lowerings for all operators in PyTorch to TorchInductor's IR, many operators in PyTorch are decomposed into a simpler set of operators that are easier to handle. These decompositions happen using AOTAutograd (Section 3.9), which is called by TorchInductor with a dictionary of desired decompositions. Decompositions are written as a Python implementation of a PyTorch operator in terms of other operators, for example the following decomposes `log2` into `log` and `mul`:

```
log2_scale = 1 / math.log(2)

@register_decomposition(torch.ops.aten.log2)
def log2(x):
  return torch.log(x) * log2_scale
```

This decomposition will be recursively traced down and normalized, and can possibly trigger additional decompositions in that process until a fixed point is reached. Note that the active decomposition set must not contain cycles.

At the time of writing, TorchInductor used 191 decompositions (387 including overloads). The majority of these decompositions are not specific to TorchInductor and are available for any other backend to use via the `torch._decomp` module, while some are TorchInductor specific.

4.3 Lowerings and Define-By-Run Loop-Level IR

The next phase of compilation is lowering from an FX graph of PyTorch operations into TorchInductor's define-by-run IR. A define-by-run IR means the IR uses executable Python code to define the bodies of loops, giving TorchInductor's IR much of the power of full Python, removing the need for a large amount of boilerplate, and allowing lowerings to be written concisely.

```
def inner_fn_buf0(index):
  i0, i1 = index
  tmp0 = ops.load("arg0_1", i0 * s1 + i1)
  tmp1 = ops.log(tmp0)
  tmp2 = ops.constant(1.4426950408889634, torch.float32)
  tmp3 = ops.mul(tmp1, tmp2)
  return tmp3

buf0_ir = TensorBox(StorageBox(ComputedBuffer(
  name='buf0',
  layout=FixedLayout('cuda', torch.float32,
                     size=[s0, s1], stride=[s1, 1]),
  data=Pointwise(inner_fn=inner_fn_buf0,
                 ranges=[s0, s1], ...))))
```

Figure 2. TorchInductor IR for `torch.log2` on a 2D tensor.

Lowering is done by symbolically interpreting the FX graph and applying lowering functions which do the conversion for a single operator. At the time of writing, TorchInductor has lowerings for 433 PyTorch operators (1605 including overloads). If an unknown operator is encountered, it is automatically converted into a fallback kernel node which runs the original PyTorch code.

In the example IR shown in Figure 2, `inner_fn_buf0` is a Python function that defines how to compute a single element of the tensor `buf0` in terms of calls to TorchInductor's primitive operators in the `ops.*` namespace. The function takes a list of SymPy [28] symbols (`i0` and `i1`) representing the symbolic coordinates of the element to be computed. SymPy symbols `s0` and `s1` represent the sizes of the tensor to be computed and are used for both sizes and strides. These size symbols are captured in a Python closure and registered on the graph object.

`TensorBox` and `StorageBox` are abstractions that match PyTorch `torch.Tensor` and `torch.Storage` objects and allow the handling of views, aliasing, and mutation during the lowering process. `ComputedBuffer` represents a tensor that will be computed using generated code (in contrast to ones created via fallback kernels or inputs). `Pointwise` represents that the `ComputedBuffer` is a data parallel pointwise computation. The IR also supports `Reduction` and `Scatter` for handling other types of operators.

The key advantage of this IR is that it is easy to construct because it has the full power of Python. One can compose different IR nodes together and embed logic within them. The example above would not be initially constructed as a single flat function, but rather many smaller function closures defined in the lowering process. The function created for `ops.mul` will call into another function created for `ops.log`, which calls into another function created for loading the input argument.

The way to compile and analyze this IR rests in the virtualized namespace for `ops.*`, which can be dynamically overridden to perform different functions. To perform analysis on this IR, we make `ops` point to an analysis pass which can perform actions like record memory accesses or high/low watermarks for strength reduction optimizations. To perform codegen with this IR, we make `ops` point to something which writes out Triton or C++ code. To transform this IR, we make use of FX tracing, which gives access to graph representation for these Python functions.

At the time of writing, the loop-level TorchInductor IR consisted of 54 primitive operators:

- *ops.load* and *ops.store* access Tensor memory from a provided buffer name and a SymPy index specifying a symbolic memory location.
- *ops.reduction* operates like *ops.store* where the reduction happens implictly inside the write. It combines stored values along the reduction dimension of the current node using a supplied reduction type. Supported reduction types are: argmin, argmax, any, max, min, prod, sum, xor_sum, and welford_combine [50].
- *ops.index_expr* converts from SymPy expressions used for indexing into values used for compute.
- *ops.indirect_indexing* converts from computed values into SymPy expressions used for indexing by introducing a new SymPy variable bound dynamically.
- *ops.masked* implements conditional execution. It takes a condition and a Python function (recursively using the same IR) with no args. This gets mapped to masks in Triton and conditionals in C++.
- *ops.load_seed*, *ops.rand*, *ops.randn*, and *ops.randint64* are used for computing random numbers.
- The remaining ops are elementwise math operations.

4.4 Scheduling

The scheduling phase of TorchInductor determines which operators get fused, what order kernels run in, and does memory planning for buffer removal and/or reuse. Scheduling starts by converting every buffer in the IR into a subclass of `BaseSchedulerNode`. `SchedulerNode` represents a standard kernel that TorchInductor will codegen the body of. `ExternKernelSchedulerNode` represents calls to library code or user-defined kernels. Additionally, `NopKernelSchedulerNode` maps to nothing, but is used to add dependency edges to ensure the ordering of kernels (for example, a *concatenate* kernel which has been handled by making producers write directly to the combined buffer). Finally, a `FusedSchedulerNode` represents a set of two or more `SchedulerNodes` fused into a single kernel.

Next, the scheduler converts the memory read/write sets of each kernel into dependency edges between nodes. Dependency edges are annotated with the symbolic memory address being read. Symbolic memory addresses are important in determining which fusions are legal. For example, if one kernel writes `buf0` in forwards order, but a consumer reads in reverse order (using `ops.load("buf0", s0 - 1 - i0)`), then those nodes cannot be fused.

Fusion is controlled by two key functions:

- `Scheduler.can_fuse(node1, node2)` returns `True` if two nodes can be fused together. This checks dependency edges, and also checks many other properties to ensure correctness of a fusion. There are some heuristics here as well, for example, if `config.aggressive_fusion=False`, then `can_fuse` will prevent fusion of nodes that do not share any common memory accesses. There is also backend specific logic here, for example, TorchInductor supports reduction-broadcast-reduction fusions for Triton but not C++.
- `Scheduler.score_fusion(node1, node2)` is used to order different fusion possibilities. Some fusions are mutually exclusive, so TorchInductor chooses the one with the higher score. The fusion score orders fusions by: 1) the category of the fusion (e.g. pointwise/reduction/template); 2) estimated bytes of memory traffic saved by the fusion; and 3) shorter distance between nodes in the original graph

In a loop, until no additional fusions remain (since some fusions can open additional fusion opportunities), TorchInductor will perform the following greedy algorithm: 1) find all fusion opportunities; 2) score each of the fusion opportunities and sort by that score; 3) for each fusion opportunity, check if that fusion remains legal and if so apply it. When two nodes are fused, any pending fusion opportunities pointing to the constituent nodes are updated to point to the new fused node.

