Torchy: A Tracing JIT Compiler for PyTorch (Extended Version)

NUNO P. LOPES, INESC-ID / Instituto Superior Técnico - University of Lisbon, Portugal

Machine learning (ML) models keep getting larger and more complex. Whereas before models used to be represented by static data-flow graphs, they are now implemented via arbitrary Python code. Eager-mode frameworks, such as PyTorch, are now the standard for developing new ML models. The semantics of eager-mode frameworks is that operations are computed straight away, and thus one can inspect the result of any operation at any point. This greatly simplifies the development process, and it enables more dynamic ML models.

Although eager-mode frameworks are more convenient, they are less efficient today as operations are dispatched to the hardware one at a time. This execution model precludes, for example, operation fusion, which is essential for executing ML workloads efficiently.

In this paper we present Torchy, a tracing JIT compiler for PyTorch, one of the mainstream eager-mode frameworks. Torchy achieves similar performance as data-flow frameworks, while providing the same semantics of straight-away execution. Moreover, Torchy works with any PyTorch program unmodified. Torchy outperforms PyTorch by up to 12x in microbenchmarks, and PyTorch's static compiler (TorchScript) by up to 5x.

1 INTRODUCTION

Machine learning (ML) models have grown significantly and have become more complex in the past few years. For example, text models have gone from 0.1 billion parameters in 2018 (ELMo [42]) to 175 billion in 2020 (GPT-3 [7]). Similarly, image classification models went from 0.3 million parameters in 2015 (ResNet-18 [20]) to 46 million in 2020 (RegNetX-12 [46]), with a roughly 7x increase in FLOPS.

It is not only that ML models are getting larger in terms of memory consumption, but they are also getting more dynamic and complex, e.g., using data-dependent control-flow [53]. Together with the increase in the size of training data, ML model training and inference is increasingly requiring more computing power.

Most ML models are developed in an ML framework, often a Python library. The first generation frameworks, such as TensorFlow [1], exposed a development model based on data-flow graphs. Executing a tensor operation only created a new node in the graph. Thus, an ML program consisted in code to first create a data-flow graph and then execute it many times over the training/inference data.

The data-flow graph abstraction is efficient for static ML models: the compiler gets to see the whole program ahead of time and thus can do whole-program optimizations. However, it is not an easy abstraction for users as there is a gap between what it seems to be executing an operation but which only creates a graph node. It is also harder for debugging as one needs to build the graph first and then pick the relevant nodes to print. Finally, it is hard to implement more dynamic ML models in these frameworks, as e.g., control-flow has to be emulated with multiplexers and/or algebraic operations such as matrix multiplications.

The second generation of frameworks expose the so-called eager-mode or imperative execution model, where the semantics is that operations are executed straight away. Thus, operations return their result rather than a graph node. This model is much simpler and obvious to users, and therefore these frameworks such as PyTorch [41] are now mainstream.

The main disadvantage of eager-mode frameworks is performance. Consider the following example PyTorch program where x and y are tensors and c is some constant:

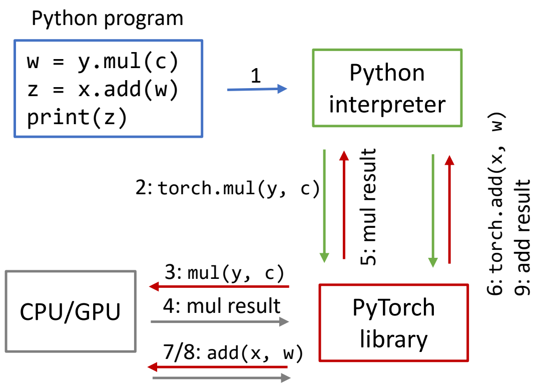


Fig. 1. Execution of an example PyTorch program.

```
w = y.mul(c)
z = x.add(w)
print(z)
```

Figure 1 shows how this program is executed. Firstly, Python reads the file and starts executing it (note that PyTorch is merely a Python library). Secondly, Python encounters the call to the `mul` function, which it dispatches to PyTorch. Thirdly, PyTorch checks in which device the tensors are placed (CPU or GPU) and calls the respective library function that implements element-wise multiplication. Finally, PyTorch gets the results from the underlying algebra library and returns a new tensor to Python. The same steps are repeated for the addition and print operations.

This example shows the immense overhead that each operation goes through in PyTorch. To offset this overhead, several "hacks" emerged. The first is to offer more complex operations that take more time to execute and therefore hide better the overhead and allow more optimized implementations as several smaller operations are fused in a single one. For example, the code above can be written with a single operation:

```
z = x.add(y, alpha=c)
```

With this change we reduced the overhead by half. However, this comes at the cognitive expense of developers: PyTorch 1.10 has more than 2,000 operations, and this keeps increasing with every release [19]! How can developers remember all operations and keep up-to-date?

Another technique that PyTorch uses to increase performance is asynchronous GPU execution. Since Python runs on the CPU and tensor GPU operations run, well, on the GPU, PyTorch runs these in parallel by returning control to Python before operations complete. Hence, as Python executes, PyTorch keeps accumulating operations in CUDA's queue. This also helps to hide the large latency of going back-and-forth the GPU. Nevertheless, operations are still dispatched one-by-one and therefore not fused. This technique is transparent to users.

PyTorch also offers a compiler named TorchScript. It offers two modes: compilation of (a subset of) Python code and tracing. Our previous example can be compiled with TorchScript as follows:

```
@torch.jit.script
def fn(x : Tensor, y : Tensor, c : int):
  w = y.mul(c)
  return x.add(w)
z = fn(x, y, c)
```

The `@torch.jit.script` annotation instructs TorchScript to compile the function and replace the definition of `fn` to hold a TorchScript function instead of a Python one. The advantage of this compilation step is that TorchScript translates the code into its own IR, which is then optimized. In particular, the two operations get fused into a single one without the developer having to remember to call the fused operation directly. Furthermore, executing the function bypasses Python entirely.

The disadvantage of TorchScript compilation mode is that it only supports a limited subset of Python. This is by design as TorchScript is a compiler for tensor operations, not a generic Python compiler. Thus, developers have to manually split up the code and create functions consisting of mostly tensor operations so they can be compiled with TorchScript.

The second mode supported by TorchScript is tracing. The function must be first called with a set of "representative" inputs. TorchScript records all PyTorch operations executed by the function (tracing) and creates the same TorchScript IR as before, which then gets optimized. The main advantage is that it supports all of Python features, as it operates below Python. The disadvantage is that it only supports functions with a single path. The following example traces function `fn` using tensors `x` and `y` as representative inputs:

```
def fn(a, b):
  if torch.argmax(a) == 42:
    return a.add(b)
  else:
    return a.mul(b)
fn = torch.jit.trace(fn, (x,y))
```

TorchScript generates a trace with either the addition or the multiplication operations, depending on the contents of `x`. All future invocations of `fn` will follow the same branch as the tracing for `x` did, regardless of the result of `argmax`. Thus, the function may return the wrong result when invoked with arguments that would go to the other branch. Moreover, TorchScript does not produce any error in this case. This makes TorchScript's tracing mode very error-prone.

As a recap, we mentioned three methods that are available today to improve the performance of PyTorch programs that involve some manual work: 1) use larger operations instead of multiple small ones whenever possible, 2) hoist consecutive tensor operations into functions and annotate them so they are compiled with TorchScript, and 3) use TorchScript tracing on existing functions where we are sure the execution will always follow the same path for every call.

The main technical difficulty in improving the status quo is that of code discoverability. Since calls to PyTorch operations are intermingled with arbitrary Python, it is hard to extract the relevant tensor operations and compile them. Even if one was to add full Python support to TorchScript's compiler, there are fundamental theoretical limits on how much of the tensor operations could be lifted.

In this paper, we present Torchy, a tracing JIT compiler for PyTorch. Torchy addresses and solves the issue of code discoverability at run-time. Tensor operations are delayed and written down on a trace rather than executed straight away. When the contents of a non-fresh tensor is requested, the whole trace is flushed. This gives the same eager-mode semantics that users expect, but enables optimizations and fusion of operations within a trace.

The key observation that makes Torchy possible is that the result of most PyTorch operations is not observed by programs: they are merely temporaries. Our study (Section 5) shows that this assumption holds in practice and that ML models execute many operations before observing their result.

Torchy achieves speedups of up to 12x in microbenchmarks when compared with plain PyTorch. For benchmarks with control-flow, Torchy outperforms TorchScript by up to 5x. Torchy offers performance that is comparable with or exceeds that of static compilation, while offering the usability of eager-model frameworks.

The main contributions of this paper are as follows:

- Identification of the sources of inefficiencies in ML workloads and shortcomings of the current solutions.
- The design and implementation of a tracing JIT for an ML eager-mode framework (PyTorch). It requires zero program changes, unlike other solutions.
- Techniques to increase the size of traces in ML workloads to increase optimization opportunities.
- Evaluation: we show that the presented solution outperforms baseline by up to 12x in microbenchmarks and that it shows promising results in ML workloads. We also quantify the impact of type/shape inference on several metrics over generated traces.