4.5 Triton Code Generation

Triton codegen is responsible for mapping TorchInductor's IR to output Triton [46] kernels. Figure 3 shows the code generated for the `log2` example above. This kernel operates on a block of `XBLOCK` elements at a time. If the number of elements is not a multiple of `XBLOCK`, some elements may be masked off at the end. During codegen, we simplify indexing. For example, the 2D strided load in the IR is converted to a contiguous load in this case. Codegen is also responsible for common subexpression elimination (CSE), which is done via a cache while printing lines of code and assigning intermediate variable names starting with `tmp`. The `pointwise` decorator encodes boilerplate code used to facilitate block size heuristics, auto-tuning, and ahead-of-time kernel compilation. The decorator is the type of kernel being generated (pointwise, reduction, or template), and its arguments are required metadata about the kernel like data alignments.

When generating reduction kernels, TorchInductor has two modes of codegen. For smaller reductions, it will generate a *persistent reduction* where the entire reduction is loaded in a single block and retained in registers/shared memory; in this case reductions map directly to Triton reduction operators. For larger reductions, TorchInductor generates a loop using an entire block as an accumulator with a call to a Triton reduction at the end of the loop.

```
@pointwise(...)
@triton.jit
def kernel(in_ptr0, out_ptr0, xnumel, XBLOCK : tl.constexpr):
    xoffset = tl.program_id(0) * XBLOCK
    xindex = xoffset + tl.arange(0, XBLOCK)[:]
    xmask = xindex < xnumel
    x0 = xindex
    tmp0 = tl.load(in_ptr0 + x0, xmask)
    tmp1 = tl.log(tmp0)
    tmp2 = 1.4426950408889634
    tmp3 = tmp1 * tmp2
    tl.store(out_ptr0 + x0, tmp3, xmask)
```

Figure 3. Generated Triton code for Figure 2.

For more complex operations (matmuls and convolutions), TorchInductor has its own template system for generating Triton code that mixes handwritten Triton with generated Triton. Templates are written using Jinja [29] with helper methods to interact with TorchInductor's codegen system.

4.6 C++ Code Generation

For the CPU backend, TorchInductor generates C++ with OpenMP [15]. Within the C++ backend there are two variants, a vectorized variant and a non-vectorized variant. The vectorized variant performs tiling and maps most operations to the `at::vec::Vectorized` class included in the PyTorch source code. This class operates on 16 elements at a time, which is the same way standard PyTorch kernels are vectorized and supports multiple SIMD instruction sets. The non-vectorized variant generates relatively standard C++ code using many C++ standard template library [24] (STL) functions. Both of these variants are parallelized using `#pragma omp for` annotations, with some heuristics to decide how many levels of loops to parallelize. Reductions are mapped to the OpenMP `reduction` annotation if the reduction dimension loop is parallelized, and a C++ loop with accumulator otherwise.

4.7 Wrapper Codegen

Wrapper codegen is responsible for generating the code that calls the kernels from Triton, C++, and external sources. It also does tensor size calculations and handles memory allocation and deallocation. There are two different wrapper codegen implementations, one that generates Python code, and another that generates C++ code. The Python backend is more flexible and supports some corner cases that the C++ one does not, while the C++ one is lower overhead.

When enabled with `mode="reduce-overhead"`, TorchInductor uses CUDA Graphs [20] to completely eliminate the overhead from wrapper code. CUDA Graphs records and replays kernel launches at the CUDA driver level and is lower overhead than even the C++ wrapper code. To ensure soundness, CUDA Graphs is only used when safety requirements are

met and is automatically disabled in some cases (for example with dynamic shapes, non-CUDA tensors, etc).

4.8 Related Deep Learning Compilers

There is lots of exciting work in the deep learning compiler space. Since most PyTorch users use GPUs, our main reason for selecting Triton [46] as an output target was its proven ability to generate kernels faster than handwritten libraries [30, 13, 43] with simple input code. Very few compilers have been able to do that consistently, and many widely used deep learning compilers simply call those libraries directly without trying to compete in GPU codegen for complex kernels.

Many compilers use designs inspired by Halide [33], including: TVM [12], nvFuser [36], and NNC [60]. These designs have a split semantics language and scheduling language that allow exploring different schedules without changing the semantics of the program. Researchers have explored many different ways of expressing the search space [18, 38, 48, 51, 58, 7, 59, 19] and searching that space automatically [57, 11, 54, 56, 6].

XLA [45] is the compiler behind TensorFlow [2] and JAX [8]. XLA provides multiple levels of IR including a high level IR, HLO, that has become a standard for TPUs [26] and similar accelerators. Many newer compilers are emerging in the MLIR [27] ecosystem, including IREE [44] (now part of OpenXLA [45]). The latest version of Triton [46] also uses MLIR for its internal representation.

5 Dynamic Shapes

Deep learning compilers commonly only work for static shapes, that is to say, they produce compiled programs which only work for a single specific configuration of input shapes, and must recompile if any input shape changes. This assumption works well for the majority of commonly run deep learning models today, but there are a few situations where it is insufficient:

- Some dimensions, such as batch size or sequence length, may vary. For example, an inference service performing adaptive batching will execute inference requests with varying batch sizes depending on how many requests it received within its batching window. We may also want to consider padding out variable-size sequences only to the maximum sequence length within a batch, which may vary from batch to batch.
- Some models exhibit *data-dependent output shapes*, that is to say, the size of their outputs and intermediates may depend on the actual input data which may vary across runs. For example, detection models may first generate a variable number of potential bounding boxes before running a more expensive image recognition model to identify if the subject is in a bounding box. The number of bounding boxes is data-dependent.
- One particularly important case of data-dependent shapes occurs when dealing with sparse representations, such as sparse tensors, jagged tensors, and graph neural networks. In all of these cases, the amount of data to be processed depends on the sparse structure of the problem, which will typically vary in a data-dependent way.

In supporting dynamic shapes, we chose not to support dynamic rank programs, e.g., programs whose inputs tensors change in dimensionality, as this pattern rarely occurs in real-world deep learning programs, and it avoids the need to reason inductively over symbolic lists of shapes.

5.1 Symbolic Shape Guards

The use of straight line traces in TorchDynamo was motivated by the need to reuse preexisting code written in Python/C++ targeting the PyTorch API. We continue this philosophy with dynamic shapes: unlike a fully symbolic system which might capture both branches of a conditional, we always pick one branch and *specialize* our trace under the assumption that this trace will only be reused when the assumptions hold. To do this, we maintain a *size hint* for every symbolic size saying what its concrete value was on the first input that triggered the just-in-time compilation. When we perform a condition on the shape of a tensor, we consult the hint to find out which branch to take and add a guard.

This greatly simplifies the symbolic shape formulas we produce, as we do not need to represent conditionals, but it means we have a much more involved system for managing guards. Consider, for example, the following program:

```
def f(x, y):
    z = torch.cat([x, y])
    if z.size(0) > 2:
        return z.mul(2)
    return z.add(2)
```

The final IR we will compile with TorchInductor will either be `torch.cat([x, y]).add(2)` or `torch.cat([x, y]).mul(2)` (with the condition flattened away), but to determine which branch we are in, we would need to know the size of `z`, an intermediate. Because TorchDynamo must know up-front if a compiled trace is valid (we do not support bailouts, like some JIT compilers), we must be able to reduce `z.size(0)` to an expression in terms of the inputs, `x.size(0) + y.size(0)`. This is done by writing *meta functions* for all operators in PyTorch. Meta functions propagate size information to the output of a tensor without actually performing computation on the node. At the time of writing, coverage for meta functions was 2657 out of 3028 PyTorch ops (including overloads), which covers the vast majority of real-world models since there is a long tail of rarely/never used operators. There is also mechanism for defining your own meta functions for custom ops.

5.2 Optimizing Dynamic Shapes Reasoning

A major motivation of dynamic shapes is to reduce compile time, as a compiler which supports only static shapes must recompile a kernel for every possible combination of possible input shapes. However, reasoning over symbolic shapes comes with its own costs: in the limit, the shape expressions for output tensors may be quite complicated. We employ a variety of strategies to reduce the performance impact of symbolic shapes reasoning:

- Our default API for dynamic shapes does not require any user annotation: we assume that all inputs are potentially dynamic, model weights are static, and we infer the true dynamism by stepping through the model and analyzing the interactions between the two. We also support a mode `assume_static_by_default` which forces all input dimensions to be assumed static unless a user explicitly marks them as dynamic with `mark_dynamic(tensor, dim)`.
- Code in PyTorch often performs tests on whether or not a size of a variable is zero or one; for example, when constructing a tensor, PyTorch computes if it is contiguous. A zero element tensor is always contiguous, so we always test if each dimension of a tensor is zero. Instead of forcing our symbolic reasoning system to rediscover this fact every trace, we instead proactively 0/1 specialize: if an input size is 0 or 1, instead of assigning it a symbolic variable, we treat it as a constant and add an appropriate guard. Specializing on 1 is important to capture broadcasting semantics in PyTorch and performance optimizations. Importantly, we can make a negative inference when we do allocate a symbolic variable: any symbolic variable *must not* equal 0/1, and so if we test if it is equal to 0/1, we can evaluate the expression to false without having to introduce a additional guard.
- As we process the user program, we incrementally simplify our symbolic expressions as we learn more facts from guards. Our current implementation simplifies unification and divisibility on the fly, and we also use SymPy [28] to help us determine if a requested guard is already statically known, in which case we can eliminate it.

5.3 Hint-Free (Unbacked) Symbolic Integers

To resolve control flow, we check the actual value of a symbolic integer to determine which branch to take and guard on. Unbacked symbolic integers arise when a size variable emerges from a data-dependent operation like `.nonzero()` or `.item()` and the actual value is unknown. It is illegal to perform control flow on these symbolic integers, so we must graph break on these operations.

Naively implemented, this is too restrictive and results in too many graph breaks. The most important enhancements to work around these are: 1) On tensor creation, PyTorch precomputes data about a tensor; for example, when using `empty_strided` to create a tensor, it will sort the strides and determine whether the tensor is non-overlapping and dense. Sorts produce a lot of guards. However, it is more common to produce a tensor directly with a higher-level API like `empty`, which is guaranteed to produce a non-overlapping and dense tensor. We modified PyTorch to avoid needlessly recomputing these properties. 2) Even if nontrivial compute is needed, sometimes a property is never used. Making these precomputed properties lazy allows us to avoid guarding on unused properties. 3) It is generally unknown whether or not data within an integer tensor may be negative. However, we provide an API `constrain_range` whereby a user can specify that a size is bounded above and below by known limits.

6 Experimental Results

We run our evaluation on three different benchmark suites. TorchBench [14] is a benchmark suite containing a diverse set of models taken from open source repositories selected from highly cited projects as ranked by Papers with Code [35]. HuggingFace [53] is a popular library for Transformer [49] models. TIMM [52] is a popular library containing vision models in PyTorch. To turn the last two libraries into a benchmark suite, we selected representative models covering every category of model available.

Our benchmarking infrastructure is open source [40] in the hope that other publications will use it. Additional results can be found in the TorchInductor Performance Dashboard [41], including: per-model performance, different TorchInductor settings, and daily updates for PyTorch nightly builds. Experiments were run on an NVIDIA A100 GPU, CUDA 11.6, and an Intel Xeon 8275CL CPU. Experiments were repeated 100 times to reduce noise, with 3 warm up iterations. We applied a timeout of 30 minutes per model and count timeouts as failures. TorchInductor was run with a PyTorch nightly build from 8/30/2023, with max-autotune, freezing, and cudagraphs enabled. Other versions used are: nvFuser 2.0; NNC 2.0; Hidet 0.2.2; TVM 0.11.1; ONNX Runtime (ONNXRT) 1.14.1; and PyTorch/XLA 2.1. For training experiments, we measure a single step of both the forwards and backwards pass excluding the optimizer.

6.1 TorchDynamo's Ability to Capture Graphs

The first section of Table 1 shows experimental results comparing TorchDynamo to TorchScript [17] in terms of their ability to capture different benchmark suites. For HuggingFace, TorchScript fails on every model because HuggingFace models returns a `ModelOutput` container class that TorchScript does not support. Most TIMM models work with TorchScript because the maintainers of TIMM use TorchScript in their workflows and have put in effort to adapt their models. On TorchBench, TorchDynamo works on more than twice as

	TorchBench	HuggingFace	TIMM
Model Count	80	46	62
Works with TorchDynamo	74 (93%)	46 (100%)	62 (100%)
Compare with TorchScript [17]	36 (45%)	0 (0%)	61 (98%)
Operators Captured	91.8%	99.8%	100%
Mean Operators per Graph	252.8	612.6	450.7
Mean Graphs per Model	21.1	7.7	1
Models with 0 graph breaks	52 (70%)	41 (89%)	62 (100%)
Models with 1 to 9 graph breaks	6 (8%)	1 (2%)	0 (0%)
Models with 10+ graph breaks	16 (22%)	4 (9%)	0 (0%)

Table 1. TorchDynamo statistics from each benchmark suite, measured using float32 inference on an NVIDIA A100 GPU.