Given the achieved results, we believe that tracing JIT compilers will become standard in ML frameworks in the future, enabling research on dynamic models and models using operations not yet supported natively by frameworks.

2 OVERVIEW

In this section, we present by example how Torchy works. Consider the following example that creates two random 4x3 matrices, computes $(x + y) \cdot y$, and prints the result:

```python
import torch, torchy
torchy.enable()
x = torch.rand(4, 3)
y = torch.rand_like(x)
z = x.add(y)
z.mul_(y)
print(z)
```

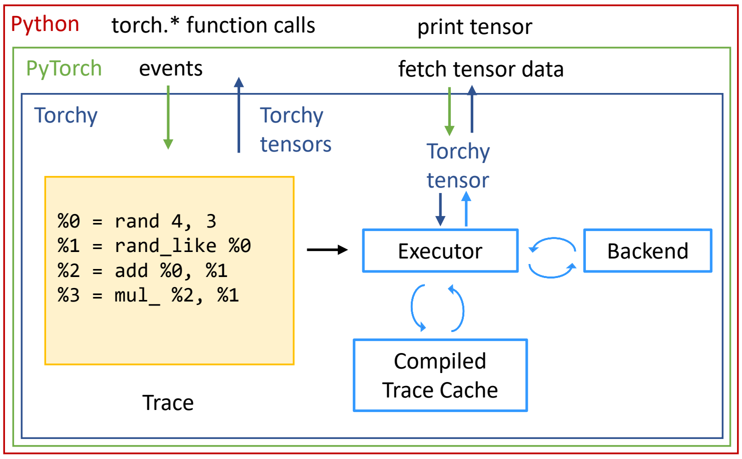
Operation `rand_like(x)` creates a new random tensor with the same shape and memory layout as `x`. It is PyTorch's convention to postfix the name of in-place operations with an underscore. Thus, `mul_` overrides the content of z with the result of the multiplication. In-place operations not only save temporary memory (useful to reduce memory peaks), but also potentially improve cache locality, as well as preserve the memory layout of the overridden operand.

When running the above program with vanilla PyTorch (i.e., without the `torchy.enable()` line), execution is done line-by-line as we have seen before. Remember that PyTorch is a Python library, and therefore each line of the program above is processed by Python, which then calls into PyTorch each operation at a time. PyTorch essentially acts as an interpreter and therefore cannot do any optimization across operations.

Torchy is a PyTorch extension, and hence it also depends on Python to discover operations over tensors. But instead of executing operations straight away, these are delayed and "shallow" tensors are returned instead. The returned tensors look and act as normal PyTorch tensors would, even if they may not have any content.

Figure 2 shows how Torchy executes our example program. When Python reaches the call to `rand`, it calls into PyTorch, which then generates an event that is intercepted by Torchy. Usually it would be handled by PyTorch itself, which would execute the operation and return the resulting tensor. Torchy, however, stores the event in the trace and returns a "shallow" tensor instead, which contains only the data type and shape information. No memory is allocated for the result.

Our trace IR syntax loosely follows that of LLVM. The first line of the trace indicates that register `%0` contains the result of executing operation the `rand` on trace's input 0 (`in<0>`). Note that we treat all operations, in place or not, uniformly: they all return a tensor even if it is just an alias to one of the input tensors. For example, `mul_` returns an alias to the first argument.



```
%0 = rand 4, 3
%1 = rand_like %0
%2 = add %0, %1
%3 = mul_ %2, %1
```

Fig. 2. Executing a PyTorch program with Torchy.

Python then moves to the `rand_like` call. PyTorch needs the shape information of the input to create a tensor with the same shape. Since we do type and shape inference, our "shallow" tensors contain this information and thus `rand_like` can be executed straight away.

When Python reaches the `print` statement, PyTorch generates an event requesting the contents of tensor `z`. Torchy gets this event and realizes the tensor is not up-to-date, i.e., there are operations pending in the trace that can *potentially* change the contents of the tensor. The trace must, therefore, be flushed. Note that the trace can be safely flushed at any time; for correctness purposes Torchy only needs to compute an over-approximation of tensors that are impacted by the operations pending in the trace.

Flushing the trace consists in executing every operation on the trace and updating the corresponding tensors appropriately. We employ a cache to store compiled programs for frequent traces. Each backend provides two functions: compilation of traces, and execution of compiled programs. It is the job of the backend to optimize traces, fuse operations, and so on.

Currently, Torchy supports only one backend, TorchScript, which is part of PyTorch. TorchScript has support for both CPUs and CUDA-enabled GPUs, and performs the standard optimizations, including operation fusion.

One feature not shown in our example trace is the tracking of externally observable tensors. Since Python is a reference-counted language, once a tensor goes out of scope in the user's program, PyTorch gets a notification. We track which tensors are just temporaries and not observable by the program anymore using this mechanism. Information about temporaries enables further optimizations, including operation fusion. We also detect which operations are not needed anymore so they can be removed from the trace.

After executing the trace, control returns back to PyTorch that can then print the tensor. Torchy delayed the execution of operations but still provided the same eager-mode semantics of PyTorch transparently.

3 TORCHY

In this section we explain how Torchy works in more detail. We start by formalizing our slightly modified version of PyTorch tensors, as well as traces. We then describe how operations are intercepted and how the trace is managed.

$$
\begin{array}{ll}
\text{dtype} & ::= \{\text{float}, \text{double}, \text{int32}, \text{bool}, \ldots\} \\
\text{shape} & ::= \mathcal{L}(\mathbb{N}_0) \\
\text{strides} & ::= \mathcal{L}(\mathbb{N}_0) \\
\text{traceidx} & ::= \mathbb{N} \uplus \{\text{None}\} \\
\text{tensorID} & ::= \mathbb{N} \\
\text{dataID} & ::= \mathbb{N} \\
\text{dataID?} & ::= \text{dataID} \uplus \{\text{None}\} \\
\text{refs} & ::= \mathbb{N} \\
\text{device} & ::= \{\text{cpu}, \text{gpu}_0, \text{gpu}_1, \ldots\} \\
\text{bytes} & ::= \mathcal{L}(\{0, \ldots, 255\}) \\
\text{materialized} & ::= \mathbb{B} \\
\text{tensor} & ::= \text{dtype} \times \text{shape} \times \text{strides} \times \text{traceidx} \times \text{dataID?} \times \text{refs} \\
\text{rawdata} & ::= \text{device} \times \text{bytes} \times \text{materialized} \times \text{refs} \\
\text{T} & ::= \text{tensorID} \rightharpoonup \text{tensor} \\
\text{RD} & ::= \text{dataID} \rightharpoonup \text{rawdata}
\end{array}
$$

Fig. 3. Type definitions of a tensor and raw data. Partial functions are denoted with $\rightharpoonup$.

3.1 Tensors

Figure 3 shows the definition of tensor and raw data. A tensor is a view over the raw data, which can be shared across several tensors. Without loss of generality, we only describe dense tensors; Torchy applies similarly to sparse tensors and other representations.

List shape contains the tensor's dimensions. For example, a 3x2 matrix has shape $= (3, 2)$. List strides determines how the data is stored, e.g., row- or column-major order. In row-major order (like C's arrays), a 3x2 matrix has strides $= (2, 1)$, meaning that to access the next element in a row we need to increment the pointer by one, while to access the same element in the following row we need to increment the pointer by two.

As an example, the matrix transpose operation in PyTorch is implemented as a view. Concretely, for an input tensor $t = (\text{dtype}, \text{shape}, \text{strides}, \text{traceidx}, \text{dataID}, \text{refs})$, it returns a new tensor t' with the same data but flipped shape, i.e.: $t' = (\text{dtype}, \text{rev}(\text{shape}), \text{rev}(\text{strides}), \text{traceidx}', \text{dataID}, 1)$, with rev being a function that reverses the input list. Because t' is just a view, changing an element of tensor t changes the corresponding transposed element of t' and vice versa.

Torchy tensors also contain a traceidx field that indicates the location in the trace if this tensor was the result of an operation that was delayed and not executed yet. We discuss the role of this field in the next section.

Each tensor optionally contains a pointer to a raw data tuple (dataID), which stores the data associated with the tensor in some device (CPU or GPU). There are very few exceptions when tensors do not have associated raw data in PyTorch.

Accessing an element of an n-dimensional tensor t, written as $t[i_1, \ldots, i_n]$ or $t[i_1][\ldots][i_n]$, accesses $d[\sum_{j=1}^{n} \text{strides}[j] \times i_j \times \text{sizeof}(\text{dtype})]$, with d being the raw data associated with t, i.e., $d = \text{RD}[\text{dataID}].\text{bytes}$. RD maps data ids to raw data, and sizeof is the size of a given data type in bytes.

Raw data lives in a specific device, either the CPU or a GPU. Note that although NUMA systems may have the RAM split across the CPUs, PyTorch considers all CPUs as a single device and parallelizes operations across all of them irrespective to which NUMA node the data is stored at.

Raw data contains a materialized bit to indicate whether the data is up-to-date or if some operation *possibly* impacting this data has been delayed and not executed yet. This bit is an addition of Torchy.

$$
\begin{array}{ll}
\text{opid} & ::= \{\text{add.Tensor}, \text{add.Scalar}, \text{matmul}, \ldots\} \\
\text{inputidx} & ::= \mathbb{N} \\
\text{arg} & ::= \text{traceidx} \cup \text{inputidx} \cup \text{device} \cup \text{dtype} \cup \text{scalar} \cup \ldots \\
\text{args} & ::= \mathcal{L}(\text{arg}) \\
\text{observable} & ::= \mathbb{B} \\
\text{op} & ::= \text{opid} \times \text{args} \times \text{observable} \\
\text{ops} & ::= \mathcal{L}(\text{op}) \\
\text{input} & ::= \text{tensor} \cup \text{generator} \cup \text{storage} \\
\text{optensor} & ::= \mathcal{P}(\text{tensorID}) \\
\text{optensors} & ::= \mathcal{L}(\text{optensor}) \\
\text{inputs} & ::= \mathcal{L}(\text{input}) \\
\text{trace} & ::= \text{ops} \times \text{optensors} \times \text{inputs}
\end{array}
$$

Fig. 4. Definition of trace. $\mathcal{P}$ is the power set operation.

Both tensors and raw data are referenced counted and automatically destroyed when the counter reaches zero. Torchy intercepts these destruction events.

PyTorch's tensors obviously track more information that the simplified definition we give in Figure 3, e.g., the corresponding Python's variable, or cached information that can be computed from data in our definition. For example, PyTorch caches whether the data is contiguous or not in a bit, even if that information can be computed from the data we store. Nevertheless, our tensor definition is sufficient to illustrate the ideas behind Torchy.

3.2 Trace

As shown in Figure 4, a trace is a triple containing 1) a list of operations (in-place or otherwise) that have been delayed, 2) a list of tensor ids corresponding to the tensors affected by each of the operations (listeners), and 3) a list of inputs to the trace. The length of ops and optensors is equal.

Each operation consists of an opid that indicates the operation. PyTorch has over 2,000 operations, including overloads such as add.Tensor and add.Scalar, which add two tensors, and a tensor and a scalar, respectively.

Operations then have a list of arguments. PyTorch has about 30 types of arguments. Constant arguments, such as scalars, are baked into the trace and included directly in the args list. As these are fixed, backends can take advantage of these when generating code.

Three types of arguments are not constant: tensors, pseudo-random number generators, and storage buffers. Traces have a separate list of non-constant inputs (the last element of the triple) to hold these. Tensors are further special cased, as these may either correspond to the result of a previous operation in the trace (tracked as traceidx), or a tensor external to the trace (tracked as inputidx, corresponding to the position in the inputs list).

Operations track one more bit of information: whether they are externally observable or not. PyTorch in particular runs over Python, which is referenced counted. Once a tensor goes out-of-scope, the program cannot observe it anymore. Torchy gets a notification when that happens and records that information. Operations that are not externally observable are just temporaries that can be deleted or more easily fused with other operations. Therefore, tracking external observability is important for optimizations.

We separate ops from optensors in the trace as ops are used for compilation, while optensors and inputs are used for trace execution.

3.3 Intercepting Operations via Tracing

Torchy intercepts all operations over tensors. When a program performs such an operation, Torchy returns a shallow tensor without computing its contents. For functional operations, Torchy returns a fresh tensor:

$$t = (\mathsf{dtype}, \mathsf{shape}, \mathsf{strides}, \mathsf{traceidx}, \mathsf{None}, 1)$$

where dtype is the inferred data type for the result of the operation, shape and strides are the inferred shape information (computed in a best-effort way as these are optional), traceidx = $|\mathsf{trace}.ops|$ the current length of the trace, carrying no data, and with reference count of 1.

Furthermore, the tensor is registered in the tensor map T with a fresh *id* and the operation is appended to the trace:

$$\begin{aligned}
&\mathsf{T} \longleftarrow \mathsf{T}[id \mapsto t] \\
&op = (opid, args, \mathsf{true}) \\
&\mathsf{trace} \longleftarrow (\mathsf{trace.ops} \cdot op, \mathsf{trace.optensors} \cdot \{id\}, input')
\end{aligned}$$

where *opid* is the intercepted operation's identifier, *args* its arguments, $f' = f[x \mapsto y]$ a function equal to f except that $f'(x) = y$, and $l \cdot e$ the result of appending e to list l. Tensor arguments that were not present in tensor.inputs are appended to this list, resulting in $input'$. That is, we share trace inputs across operations. Each tensor added to the input list gets its reference counter incremented.

In-place operations do not create a new tensor; they simply mark the overridden operand (*id*) as not fresh. The operation is appended to the trace as before (where RD maps dataID into raw data tuples):

$$\begin{aligned}
&\mathsf{T}[id].\mathsf{traceidx} \longleftarrow |\mathsf{trace}.ops| \\
&\mathsf{RD}[\mathsf{T}[id].\mathsf{dataID}].\mathsf{materialized} \longleftarrow \mathsf{false}
\end{aligned}$$

Updating the idx makes it easier to compute which operations are not referenced by other operations and thus can be removed from the trace. Subsequent operations that take this tensor as input will reference the last idx, which in turn references the previous in-place operation on that tensor and so on, trivially marking all in-place operations as necessary.

3.4 Accessing Tensor Data

Access to tensor's data/properties in PyTorch is done through C++ virtual methods. This indirection allows us to intercept such calls.

Shape information is inferred during tracing. Therefore, we can answer those queries without computing the result of the operation. However, data accesses require computing the result. Such functions first check if the tensor is fresh (i.e., for a tensor t, $t.\mathsf{dataID} \neq \mathsf{None} \wedge \mathsf{RD}[t.\mathsf{dataID}].\mathsf{materialized}$ is true). If not, the trace is flushed first so the tensor is updated.

The materialized bit is stored in the raw data rather than in the tensor directly as tensors can share data. For example, if two tensors have the same dataID, and the user does an in-place operation on one of the tensors, printing the other one must trigger a trace flush. Hence, the condition to detect whether a tensor is fresh must be derived from the raw data tuple as that is what is shared amongst tensor views.

3.5 Flushing the Trace

Flushing the trace amounts to computing the result of each of the operations and updating their respective output tensors (if any). An operation only needs to be executed if 1) it is externally observable (i.e., its respective optensors is non-empty), 2) its result is used by another operation in the trace that has to be executed, or 3) it has some side effect.

The trace is first compiled using the ops, optensors, and inputs data. This allows the backend to specialize the code for the specific tensor shapes of the input. The information of optensors is only used to determine which operations are externally observable and that constitute the set of outputs of the trace. The result of the compilation is stored in a cache.

The compiled function takes inputs and produces a set of tensors as output. Each tensor in optensors is then overridden with the output tensor. Note that this is a cheap operation as raw data (the big chunk) is not copied; it is a mere pointer copy (the dataID).

4 IMPLEMENTATION

Torchy is implemented in C++14 like PyTorch itself, and consists of about 3,000 lines of code. We have an additional 35,000 lines of code that are automatically generated by a Python script. This includes dispatch wrappers, redispatch handlers for the interpreter backend, and operation names (for debugging).

Torchy is implemented as a PyTorch extension and can be installed via, e.g., `pip install torchy`. To use Torchy, users only need to write:

```python
import torchy
torchy.enable()
x = torch.add(...)
...
```

Torchy is open-source and available at https://github.com/nunoplopes/torchy.

4.1 Data structures

For the trace we use a fixed-size buffer with capacity for up to 64 operations. In all the models we tried, we observed that traces were usually shorter than 64 operations (numbers shown later in Section 5.3.3). By using a fixed-size buffer we avoid (slow) memory allocations in the fast path, i.e., when executing a previously seen trace that we have already compiled and cached.

Similarly, optensors in the trace is a buffer with space for three tensors only. This data structure records which tensors share storage with an operation in the trace. In our benchmark suite we observed the common case does not have more than three aliases per tensor, so again we avoid memory allocations in the common case. When our assumption is violated, we have to flush the trace.

We add one extra field to both tensors and raw data structures as mentioned before. For tensors, we created our own `TorchyTensor` that inherits from PyTorch's `TensorImpl` and adds the extra traceidx field. Nevertheless, creating our class was already necessary to intercept events delivered through virtual methods (described below). For the materialized bit in raw data, we patched PyTorch's `StorageImpl` class to give us this extra bit for free as there were some bits available in padding. Therefore, we increase the tensor size by 8 bytes and do not increase the size of raw data.

For tensorID and dataID we use pointers directly. Therefore, maps T and RD are simple memory dereferences.

4.2 Event Interception

For correctness, Torchy has to intercept all observable events in a PyTorch program. PyTorch has three types of such events: dispatched events, C++ virtual methods, and non-interceptable. Furthermore, we decide to passthrough some events to keep the design simple.