	Inference	Training
TorchDynamo	5%	1%
Lazy Tensors	38%	90%
Lazy Tensors + cross-iteration pipelining	31%	86%

Table 2. Overheads (lower is better) as a percentage of eager PyTorch execution time for graph capture. This experiment uses the same kernels as eager PyTorch, so overheads are graph capture cost only. Measured using float32 TorchBench on an NVIDIA V100 GPU.

many models as TorchScript. TorchBench is the most representative of the three benchmark suites for graph capture comparison because it is made up of models taken from diverse sources.

The second section of Table 1 provides statistics about the quality of graphs captured by TorchDynamo, normalized as a percentage of working models. Unlike prior systems, which were all-or-nothing, TorchDynamo can capture partial programs and multiple graphs. TorchDynamo is able to capture a single whole-program graph most of the time, and even when there are graph breaks, typical graphs are hundreds of operators in size. The most common reason for graph breaks are: usage of non-PyTorch libraries such as numpy [21]; conversion to Python types such as `tolist()`; and data-dependent control flow operations. There is support for compiling numpy operations with `torch.compile`, which is not enabled for this experiment.

6.2 Overheads of Graph Capture

Table 2 measures the runtime overheads introduced by graph capture for both TorchDynamo and Lazy Tensors. The other systems are ahead-of-time and do not introduce runtime overhead. We run each system using the same kernels as PyTorch eager, so slowdowns are from graph capture overhead only. We take the geometric mean slowdown on TorchBench and subtract 1 to get a percentage overhead added. As with all our results, we exclude warm up iterations from timing, so this is measuring steady-state performance.

While the overheads of TorchDynamo are less than 5%, Lazy Tensors adds a large amount of overhead. These Lazy Tensor overheads are not uniform across models. For training with cross-iteration pipelining: one third of models are better than 10% overhead, one third of models are between 10% and 66% overhead, and one third of models are between 66% and 1759% overhead.

One way to mitigate Lazy Tensor overheads in training and offline inference is cross-iteration pipelining. This helps with the fact that for a single iteration of Lazy Tensors, the GPU is idle while the CPU captures, then the CPU is idle while the GPU executes what was captured. By running multiple iterations one can overlap the capture of iteration N with the execution of iteration $N-1$. *Lazy Tensors + cross-iteration pipelining* in Table 2 measures this amortization effect by measuring 10 iterations rather than 1 iteration. There is a small improvement in Lazy Tensor overheads from this strategy.

For many models, Lazy Tensor capture is too slow to saturate the GPU. This is especially true for smaller models or ones with large numbers of operations. In these cases, pipelining does not help because the limiting factor is Lazy Tensor overheads. For some PyTorch models, there is code like *if torch.any(torch.isnan(x))* or *print(loss.item())*. Both of these operations take values from within PyTorch tensors and convert them into Python *bool* or *float* types. This type of code is fast in eager mode PyTorch, but defeats any cross iteration pipelining, because with a (not yet computed) Lazy Tensor, you have no way of knowing what *torch.any()* should return (which controls the branch the code will take) or the values to print. Since Lazy Tensors has zero visibility into the Python code calling it, this pattern forces a flush of any accumulated pipeline of ops and requires the CPU capture to stall and wait for the GPU to catch up.

6.3 TorchInductor Speedups

Table 3 shows the geometric mean speedups of TorchInductor and six other TorchDynamo backends over PyTorch eager across our three benchmark suites and many configurations. In this experiment, we hold the graph capture mechanism (TorchDynamo) constant and only vary the backend compiler, so every backend gets the same input graphs and incurs the same capture overheads. Figure 4 is based on the same data as Table 3, but shows the Cumulative Distribution Function (CDF) of speedups with the three benchmark suites combined. This helps better understand how speedups are distributed.

TorchInductor is faster than other backends in most cases. nvFuser [36] and NNC [60] both have speedups clustered around 1× because they make use of eager PyTorch kernels and only generate code for a subset of PyTorch. PyTorch/XLA [42] has more varied performance, in many cases generating large speedups and, in other cases, large slowdowns which pull down the average. It performs better for GPU float16 inference compared to other configurations, especially on the vision models in TIMM. ONNX Runtime [16],

		TorchBench				HuggingFace				TIMM			
		Inference		Training		Inference		Training		Inference		Training	
		Models Working	Geomean Speedup	Models Working	Geomean Speedup	Models Working	Geomean Speedup	Models Working	Geomean Speedup	Models Working	Geomean Speedup	Models Working	Geomean Speedup
NVIDIA A100 GPU float32	None (TorchDynamo-only)	74/74	0.95×	59/59	0.99×	46/46	1.01×	46/46	0.98×	62/62	1.00×	62/62	1.00×
	TorchInductor	74/74	**2.73**×	58/59	**1.38**×	46/46	**1.47**×	46/46	**1.24**×	62/62	**2.48**×	62/62	**1.38**×
	nvFuser [36]	53/74	1.23×	45/59	1.04×	38/46	1.09×	36/46	1.09×	57/62	1.16×	56/62	1.03×
	NNC [60]	53/74	1.12×	42/59	1.03×	21/46	0.98×	21/46	0.94×	57/62	1.02×	58/62	0.96×
	PyTorch/XLA [42]	57/74	0.80×	42/59	0.73×	33/46	1.03×	18/46	0.98×	53/62	1.24×	52/62	1.11×
	ONNXRT [16]	34/74	0.86×	N/A	N/A	21/46	0.84×	N/A	N/A	29/62	0.92×	N/A	N/A
	TVM [12]	41/74	0.16×	N/A	N/A	22/46	0.09×	N/A	N/A	37/62	0.13×	N/A	N/A
	Hidet [18]	15/74	0.54×	N/A	N/A	0/46	N/A	N/A	N/A	5/62	0.30×	N/A	N/A
NVIDIA A100 GPU float16	None (TorchDynamo-only)	74/74	0.95×	57/57	0.99×	45/45	1.00×	45/45	0.97×	60/60	1.00×	60/60	1.00×
	TorchInductor	74/74	**2.59**×	57/57	**1.50**×	45/45	**1.91**×	45/45	**1.45**×	60/60	**2.77**×	60/60	**1.50**×
	nvFuser [36]	53/74	1.27×	45/57	1.04×	37/45	1.07×	35/45	1.04×	56/60	1.13×	54/60	1.01×
	NNC [60]	53/74	1.14×	46/57	1.03×	35/45	0.98×	37/45	0.94×	56/60	1.00×	56/60	0.95×
	PyTorch/XLA [42]	56/74	0.82×	42/57	0.80×	33/45	1.16×	18/45	0.24×	53/60	1.59×	50/60	1.27×
	ONNXRT [16]	34/74	0.86×	N/A	N/A	21/45	0.84×	N/A	N/A	29/60	0.92×	N/A	N/A
	TVM [12]	40/74	0.17×	N/A	N/A	31/45	0.18×	N/A	N/A	34/60	0.10×	N/A	N/A
	Hidet [18]	15/74	0.57×	N/A	N/A	0/45	N/A	N/A	N/A	5/60	0.46×	N/A	N/A
Intel Xeon 8275CL CPU float32	None (TorchDynamo-only)	74/74	1.00×	57/57	0.99×	46/46	1.00×	46/46	1.00×	62/62	1.06×	61/61	1.00×
	TorchInductor	74/74	**1.39**×	57/57	**1.35**×	45/46	**2.54**×	40/46	**1.36**×	58/62	**2.55**×	52/61	**1.42**×
	NNC [60]	51/74	1.14×	47/57	0.99×	38/46	1.15×	38/46	0.92×	58/62	1.04×	58/61	0.85×
	PyTorch/XLA [42]	58/74	0.27×	43/57	0.21×	33/46	0.38×	16/46	0.34×	50/62	0.38×	37/61	0.25×
	ONNXRT [16]	46/74	1.06×	N/A	N/A	35/46	0.64×	N/A	N/A	54/62	1.02×	N/A	N/A
	TVM [12]	44/74	0.64×	N/A	N/A	14/46	0.10×	N/A	N/A	47/62	1.46×	N/A	N/A

Table 3. Geometric mean speedups (higher is better) over PyTorch eager for different TorchDynamo backends and the number of models they work on in each benchmark suite. Only working models are included in speedup calculation. Comparisons use same precision in eager mode. N/A means the backend does not support that configuration. All backends use TorchDynamo as frontend to capture graphs in this experiment and receive the same initial graphs. *None* is included as a way to estimate overheads (or speedups) of TorchDynamo without any graph optimizations applied.

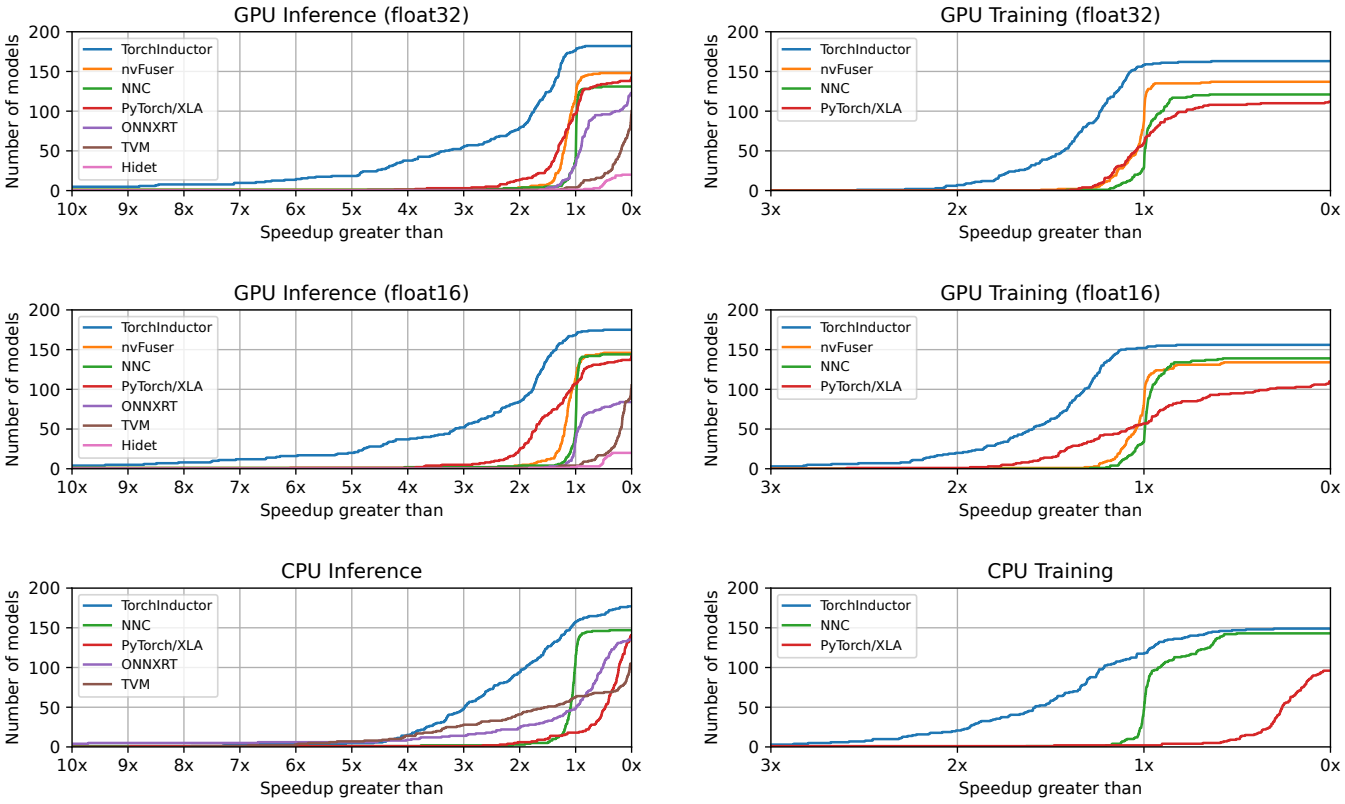

Figure 4. Cumulative Distribution Function (CDF) of speedups over PyTorch eager mode. For speedups (x-axis) higher is better and PyTorch eager is 1×. Same underlying data as Table 3.

	Inference	Training
All TorchInductor optimizations	1.91×	1.45×
Without loop/layout reordering	1.91× (-0.00)	1.28× (-0.17)
Without matmul templates	1.85× (-0.06)	1.41× (-0.04)
Without parameter freezing	1.85× (-0.06)	1.45× (-0.00)
Without pattern matching	1.83× (-0.08)	1.45× (-0.00)
Without cudagraphs	1.81× (-0.10)	1.37× (-0.08)
Without fusion	1.68× (-0.23)	1.27× (-0.18)
Without inlining	1.58× (-0.33)	1.31× (-0.14)
Without fusion and inlining	0.80× (-1.11)	0.59× (-0.86)

Table 4. Ablation study measuring the impact of removing optimizations from TorchInductor. Geometric mean speedups over eager PyTorch on float16 HuggingFace on an NVIDIA A100 GPU. Parenthesis is difference from *All TorchInductor optimizations*.