4.2.1 Dispatched Events. These events are operations over tensors (e.g., element-wise addition and subtraction, matrix multiplication) and they go over PyTorch's dispatcher. To intercept these operations, we generate a function for each of the PyTorch's tensor operations (aka ATen). Fortunately, PyTorch's source code contains a database with all operations including the types of the arguments and whether the operation is in-place or not.

We have three types of wrappers: 1) in-place operations, which register the operation on the trace, 2) functional operations, which register the operation on the trace, do type and shape inference, and return a new `TorchyTensor`, and 3) non-supported operations, which redispatch the operation straight away.

The wrappers for in-place operations also perform shape inference to record whether the shape changes or not. As we do not store the different shape in case it changes to avoid additional memory consumption, we have to flush the trace when this rare case happens.

Torchy only supports operations that return a single tensor. Support for operations that return multiple tensors is simple in principle (requires extending the definition of optensor) but since these operations are uncommon we leave it for future work. Operations that return something other than tensors (e.g., an integer) cannot be delayed without changes to the PyTorch API as it is not possible to overload primitive C++ types. We would need to be able to return a "delayed" integer object to Python, for example.

The behavior of PyTorch's dispatcher is influenced by both a global state and by the tensor arguments of the operation. This allows users to, e.g., have parts of their program run through Torchy and other parts run natively or with other methods. The easiest way of using Torchy, however, is by changing the global state using `torchy.enable()` to force all operations to be intercepted by Torchy.

4.2.2 C++ virtual methods. Internal access to tensors' properties (e.g., get tensor shape, get tensor raw data, change strides) is done via C++ virtual methods of the `TensorImpl` class. We overload this class via our own `TorchyTensor`.

When an overloaded method is called there are three cases: 1) the tensor has been already materialized, in which case we redirect the call to the native method, 2) we have inferred the information (e.g., the shape) and therefore we can provide that information ourselves, and 3) the tensor has not been materialized and we do not know how to reply to the query. In this last case we flush the trace to materialize the tensor.

In theory, `TorchyTensor` could answer all queries on its own except the request for the raw data, which amounts to executing the operation. However, to keep the design simple and the memory overhead low, we give up in some cases and flush the trace instead. For example, we do not allocate space in the `TorchyTensor` for the shape information; we reuse the space in `TensorImpl` instead. This solution works for most cases but breaks for the rare in-place operations that change the shape. Since we can only keep one shape at a time and we need to keep the original shape for when the operation is executed, we use a bit in `TorchyTensor` to indicate whether the shape information is fresh or not. We flush the trace when some data is accessed and is not fresh.

The cases where we give up tracking some of the tensor's information are rare, which justifies our tradeoff in saving memory vs sometimes having to cut a trace short earlier than needed.

4.2.3 Non-interceptable. Unfortunately PyTorch does not allow us to intercept all the necessary events. First, we have non-virtual methods in the `TensorImpl` class (e.g., data type and device). These are easy to account for; all we need is to compute this information during tensor creation.

The second class of events are essentially shortcomings in PyTorch's API. These are hard to debug as they usually only manifest by making a program run with Torchy produce different results than when run without. One case we found is when mixing Torchy and non-Torchy tensors, e.g., `some_tensor.set_(torchy_tensor)`. This operation overrides `some_tensor`'s data with that of `torchy_tensor`. The issue is that we cannot track future events through the C++ virtual methods on the non-Torchy tensor. We have patched PyTorch to fix the copy operation to go through the virtual methods, which will then trigger a trace flush.

Another example, also originating when running Torchy selectively, is when doing an in-place operation on a non-Torchy tensor. If the tensor's reference count is one, we replace the underlying `TensorImpl` with our own `TorchyTensor`. This allows us to intercept future events on this tensor. If the tensor is shared, we cannot replace it as we do not know the other references to patch them. Therefore, we have to flush the trace straight away as we would not be able to intercept virtual method events through the other references.

4.2.4 Passthrough Events. A fundamental part of the design of the trace is what is the language of events it supports. To keep the design simple and focused on the common case, we decided to keep the trace's language over tensor algebra operations only (PyTorch's ATen). Thus, we are unable to delay all events, and need to pass through some straight away and potentially flush the trace. We observe in benchmarks that these events are rare and therefore we argue this is a good tradeoff.

We passthrough the `TensorImpl::shallow_copy_from` method, which copies the properties (shape, etc) from another tensor. This contrasts with `shallow_copy` that essentially creates an alias for a tensor, and we fully support through the optensor set.

When copying from another tensor we may need to flush the trace straight away rather than keeping an alias. There are two cases: The first is when the tensor we are overriding is an input to the trace. We do not copy input tensors; we merely keep a reference. Therefore, any change in place that cannot be delayed through the trace mechanism must trigger a trace flush. The second case is when the tensor being overridden is not materialized. We must flush the trace first, as otherwise we would end up undoing the copy when materializing the tensor.

4.3 Data Type and Shape Inference

In PyTorch, all tensors must have a data type as the access to this information is not observable. Therefore, we need to compute the data type for all tensors created for delayed operations. Shape information, however, is optional as accesses to it are observable. Nevertheless, we try to infer shape information whenever possible to increase the size of traces.

For most cases, data type inference is simple: an addition of two floats yields a float. We have promotion rules for when the operands have different types. There are also operations that return a fixed type. For example, `argmax` always returns an integer-typed tensor regardless of the type of the input.

To reduce the implementation effort, we implemented a program that calls all the roughly 2,000 PyTorch operations with several combinations of parameters. For example, the program calls the addition operation with two tensors of all the possible data types and samples of tensor shapes (impossible to cover all as the number of possibilities is infinite). The program then selects one of the functions we hand-wrote that covers the input/output behavior of each of the functions.

The type inference results are sound, as our program covers all combinations of input tensor types. However, it is not for shape inference. As the number of shapes is infinite, we have to sample the input space. The used shapes were crafted manually to trigger all the known corner cases, and therefore we expect the resulting shape inference functions to be correct in practice. Moreover, when Torchy is run in debug mode, it checks whether the inferred type/shape are correct. While we did have some issues early in the development, we have not triggered any assertion for months while running dozens of models.

We wrote 27 functions for type inference and 29 functions for shape and strides inference, for a total of about 600 lines of code. These functions cover the majority of PyTorch operations.

During the development of Torchy we upgraded from PyTorch 1.9 to 1.10 and then to 1.11, and no new functions were required. We just had to rerun the tool to find the right inference function for the new operations, which made the upgrading effort very low.

This tool has the unwanted side-effect of fuzzing PyTorch's ATen API, which had not been done before. Obviously it triggered several crashes in some operations when using uncommon combinations of tensor types and shapes. We also uncovered a bug in the promotion of complex and double types, where the implementation of some operations (but not all) deviates from the documentation.[1]

After we developed our tool to find the right data type/shape inference function for each operation, PyTorch gained a new API for inferring this very same information. Meta tensors behave like ordinary tensors except that they do not have storage and therefore operations on them only compute the shape information and the other properties and skip computing the raw data result. The implementation of meta tensors is ongoing, and the coverage is still limited to about 10% of the operations. We are in discussions with the PyTorch team to use our tool to generate the implementation of meta tensors as we can automatically infer the right functions for most of the operations. Nevertheless, early results show that this API is too slow for our uses as it allocates several chunks of temporary memory internally. On the other hand, our design strives to avoid allocating memory dynamically.

4.4 Backends

We implemented two backends: an interpreter and compilation through TorchScript. The interpreter is straightforward: it redispatches one operation at a time using PyTorch's dispatch mechanism. It performs no optimizations. The TorchScript backend produces unoptimized TorchScript IR, which is the same SSA-based representation produced by TorchScript tracing and compilation modes described previously.

Traces are compiled the first time they are executed, and the compiled code is cached for subsequent executions. The compilation step for the interpreter is a no-op.

TorchScript performs several classic compiler optimizations (dead code elimination, constant propagation, etc), as well as operation fusion, which is the main driver of performance for machine learning models.

5 EVALUATION

5.1 Setup

Experiments were run on an Azure NC12s v3 VM, which provides 12 cores of Intel Xeon E5-2690 v4 (Broadwell) CPUs, 224 GB of RAM, and two nVidia Tesla V100 GPUs. RAM was not a bottleneck in any experiment.

We used a development version of PyTorch 1.11, git hash `e52d0e77`. PyTorch was compiled with Intel MKL 2020.4, LLVM 10 (for the JIT), and CUDA 11.5. Benchmarks used torchvision 0.9.1 and Hugging Face's transformers 4.11.3.

PyTorch was configured to run CPU operations in parallel across all cores and GPU operations in a single GPU. All benchmarks were run with 32-bit floats so they could be run natively on the CPU as well.

[1]https://github.com/pytorch/pytorch/issues/60941

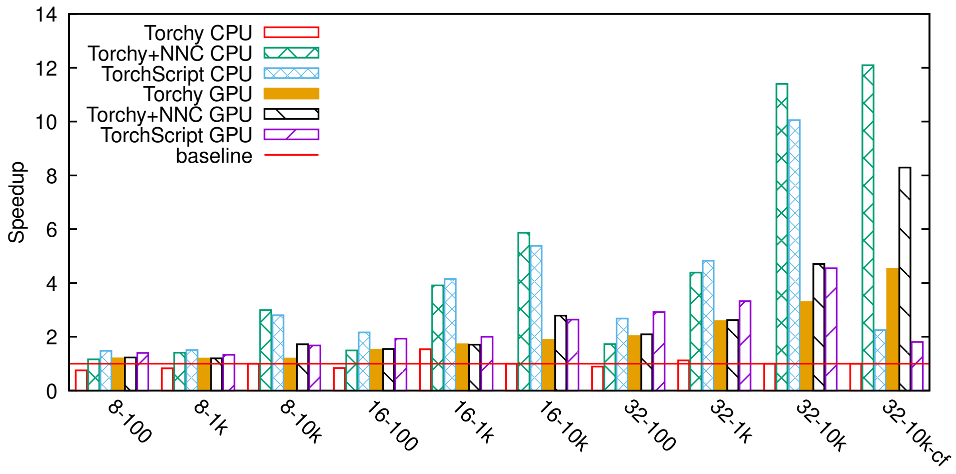


Fig. 5. Speedup of microbenchmarks with 8, 16, and 32 elementwise operations over square matrices with $n = 100, 1000, 10000$. The last benchmark on the right contains control-flow (cf); the others are straight-line code.

5.2 Microbenchmarks

We start by comparing the performance of Torchy in microbenchmarks against PyTorch and TorchScript on CPUs and GPUs. We run a simple program with 8, 16, and 32 elementwise operations (a mix of addition, subtraction, multiplication, and division). Each program is run with square matrices with $n = 100$, $n = 1000$, and $n = 10000$.

Programs run the set of operations in a loop for at least a thousand times (we adjust the number of iterations to keep the running time roughly constant across all programs to keep benchmarking time reasonable). We force trace flushing between loop iterations, which makes the trace size equal to the number of operations mentioned above. This setup represents a best-case scenario where we encounter the same trace many times.

We run Torchy with the TorchScript backend twice for each device: once with the NNC compiler enabled, and another with it disabled. The NNC compiler fuses operations and JIT compiles the resulting fused operations using LLVM for CPUs and CUDA's compiler for GPUs. Both TorchScript and NNC are part of PyTorch.

Figure 5 shows the speedup for each of the programs. Firstly, we observe that the overhead of Torchy is low: speedup varies between 0.75x for the smallest benchmark to 1.5x. Even when running Torchy without optimizations we observe performance improvements in some benchmarks. This is due to cache effects: baseline PyTorch goes back-and-forth to Python between operations, which makes parts of tensors to be evicted from the cache. Torchy, on the other hand, executes all the operations in sequence thus reducing the cache trashing effect.

As expected, the longer the traces and the larger the input the better. Longer traces expose more opportunities for fusion. In fact, NNC fuses our benchmark traces into a single loop, thus it eliminates the use of temporaries and improves cache locality. This explains the 12x speedup on the CPU.

Finally, we are interested in comparing the performance of Torchy and TorchScript. Internally they both use the same NNC compiler, which explains why they get very similar speedups. It is worth noting that Torchy requires zero changes to the program, while TorchScript requires users

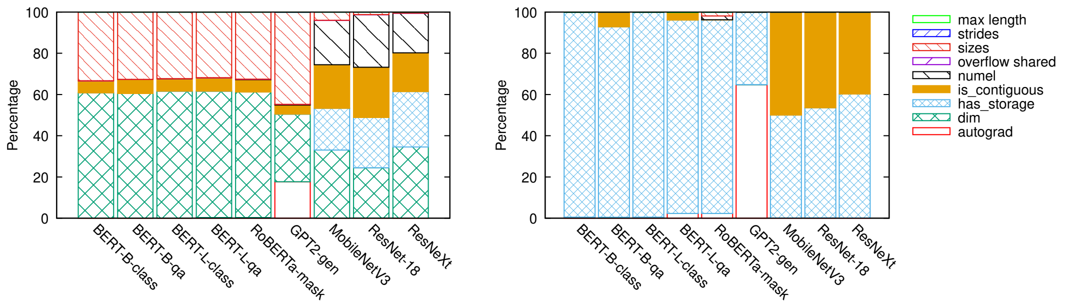


Fig. 6. Flush reasons without (left) and with (right) shape inference. The flush reasons are as follows (in order of the legend): reached maximum trace size, unknown stride, unknown shape, reached limit of tensor aliases, unknown number of elements, unknown data contiguity, missing storage data (not inferred), unknown number of dimensions, missing gradient (not inferred).

to annotate the code to mark the blocks that should be compiled. Torchy hence achieves similar performance without any user intervention.

Torchy outperforms TorchScript significantly on the rightmost benchmark of Figure 5. While all the other benchmarks consist of straight-line code, which is the best-case scenario for TorchScript, this control-flow (`cf`) benchmark contains an `if` statement with different operations in the two branches. The NNC compiler does not support control-flow, hence TorchScript gets limited fusion in this benchmark, explaining the poor performance. Torchy, on the other hand, creates two traces for the function so that NNC can fuse all operations.

TorchScript has an alternative fusion algorithm to NNC for nVidia GPUs named NVFuser. We compared the performance with NNC, but NVFuser was never faster than NNC, and thus we omit the results due to lack of space. Nevertheless, NVFuser has been under heavy development for the past year, and thus we expect the situation to change in the near future.

5.3 Machine Learning Models

In the previous section we showed the speedup of Torchy in an idealized scenario. We now benchmark Torchy by running inference queries on a set of off-the-shelf machine learning models. These are popular models which have been extensively optimized by hand already, and thus we do not expect significant speedups.

We run CNN models from torchvision (ResNet-18 [20], ResNeXt [52], MobileNetV3-Large [21]) and transformer models from Hugging Face (BERT Base/Large [12], GPT-2 [45], RoBERTa Large [35]). For image tasks we used the MNIST dataset (handwritten digits). For text tasks we used sentences and questions from Wikipedia. We used pre-trained weights from the respective packages.

5.3.1 Impact of Shape Inference on Flush Reasons. First we investigate what is the impact of doing shape inference. We expect traces to get longer as we potentially reduce the number of flushes when only shape information is accessed.

Figure 6 shows the frequency of each reason that triggered a trace flush with and without shape inference. We observe that for the BERT family our shape inference handles most of the queries, and thus the dominant flush reason becomes access to tensors' data, which is exactly what we want. Access to a tensor's data requires computing the result of the operations that impact that tensor, and thus is the only query that in principle we cannot avoid flushing the trace.

The non-BERT models also show a substantial reduction in flushes for missing shape information. However, they miss stride information (required for is_contiguous). The reason is twofold: we did

Benchmark	Without shape inference	With shape inference	Reduction
BERT-B-class	88	33	63%
BERT-B-qa	40	19	53%
BERT-L-class	88	33	63%
BERT-L-qa	40	19	53%
RoBERTa-mask	88	42	52%
GPT2-gen	247	96	61%
MobileNetV3	2124	2092	1.5% (26%)
ResNet-18	2048	2038	0.5% (21%)
ResNeXt	2065	2048	0.8% (26%)

Table 1. Number of unique traces, with and without shape inference. In the rightmost column we show the percentage of reduction of the number of traces when doing shape inference. In parentheses, we give the percentage ignoring the 2,000 single-use traces that should have been de-optimized.

not implement stride inference for all operations, and PyTorch has a performance bug in the `arange` operation that makes it produce an intermediate zero-sized tensor regardless of the final shape (which is later resized). Since we do not keep extra storage for shape/stride information, we require PyTorch to get this information right for the intermediate tensors, otherwise we have to give up shape/stride inference. This behavior is a performance bug in PyTorch and a fix is underway.

For the GPT2 benchmark, the dominant flush reason is the lack of gradient information. Our implementation is very conservative and flushes the trace straight away whenever any information about a gradient is requested. However, in most cases, PyTorch is merely accumulating operations on the gradient tape. These can be delayed and in theory do not need to trigger a trace flush. We are in contact with PyTorch developers in order to implement a solution that will allow us to delay gradient accumulation efficiently.

5.3.2 Impact of Shape Inference on Number of Traces. Table 1 shows the number of unique traces for our benchmarks, with and without shape inference. Even if we do not have complete coverage for shape inference, enabling it gives a very expressive reduction.

The torchvision benchmarks have a much higher number of traces. These benchmarks consist in 2,000 inference queries over images from the MNIST dataset. For each image, we intercept an operation of the following form:

```
%0 = select.int in<0>, 0, i
```

where i is the loaded image number. Thus, the number of traces grows linearly with the number of inference queries instead of being constant like in Hugging Face's benchmarks. A simple option would be to special case the handling of `select.int` to avoid baking the constant in the trace and rather pass it as an input to the trace (like we do for tensors). Another is to implement de-optimization, where similar traces would be merged and some constants promoted to inputs. We leave this for future work.