TVM [12], and Hidet [18] are inference-only and fail to run many models due to missing operator implementations and other issues. On CPU, the ONNX runtime generates speedups above 8× for 5 models (compared to 1 for TorchInductor), but these results did not generalize – more than half of models show slowdowns. On GPU, TVM and Hidet produce slowdowns for all except 4, and 2, models respectively. On CPU, TVM performs significantly better for some models while generating large slowdowns on others. TVM would have been the second fastest CPU inference backend on TorchBench (behind TorchInductor) if we excluded the models where it generated large slowdowns.

6.4 Sources of TorchInductor Speedups

Table 4 explores where TorchInductor's speedups are coming from by disabling optimizations one at a time an measuring the impact on geometric mean speedup on HuggingFace models. If removing a specific optimization results in a bigger slowdown, this implies that it is responsible for more of the speedup.

The biggest speedups in TorchInductor come from combining pointwise, reduction, and scatter kernels together into a smaller number of fused kernels, which reduces memory traffic since values can be reused without requiring a round trip to memory. In TorchInductor, these kernel combinations happen in two places: 1) *Inlining* happens during lowering, and duplicates the body of pointwise kernels into all their consumers when thresholds are met. 2) *Fusion* happens during scheduling, and combines remaining kernels together, and also does horizontal consumer/consumer fusions. There is a lot of overlap between those passes, so we also include a line *without fusion and inlining* that disables both. Without both of those passes, TorchInductor generates slowdowns rather than speedups. This is because the decompositions performed by TorchInductor break up larger optimized operators into many smaller primitive operators, and we rely on fusions to recombine them to recover 1× performance.

The remaining optimizations measured in Table 4 are: 1) *Loop/layout reordering* uses a voting algorithm to reorder loops in kernels and change data layouts to match usage. 2) *Matmul templates* use Triton templates with pointwise epilogue fusion for matrix multiply instead of cuBLAS/cuDNN. There is an autotuner (enabled by `mode="max-autotune"`) to select when to use these templates. Without this optimization, TorchInductor does not use templates at all. 3) *Parameter freezing* is an inference-only optimization that constant-folds away parts of the model that only depend on parameters. 4) *Pattern matching* uses graph-level peephole optimizations to rewrite the input graph before it is lowered to TorchInductor. 5) *Cudagraphs* is a way to reduce kernel launch overheads at the CUDA driver level. TorchInductor will automatically use this when static analysis shows it to be safe and it is enabled in the configuration.

7 Conclusions

In this paper, we presented two extensions to PyTorch: TorchDynamo and TorchInductor, which deliver speedups through graph compilation in PyTorch programs while retaining the flexibility and usability of the eager programming model PyTorch is known for. By enabling graph compilation in PyTorch programs, we hope to empower researchers and practitioners to tackle larger and more complex machine learning problems with greater efficiency and flexibility.

Acknowledgements

We gratefully thank the anonymous reviewers and our shepherd, Martin Maas, for their suggestions and feedback that helped improve this paper. Thanks to Brett Simmers for proofreading. Thanks to everyone working on Triton, TorchInductor's GPU backend would not have been possible without it. Thanks to the Intel PyTorch team: Guobing Chen, Leslie Fang, Jiong Gong, Xuan Liao, Yudong Si, Chuanqi Wang, Eikan Wang, Chunyuan Wu, Weiwen Xia, Xiaobing Zhang, Fan Zhao, and Beilei Zheng. Their work greatly improved TorchInductor's CPU backend. Finally, thanks to the thousands of people who have contributed code to PyTorch. This work would not have been possible without the countless contributions that collectively made PyTorch what is is today.

A Artifact Appendix

A.1 Abstract

The source code for this work is included in PyTorch which is available at https://github.com/pytorch/pytorch/. TorchDynamo can be found in the `torch/_dynamo` directory and TorchInductor can be found in the `torch/_inductor` directory. Benchmarking code to reproduce the results in the

paper can be found at https://github.com/pytorch/pytorch/tree/main/benchmarks/dynamo.

Since this paper includes a large number of experiments that in aggregate will take weeks to run, the instructions here will focus on reproducing the TorchInductor GPU HuggingFace results. The workflow to reproduce other results is very similar to this and is described at the end. Additional instructions are included in the `README.md` included in the `benchmarks/dynamo` directory in PyTorch.

A.2 Artifact check-list (meta-information)

- **Binary:** distributions available at https://pytorch.org/
- **Hardware:** NVIDIA A100 GPU, Intel Xeon 8275CL CPU
- **Metrics:** Geomean speedup over PyTorch eager mode
- **How much disk space required (approximately)?:** 50 GB
- **How much time is needed to prepare workflow (approximately)?:** 1 hour
- **How much time is needed to complete experiments (approximately)?:** < 1 day per-backend, per-configuration for most experiments
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** BSD-3

A.3 Description

A.3.1 How to access.

- Source code and benchmark code: https://github.com/pytorch/pytorch/
- PyTorch binaries: https://pytorch.org/
- TorchBench: https://github.com/pytorch/benchmark/

A.3.2 Hardware dependencies.

- To match configurations in this paper: NVIDIA A100 GPU and Intel Xeon 8275CL CPU
- Benchmarks can run with an NVIDIA GPU with SM80+ and 40GB+ of memory, most benchmarks can run with less
- CPU results can be run without a GPU

A.3.3 Software dependencies.

- A recent Linux distribution
- NVIDIA kernel drivers
- CUDA version compatible with the chosen version of PyTorch
- gcc/g++ compatible with the chosen CUDA
- Miniconda installed (https://docs.conda.io/projects/miniconda/en/latest/)
- PyTorch (and dependencies)
- Additional python packages: pandas, scipy, psutil, and tqdm

A.4 Installation

There are a number of options to install PyTorch which are described on https://pytorch.org/. A minimal installation including dependencies can be achieved using the following commands:

```
# create a new conda environment
conda create --name=pt2 python=3.10
conda activate pt2

# install dependencies for benchmark code
conda install pandas scipy psutil tqdm

# install PyTorch using release build
conda install pytorch torchvision torchaudio pytorch-cuda=12.1 \
    -c pytorch -c nvidia
```

Next, download the PyTorch source code in order to access benchmarking scripts:

```
# clone the PyTorch repository to get benchmark code
git clone --recursive --branch=release/2.1 \
    https://github.com/pytorch/pytorch

# benchmark code should be run from the root PyTorch directory
cd pytorch
```

A.5 Experiment workflow

To reproduce TorchInductor speedups over eager PyTorch on HuggingFace, float16, GPU, inference run:

```
TORCHINDUCTOR_MAX_AUTOTUNE=1 ./benchmarks/dynamo/huggingface.py \
    --performance --no-skip \
    -dcuda --float16 --inference \
    --inductor --freezing \
    --output=`pwd`/results.csv
```

This downloads HuggingFace models and runs them both with and without TorchDynamo to compute speedups compared to PyTorch eager mode. Results are written to `results.csv` in the current working directory. If one runs additional experiments, `--output` should be set to a unique absolute filename for each one.