In Table 1 we also give the reduction percentage when ignoring the 2,000 single-use traces. This is a more meaningful number for comparing with the other benchmarks.

5.3.3 Impact of Shape Inference on Trace Size. Figure 7 shows the impact of shape inference on trace's sizes. The percentages given are relative to execution frequency. For example, about 71% of the traces executed for ResNeXt without shape inference had a single operation.

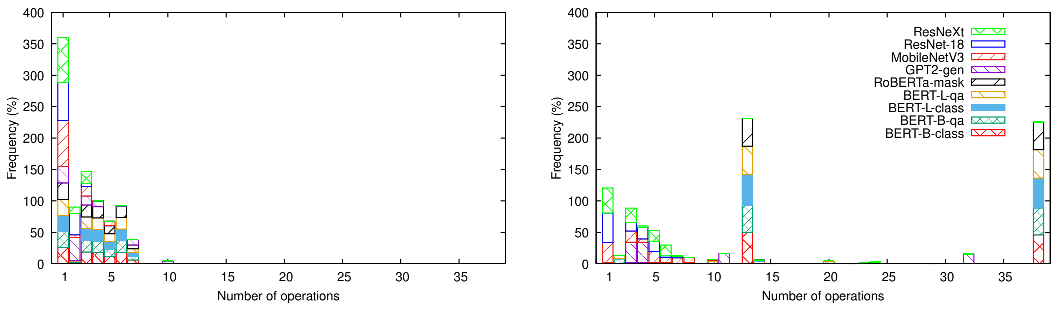


Fig. 7. Execution frequency of each trace size (number of operations) per benchmark without shape inference (left) and with shape inference (right).

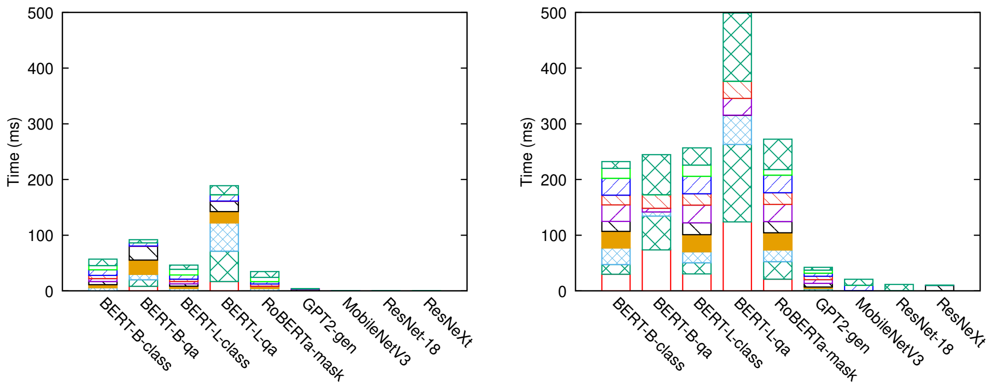


Fig. 8. Time (in ms) taken to execute the ten most frequent traces of each benchmark using Torchy with NNC, without (left) and with (right) shape inference.

We observe that shape inference increases trace size significantly. Without inference the maximum size was 10 operations, while with inference it is 38. Traces are longer for the BERT models, which is consistent with the fact that we have better inference coverage for these models. For example, GPT-2 improves with shape inference, as we can see the distribution of trace sizes more towards the right, but the maximum size is 32, and the most frequent is only of size 4.

Models from torchvision have their most frequent traces with size 1 only because of the `select.int` issue explained before.

5.3.4 Impact of Shape Inference on Trace Run Time. Figure 8 shows the running time of the ten most frequent traces for each benchmark, with and without shape inference. The most frequent trace is at the bottom, and the tenth-most frequent at the top.

First, we observe that shape inference has a very significant impact on the time each trace takes to execute. We had already seen that traces become larger, and now we confirm that the running time increases as well. This is good as longer-running traces hide the overhead of tracing more easily.

For example, for BERT large, we have multiple traces that take more than 100ms to execute. RoBERTa shows the largest improvement, with the top ten traces taking almost 8x longer to execute when doing shape inference.

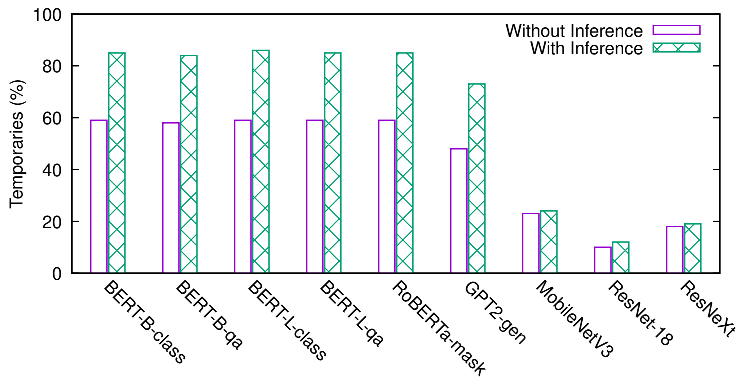


Fig. 9. Percentage of tensor operations that are temporaries, i.e., not observable by the program, aggregated over all traces. The higher the better.

Secondly, we observe that although shape inference has an impact on GPT2 and torchvision benchmarks, the running time is still fairly low. One of the reasons is that many of the top traces for these benchmarks have just one or two operations. Thus, we need to further improve shape inference to avoid unnecessary flushes in these benchmarks.

5.3.5 Impact of Shape Inference on Number of Temporaries. For the sake of optimization, it is better when the result of most operations in a program are not directly observable by the program, i.e., they are merely temporaries. This allows the compiler to reorder operations, rewrite arithmetic expressions to more efficient variants, and so on.

Figure 9 shows the percentage of tensor operations that are temporaries, aggregated across all executed traces. The higher the better, but it will never be 100% as traces must have at least one observable output or otherwise the trace would not need to be executed at all.

We observe that shape inference improves the percentage of temporaries considerably. The exception, as usual, are the torchvision benchmarks because they have many single-operation traces, which have 0% of temporaries.

Nevertheless, there is still room for improvement, as in benchmarks with traces of size 38, we would like to have just a single output, giving us 97% of temporaries. Our best benchmarks have 86% of temporaries, which is 11 percentage points away from the optimum.

5.3.6 Performance. We show the speedup of Torchy in Figure 10, with and without operation fusion (NNC). We do not show numbers for TorchScript as the model implementations we used do not natively support it.

Unsurprisingly, Torchy only has a speedup over one in a single case. There are several reasons for the poor performance, especially when considering the good results in microbenchmarks.

Firstly, we observe that the performance in the torchvision benchmarks is lower than those of Hugging Face. The main reason is that the way that torchvision loads the dataset triggers an explosion in the number of traces (the `select.int` issue). This induces a significant overhead in terms of compilation time.

Secondly, the implementation of the models we used has been heavily optimized by hand over the years. Obviously, there are few opportunities left for the compiler. Moreover, PyTorch has already added native implementations for the most expensive operations in these models. Therefore, the opportunities left for the compiler to optimize are mostly in the cheaper parts of the code. Nevertheless, the NNC compiler improves performance over baseline Torchy.

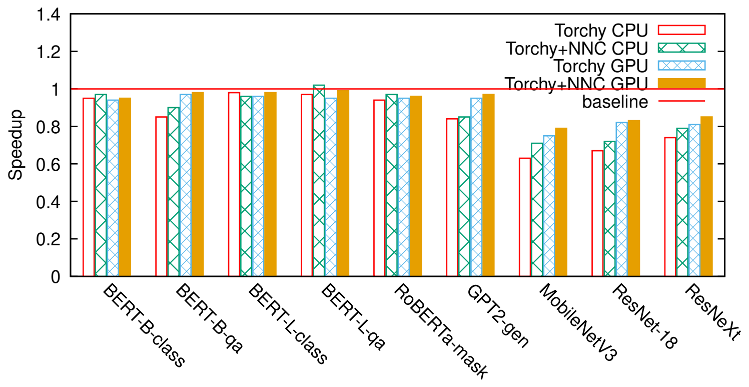


Fig. 10. Speedup of Torchy (with shape inference) in several inference tasks on a diverse set of standard machine learning models.

Finally, the NNC compiler itself is still in early stages. We observed multiple opportunities for improvement in the benchmarks:

- No support for fusing in-place operations. This is fundamental as in-place operations reduce peak memory consumption and improve cache behavior.
- No support for changing functional into in-place operations whenever possible (i.e., when the overridden tensor is a temporary as it is not used further).
- No support for merging equivalent operations. We observed equivalent `view` operations in a trace that, if canonicalized, would expose opportunities for merging multiple matrix multiplications.
- No support for fusing operations with operands placed in different devices, even if the semantics is to temporarily move the operands together into a same device (e.g., the semantics could be emulated with additional copy operations).
- The fusion algorithm does not support batched matrix multiplications (even on the trivial case with a single batch). These are very common in transformer models.
- TorchScript does not support the equivalent of C++'s call by r-value reference. If we detect a trace's input tensor is not externally observable anymore, we could mark it as a temporary in the TorchScript function if it supported such functionality. This would enable TorchScript to change functional operations over these tensors to their in-place equivalents.

We are in discussions with the TorchScript team to improve their compiler based on our findings. These improvements are, however, orthogonal to this paper, which focuses on producing traces and executing them with *some* backend.

A final positive note is that the overhead of PyTorch is fairly low. For the Hugging Face models, we see overheads around 2%, with the worst case being 23%. This result is very encouraging as it shows that a good optimizer will enable Torchy to outperform baseline easily as there is little overhead to offset.

6 RELATED WORK

The work closest to ours is LazyTensor [50]. Like Torchy, it is a tracing compiler for PyTorch. It was developed mostly by Google in parallel to Torchy to support their TPU [28] in PyTorch. The TPU (as well as other specialized devices like GraphCore's IPU [27]) have long compilation times even for single operations, which means the overhead of the standard PyTorch eager mode is too high (much higher than CUDA GPUs). LazyTensor does not support in-place operations natively (they are converted to functional), unlike Torchy which fully supports them. This results in LazyTensor consuming more memory than Torchy, as well as being slower. Furthermore, LazyTensor's internal IR is graph-based, while ours is a simple fixed-size buffer. Thus, their design incurs in significant overhead from constant memory allocations and pointer chasing. On the other hand, LazyTensor natively supports capturing very long traces and even single-trace models. Combining both approaches is likely a good idea that we leave for future work.

LazyTensor is still in development and its code base is already an order of magnitude larger than Torchy's, which suggests that Torchy's design is better. The design of Torchy was optimized using a diverse set of benchmark models, for which we present several useful metrics in this paper.

TeaTorch [24] is a static type checker for PyTorch. Like Torchy, it infers the shapes of tensors, but it uses an SMT solver (Z3) to do so. TeaTorch can potentially infer more shapes than Torchy, specifically those of more dynamic operations. However, SMT solvers are too slow for our use.

BELE [54] is a lazy evaluator for numeric Python packages (NumPy, Pandas, Spark). Because it is a generic framework, it incurs in significantly more overhead than Torchy. It also requires a lot of manual work in order to support the APIs made lazy (unlike Torchy). On the other hand, BELE supports API redirection (e.g., from Pandas to Spark).

DTR [31] is a dynamic tensor rematerialization library for PyTorch that uses a greedy algorithm to decide when to keep temporaries in memory and when to recompute them. Like Torchy, DTR intercepts all PyTorch operations.

DyNet [38] is an eager-mode framework that support lazy evaluation. Terra [30] creates a trace DAG and attempts to synchronize Python programs and traces by finding cut-points heuristically.

JAX [15] and TensorFlow [2] both support tracing similar to TorchScript's tracing mode.

AutoGraph [36] is a compiler for TensorFlow's eager mode that rewrites imperative code with control-flow into data-flow graph creating code. The goal is the same as TorchScript's compiler, but works instead by manipulating Python programs' ASTs directly. TorchDynamo [13] works at Python bytecode level and captures sequences of tensor operations. Janus [23] is also a compiler for TensorFlow's eager mode but uses profiling and speculative execution to deal with control flow and type inference.

XLA [11] is TensorFlow's whole-program compiler, which performs graph rewrites, fusion and scheduling of operations, picks layout for tensors, etc. Glow [48] is similar in goals, but targets PyTorch. The ONNX Runtime [10] takes ONNX graphs as input, which can be generated by most ML frameworks. All three require the whole ML model's graph to be provided somehow (usually by hand using techniques like tracing and others).

Several algorithms have been developed for fusing operations in ML graphs, such as DeepCuts [29] and DNNFusion [39]. Weld [40] optimizes data movements across data-intensive operations.

Xtat [44] tunes the heuristics of XLA using a variety of search techniques. Substantial speedups are achieved by tuning the heuristics of the fusion algorithm, layout of tensors, and so on. Astra [49] uses profiling information to benchmark multiple variations at a time.

There are several DSLs for the implementation of kernels, usually exposing an Einstein-like notation instead of allowing explicit control-flow [33]. Schedules for the generated loops (e.g., tiling parameters) are either specified manually or obtained automatically by running with many parameters and picking the best (so called auto-tuning). For example, AKG [55], Polly [18], Tensor Comprehensions [51] and Tiramisu [3] use polyhedral compilation techniques. Halide [47] uses a simpler interval analysis to compute the iteration space for each loop. taco [32] supports a variety of sparse tensor formats. AutoTVM [9] learns how to optimize kernels written in TVM [8] using a machine learning model. SDFG [5] exposes a graphical interface for developers to explore the impact of graph transformations on their kernels. RLibm [34] synthesizes correctly rounded floating-point implementations of math functions, while most other compilers for ML workloads often ignore precision. These tools are orthogonal to Torchy and could be used for generating code for fused operations from the traces produced by Torchy.

Another technique for generating kernels is through sketching where most of the code is written by hand (including loops), but some key difficult parts are left for the compiler. For example, Swizzle Inventor [43] synthesizes index expressions for array indexing, including complex data shuffles.

TASO [25] is a superoptimizer for ML compilers that synthesizes peephole optimizations automatically. MetaFlow [26] uses a backtracking algorithm to optimize ML programs so that it can try transformations that do no improve performance locally but may improve performance globally. OCGGS [14] explores multiple graph rewriting search techniques such as dynamic programming and sample-based.

Tracing JIT compilers are used in a variety of contexts. For example, Dynamo [4] and Valgrind [37] use tracing to discover binary code and to handle self-modifying code. Several tracing compilers exist for static and dynamic languages, including HotpathVM [17] and trace-JIT [22] for Java, TraceMonkey [16] for JavaScript, and PyPy [6] for Python. These compilers usually trace across function boundaries like Torchy, but they operate at a much lower level. Torchy traces tensor operations, not individual programming language/assembly operations, which is good as tensor operations take longer to execute, so we have less overhead. On the other hand, the fact that we do not have access to the program control-flow makes it harder to decide when to flush a trace as we cannot rely on the program's natural cut-points like branches and loop heads.

Torchy's tracing serves mainly two purposes: code discovery and code specialization for observed tensor shapes. This is a blend of the goals of tracers for binary code and dynamic languages.

7 DISCUSSION AND FUTURE WORK

We believe that Torchy is a good step in the direction of improving performance of eager-mode ML frameworks such as PyTorch, even if the performance with real models is not great yet. In the evaluation section we discussed several opportunities for improvement in the backend we used (NNC), which are orthogonal to this paper. Nevertheless, traces are the holy grail for compilers, even more so for those specialized in tensor operations, as our traces consist of straight-line tensor operations only. We should see better performance as we plug in more backends to foster competition.

We note that the benchmark models we used are close to the worst-case scenario. These models have been designed with the constraints of the current generation of ML frameworks in mind. Similarly, frameworks like PyTorch have been extensively optimized for these popular models, leaving us with few optimization opportunities. Nevertheless, it is important that we work on a general solution to improve the performance of ML workloads that does not require tweaking models and frameworks together as this defeats the purpose of having frameworks altogether. We believe the solution lies in JIT tracing, as the microbenchmarks attest.

Related to this paper, there are some considerations that are worth exploring in future work. For example, Torchy always flushes whole traces at once. However, we could flush just portions of it (the cone-of-influence of the relevant tensor). But there are trade-offs in terms of fusion and other optimization opportunities, as well as latency.

ML models execute most traces millions of times. It seems worthwhile to have a tiered compilation architecture with really expensive optimizations in the last tier, including possibly superoptimization.

Torchy still incurs in the unavoidable overhead of Python as it has to trace every single operation every time. It would be interesting to combine tracing of Python and of PyTorch in a same compiler to avoid this overhead.

Another possible avenue to reduce Torchy's overhead would be to integrate Torchy natively in PyTorch. Right now we incur in unnecessary overhead due to extra memory allocations of Torchy tensors (to replace PyTorch's original ones), which would be eliminated if Torchy was integrated. Moreover, the overhead when tracing is disabled is very small (just an extra branch on each tensor property access), which further justifies a full integration.

Torchy does not support precise exceptions, as we delay the execution of tensor operations completely. Even type/shape mismatches are delayed as our inference algorithm does not generate user errors. This difference in semantics is detectable by programs. However, PyTorch's CUDA backend already executes operations asynchronously, thus programs have already to be written with imprecise exceptions in mind. We believe precise exceptions are not needed for ML programs, although stopping immediately on typing errors may be a convenient feature to offer in the future.

We came across a case where de-optimization is clearly needed (for the torchvision benchmarks). It is unclear if further mechanisms are needed to tame trace specialization as tracing can potentially trigger a path explosion phenomenon as traces follow single paths. Similarly, when tracing loops we do not detect when an iteration starts or ends, which may lead to generating many traces for a loop, each starting at a different operation in the loop body. We have not observed any of these behaviors in practice, but further investigation is needed.