A.6 Evaluation and expected results

The chosen output file (`results.csv`) should contain 46 entries showing speedup numbers (and other metrics) for each model. All models should be working (failures are represented as a zero speedup) and the geomean of all the speedups should be similar to the speedups reported in the paper.

A.7 Experiment customization

The above command can be customized in many ways:

- `./benchmarks/dynamo/huggingface.py` can be substituted with the scripts `./benchmarks/dynamo/timm_models.py` or `./benchmarks/dynamo/torchbench.py` for the three benchmark suites. Note that TorchBench requires additional installation steps, while the other two auto-download dependencies.
- `-dcuda` can be replaced with `-dcpu` for CPU
- `--float16` can be replaced with `--float32` or `--amp`
- `--inference` can be replaced with `--training`
- `--inductor` can be replaced with `--backend=eager` (for "None"), `--backend=nvfuser`, `--backend=nnc` , `--xla`, `--backend=onnxrt`, `--backend=tvm`, or `--backend=hidet`. Note that each backend has different dependencies and setup instructions.

- `--freezing` and/or `TORCHINDUCTOR_MAX_AUTOTUNE=1` can be removed to disable those optimizations in TorchInductor. Many more optimization flags can be found in `torch/_inductor/config.py`.
- Many other options and backends are available via `--help`

The results in this paper include the combinatorial product of most of these flags.

A.8 Notes

- Speedups and model coverage results have improved in recent versions of PyTorch compared to results shown in this paper. We recommend running the latest PyTorch version for future comparisons.
- Performance results can be sensitive to environment setup, such as hardware and CUDA versions, so some small differences are expected.
- Additional installation steps are required for TorchBench and non-TorchInductor backends.
- A performance dashboard based on these scripts is available at https://hud.pytorch.org/benchmark/compilers.

References

[1] Martin Abadi et al. 2016. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 265–283. https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf.

[2] [SW] Martín Abadi et al., TensorFlow, Large-scale machine learning on heterogeneous systems Nov. 2015. DOI: 10.5281/zenodo.4724125.

[3] Hameer Abbasi, Edward Z Yang, and Ralf Gommers. 2020. Improving subclassing Tensor by propagating subclass instances. https://github.com/pytorch/rfcs/blob/master/RFC-0001-torch-function-for-methods.md. (Aug. 2020).

[4] Akshay Agrawal et al. 2019. TensorFlow Eager: A Multi-Stage, Python-Embedded DSL for Machine Learning. *CoRR*, abs/1903.01855. http://arxiv.org/abs/1903.01855 arXiv: 1903.01855.

[5] Rami Al-Rfou et al. 2016. Theano: A Python framework for fast computation of mathematical expressions. *CoRR*, abs/1605.02688. http://arxiv.org/abs/1605.02688 arXiv: 1605.02688.

[6] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. Opentuner: an extensible framework for program autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation* (PACT '14). Association for Computing Machinery, Edmonton, AB, Canada, 303–316. ISBN: 9781450328098. DOI: 10.1145/2628071.2628092.

[7] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: a polyhedral compiler for expressing fast and portable code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization* (CGO 2019). IEEE Press, Washington, DC, USA, 193–205. ISBN: 9781728114361.

[8] [SW] James Bradbury et al., JAX: composable transformations of Python+NumPy programs version 0.3.13, 2018. URL: http://github.com/google/jax.

[9] Dino Viehland Brett Cannon. 2016. PEP 523: adding a frame evaluation API to CPython. https://peps.python.org/pep-0523/. (2016).

[10] Jack Cao. 2022. PyTorch/XLA 2022 Q4 dev update. https://dev-discuss.pytorch.org/t/pytorch-xla-2022-q4-dev-update/961. (2022).

[11] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to optimize tensor programs. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (NIPS'18). Curran Associates Inc., Montréal, Canada, 3393–3404.

[12] Tianqi Chen et al. 2018. TVM: an automated End-to-End optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, (Oct. 2018), 578–594. ISBN: 978-1-939133-08-3. https://www.usenix.org/conference/osdi18/presentation/chen.

[13] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: efficient primitives for deep learning. (2014). arXiv: 1410.0759 [cs.NE].

[14] Will Constable et al. 2020. TorchBench: a collection of open source benchmarks for PyTorch performance and usability evaluation. https://github.com/pytorch/benchmark. (Sept. 2020).

[15] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5, 1, 46–55.

[16] ONNX Runtime developers. 2021. ONNX runtime. https://www.onnxruntime.ai. (2021).

[17] Zachary DeVito et al. 2018. TorchScript. https://pytorch.org/docs/1.9.0/jit.html. (Sept. 2018).

[18] Yaoyao Ding, Cody Hao Yu, Bojian Zheng, Yizhi Liu, Yida Wang, and Gennady Pekhimenko. 2023. 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* (ASPLOS 2023). Association for Computing Machinery, Vancouver, BC, Canada, 370–384. ISBN: 9781450399166. DOI: 10.1145/3575693.3575702.

[19] Siyuan Feng et al. 2022. TensorIR: an abstraction for automatic tensorized program optimization. (2022). arXiv: 2207.04296 [cs.LG].

[20] Alan Gray. 2019. Getting started with CUDA graphs. https://developer.nvidia.com/blog/cuda-graphs/. (2019).

[21] Charles R. Harris et al. 2020. Array programming with NumPy. *Nature*, 585, 357–362. DOI: 10.1038/s41586-020-2649-2.

[22] Horace He. 2019. The state of machine learning frameworks in 2019. https://thegradient.pub/state-of-ml-frameworks-2019-pytorch-dominates-research-tensorflow-dominates-industry/. (2019).

[23] Mike Innes et al. 2017. On machine learning and programming languages. https://julialang.org/blog/2017/12/ml-pl/. (Dec. 2017).

[24] ISO. 1998. *ISO/IEC 14882:1998: Programming languages — C++*. (Sept. 1998), 732. http://webstore.ansi.org/ansidocstore/product.asp?sku=ISO%2FIEC+14882%2D1998.

[25] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross B. Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. *CoRR*, abs/1408.5093. http://arxiv.org/abs/1408.5093 arXiv: 1408.5093.

[26] Norman P. Jouppi et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (ISCA '17). Association for Computing Machinery, Toronto, ON, Canada, 1–12. ISBN: 9781450348928. DOI: 10.1145/3079856.3080246.

[27] Chris Lattner et al. 2021. MLIR: scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2–14. DOI: 10.1109/CGO51591.2021.9370308.

[28] Aaron Meurer et al. 2017. SymPy: symbolic computing in Python. *PeerJ Computer Science*, 3, (Jan. 2017). DOI: 10.7717/peerj-cs.103.

[29] Adrian Mönnich, Armin Ronacher, David Lord, Grey Li, Joshua Bronson, Markus Unterwaditzer, and Philip Jones. 2023. Jinja project. https://github.com/pallets/jinja. (2023).

[30] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. 2023. CUDA. https://developer.nvidia.com/cuda-toolkit. (2023).

[31] 2023. ONNX. https://onnx.ai/. (2023).

[32] 2019. *Pytorch: an imperative style, high-performance deep learning library. Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Curran Associates Inc., Red Hook, NY, USA, 12 pages.

[33] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '13). Association for Computing Machinery, Seattle, Washington, USA, 519–530. ISBN: 9781450320146. DOI: 10.1145/2491956.2462176.

[34] James Reed, Zachary DeVito, Horace He, Ansley Ussery, and Jason Ansel. 2022. Torch.fx: practical program capture and transformation for deep learning in python. In *Proceedings of Machine Learning and Systems*. D. Marculescu, Y. Chi, and C. Wu, (Eds.) Vol. 4, 638–651. https://proceedings.mlsys.org/paper/2022/file/ca46c1b9512a7a8315fa3c5a946e8265-Paper.pdf.

[35] Elvis Saravia. 2021. Papers with Code 2021: a year in review. https://medium.com/paperswithcode/papers-with-code-2021-a-year-in-review-de75d5a77b8b. (2021).

[36] Christian Sarofeen, Piotr Bialecki, Jie Jiang, Kevin Stephano, Masaki Kozuki, Neal Vaidya, and Stas Bekman. 2022. Introducing nvFuser, a deep learning compiler for PyTorch. https://pytorch.org/blog/introducing-nvfuser-a-deep-learning-compiler-for-pytorch/. (2022).

[37] Frank Seide and Amit Agarwal. 2016. CNTK: microsoft's open-source deep-learning toolkit. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (KDD '16). Association for Computing Machinery, San Francisco, California, USA, 2135. ISBN: 9781450342322. DOI: 10.1145/2939672.2945397.

[38] Junru Shao et al. 2022. Tensor program optimization with probabilistic programs. (2022). arXiv: 2205.13603 [cs.LG].

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

[40] PyTorch Team. 2023. TorchDynamo Benchmarking Code. https://github.com/pytorch/pytorch/tree/main/benchmarks/dynamo. (2023).

[41] PyTorch Team. 2023. TorchInductor Performance Dashboard. https://hud.pytorch.org/benchmark/compilers. (2023).

[42] PyTorch XLA Team. 2023. PyTorch/XLA. https://github.com/pytorch/xla. (2023).

[43] Vijay Thakkar et al. 2023. CUTLASS. https://github.com/NVIDIA/cutlass. Version 3.0.0. (Jan. 2023).

[44] [SW] The IREE Authors, IREE Sept. 2019. URL: https://github.com/openxla/iree.

[45] The XLA Team. 2017. XLA - Tensorflow, compiled. https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html. (Mar. 2017).

[46] Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In (MAPL 2019). Association for Computing Machinery, Phoenix, AZ, USA, 10–19. ISBN: 9781450367196. DOI: 10.1145/3315508.3329973.

[47] Seiya Tokui et al. 2019. Chainer: A Deep Learning Framework for Accelerating the Research Cycle. *CoRR*, abs/1908.00213. http://arxiv.org/abs/1908.00213 arXiv: 1908.00213.

[48] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor comprehensions: framework-agnostic high-performance machine learning abstractions. (2018). arXiv: 1802.04730 [cs.PL].

[49] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (NIPS'17). Curran Associates Inc., Long Beach, California, USA, 6000–6010. ISBN: 9781510860964.

[50] B. P. Welford. 1962. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4, 3, 419–420. DOI: 10.1080/00401706.1962.10490022.

[51] Jian Weng, Animesh Jain, Jie Wang, Leyuan Wang, Yida Wang, and Tony Nowatzki. 2021. Unit: unifying tensorized instruction compilation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 77–89. DOI: 10.1109/CGO51591.2021.9370330.

[52] Ross Wightman. 2019. PyTorch image models. https://github.com/rwightman/pytorch-image-models. (2019). DOI: 10.5281/zenodo.4414861.

[53] Thomas Wolf et al. 2020. Transformers: State-of-the-Art Natural Language Processing. In Association for Computational Linguistics, (Oct. 2020), 38–45. https://www.aclweb.org/anthology/2020.emnlp-demos.6.

[54] Jiarong Xing, Leyuan Wang, Shang Zhang, Jack Chen, Ang Chen, and Yibo Zhu. 2022. Bolt: bridging the gap between auto-tuners and hardware-native performance. In *Proceedings of Machine Learning and Systems*. D. Marculescu, Y. Chi, and C. Wu, (Eds.) Vol. 4, 204–216. https://proceedings.mlsys.org/paper_files/paper/2022/file/38b3eff8baf56627478ec76a704e9b52-Paper.pdf.

[55] Shangdi Yu and Horace He. 2023. Transcending runtime-memory tradeoffs in checkpointing by being fusion aware. In *Proceedings of Machine Learning and Systems*.

[56] Bojian Zheng et al. 2022. Dietcode: automatic optimization for dynamic tensor programs. In *Proceedings of Machine Learning and Systems*. D. Marculescu, Y. Chi, and C. Wu, (Eds.) Vol. 4, 848–863. https://proceedings.mlsys.org/paper_files/paper/2022/file/fa7cdfad1a5aaf8370ebeda47a1ff1c3-Paper.pdf.

[57] Lianmin Zheng et al. 2020. Ansor: generating high-performance tensor programs for deep learning. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation* (OSDI'20) Article 49. USENIX Association, USA, 17 pages. ISBN: 978-1-939133-19-9.

[58] Size Zheng et al. 2022. AMOS: enabling automatic mapping for tensor computations on spatial accelerators with hardware abstraction. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (ISCA '22). Association for Computing Machinery, New York, New York, 874–887. ISBN: 9781450386104. DOI: 10.1145/3470496.3527440.

[59] Hongyu Zhu et al. 2022. ROLLER: fast and efficient tensor compilation for deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, (July 2022), 233–248. ISBN: 978-1-939133-28-1. https://www.usenix.org/conference/osdi22/presentation/zhu.

[60] Mikhail Zolotukhin. 2021. NNC walkthrough: how PyTorch ops get fused. https://dev-discuss.pytorch.org/t/nnc-walkthrough-how-pytorch-ops-get-fused/125. (2021).