8 CONCLUSION

We presented Torchy, a tracing JIT compiler for PyTorch. Torchy solves the fundamental problem of code discovery for the compilation of machine learning codebases by tracing tensor operations at run time. Torchy works transparently with any codebase unmodified unlike other solutions.

Torchy offers the same eager semantics as PyTorch, while enabling optimizations such as operation fusion and code specialization through type and shape inference. These optimizations are not possible in PyTorch as it executes one operation at a time.

Torchy outperforms PyTorch by up to 12x in microbenchmarks, offering performance close to or exceeding that of data-flow frameworks and the convenience of eager-mode frameworks. Given these results, we believe tracing JITs will become standard in ML frameworks and enable the development of a new generation of models not supported well by current frameworks.

Acknowledgments. This work was partially supported by national funds through FCT, Fundação para a Ciência e a Tecnologia, under project UIDB/50021/2020.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*. https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

[2] Akshay Agrawal, Akshay Naresh Modi, Alexandre Passos, Allen Lavoie, Ashish Agarwal, Asim Shankar, Igor Ganichev, Josh Levenberg, Mingsheng Hong, Rajat Monga, and Shanqing Cai. 2019. TensorFlow Eager: A Multi-Stage, Python-Embedded DSL For Machine Learning. In *SysML*. https://mlsys.org/Conferences/2019/doc/2019/88.pdf

[3] 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 *CGO*. https://doi.org/10.1109/CGO.2019.8661197

[4] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. 2000. Dynamo: A Transparent Dynamic Optimization System. In *PLDI*. https://doi.org/10.1145/349299.349303

[5] Tal Ben-Nun, Johannes de Fine Licht, Alexandros N. Ziogas, Timo Schneider, and Torsten Hoefler. 2019. Stateful Dataflow Multigraphs: A Data-Centric Model for Performance Portability on Heterogeneous Architectures. In *SC*. https://doi.org/10.1145/3295500.3356173

[6] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 2009. Tracing the Meta-Level: PyPy's Tracing JIT Compiler. In *ICOOOLPS*. https://doi.org/10.1145/1565824.1565827

[7] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. *CoRR* abs/2005.14165 (2020). https://arxiv.org/abs/2005.14165

[8] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *OSDI*. https://www.usenix.org/system/files/osdi18-chen.pdf

[9] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to Optimize Tensor Programs. In *NeurIPS*. https://doi.org/10.5555/3327144.3327258

[10] ONNX Runtime developers. 2021. ONNX Runtime. https://onnxruntime.ai

[11] XLA developers. 2021. XLA: Optimizing Compiler for Machine Learning. https://www.tensorflow.org/xla

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL*.

[13] Facebook. 2022. TorchDynamo (under development). https://github.com/facebookresearch/torchdynamo

[14] Jingzhi Fang, Yanyan Shen, Yue Wang, and Lei Chen. 2020. Optimizing DNN Computation Graph Using Graph Substitutions. *Proc. VLDB Endow.* 13, 12 (July 2020), 2734–2746. https://doi.org/10.14778/3407790.3407857

[15] Roy Frostig, Matthew James Johnson, and Chris Leary. 2018. Compiling machine learning programs via high-level tracing. In *SysML*. https://mlsys.org/Conferences/2019/doc/2018/146.pdf

[16] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. 2009. Trace-Based Just-in-Time Type Specialization for Dynamic Languages. In *PLDI*. https://doi.org/10.1145/1542476.1542528

[17] Andreas Gal, Christian W. Probst, and Michael Franz. 2006. HotpathVM: An Effective JIT Compiler for Resource-Constrained Devices. In *VEE*. https://doi.org/10.1145/1134760.1134780

[18] Roman Gareev, Tobias Grosser, and Michael Kruse. 2018. High-Performance Generalized Tensor Operations: A Compiler-Oriented Approach. *ACM Trans. Archit. Code Optim.* 15, 3, Article 34 (sep 2018). https://doi.org/10.1145/3235029

[19] Horace He. 2021. Where do the 2000+ PyTorch operators come from?: More than you wanted to know. https://dev-discuss.pytorch.org/t/where-do-the-2000-pytorch-operators-come-from-more-than-you-wanted-to-know/373

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *CVPR*. https://doi.org/10.1109/CVPR.2016.90

[21] Andrew Howard, Mark Sandler, Bo Chen, Weijun Wang, Liang-Chieh Chen, Mingxing Tan, Grace Chu, Vijay Vasudevan, Yukun Zhu, Ruoming Pang, Hartwig Adam, and Quoc Le. 2019. Searching for MobileNetV3. In *ICCV*. https://doi.org/10.1109/ICCV.2019.00140

[22] Hiroshi Inoue, Hiroshige Hayashizaki, Peng Wu, and Toshio Nakatani. 2012. Adaptive Multi-Level Compilation in a Trace-Based Java JIT Compiler. In *OOPSLA*. https://doi.org/10.1145/2384616.2384630

[23] Eunji Jeong, Sungwoo Cho, Gyeong-In Yu, Joo Seong Jeong, Dong-Jin Shin, Taebum Kim, and Byung-Gon Chun. 2019. Speculative Symbolic Graph Execution of Imperative Deep Learning Programs. *SIGOPS Oper. Syst. Rev.* 53, 1 (jul 2019), 26–33. https://doi.org/10.1145/3352020.3352025

[24] Ho Young Jhoo, Sehoon Kim, Woosung Song, Kyuyeon Park, DongKwon Lee, and Kwangkeun Yi. 2022. A Static Analyzer for Detecting Tensor Shape Errors in Deep Neural Network Training Code. In *ICSE*. https://doi.org/10.1145/3510454.3528638

[25] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions. In *SOSP*. https://doi.org/10.1145/3341301.3359630

[26] Zhihao Jia, James Thomas, Todd Warszawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2019. Optimizing DNN Computation With Relaxed Graph Substitutions. In *SysML*. https://proceedings.mlsys.org/paper/2019/file/b6d767d2f8ed5d21a44b0e5886680cb9-Paper.pdf

[27] Zhe Jia, Blake Tillman, Marco Maggioni, and Daniele Paolo Scarpazza. 2019. Dissecting the Graphcore IPU Architecture via Microbenchmarking. *CoRR* abs/1912.03413 (2019). http://arxiv.org/abs/1912.03413

[28] Norman P. Jouppi, Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, and David Patterson. 2020. A Domain-Specific Supercomputer for Training Deep Neural Networks. *Commun. ACM* 63, 7 (jun 2020), 67–78. https://doi.org/10.1145/3360307

[29] Wookeun Jung, Thanh Tuan Dao, and Jaejin Lee. 2021. DeepCuts: A Deep Learning Optimization Framework for Versatile GPU Workloads. In *PLDI*. https://doi.org/10.1145/3453483.3454038

[30] Taebum Kim, Eunji Jeong, Geon-Woo Kim, Yunmo Koo, Sehoon Kim, Gyeongin Yu, and Byung-Gon Chun. 2021. Terra: Imperative-Symbolic Co-Execution of Imperative Deep Learning Programs. In *NeurIPS*. https://proceedings.neurips.cc/paper/2021/file/0b32f1a9efe5edf3dd2f38b0c0052bfe-Paper.pdf

[31] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. 2021. Dynamic Tensor Rematerialization. In *ICLR*. https://openreview.net/forum?id=Vfs_2RnOD0H

[32] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 77 (Oct. 2017). https://doi.org/10.1145/3133901

[33] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. 2021. The Deep Learning Compiler: A Comprehensive Survey. *IEEE Transactions on Parallel and Distributed Systems* 32, 3 (2021), 708–727. https://doi.org/10.1109/TPDS.2020.3030548

[34] Jay P. Lim and Santosh Nagarakatte. 2021. High Performance Correctly Rounded Math Libraries for 32-Bit Floating Point Representations. In *PLDI*. https://doi.org/10.1145/3453483.3454049

[35] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *CoRR* abs/1907.11692 (2019). http://arxiv.org/abs/1907.11692

[36] Dan Moldovan, James M Decker, Fei Wang, Andrew A Johnson, Brian K Lee, Zachary Nado, D Sculley, Tiark Rompf, and Alexander B Wiltschko. 2019. AutoGraph: Imperative-Style Coding With Graph-Based Performance. In *SysML*. https://mlsys.org/Conferences/2019/doc/2019/194.pdf

[37] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *PLDI*. https://doi.org/10.1145/1250734.1250746

[38] Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, Kevin Duh, Manaal Faruqui, Cynthia Gan, Dan Garrette, Yangfeng Ji, Lingpeng Kong, Adhiguna Kuncoro, Gaurav Kumar, Chaitanya Malaviya, Paul Michel, Yusuke Oda, Matthew Richardson, Naomi Saphra, Swabha Swayamdipta, and Pengcheng Yin. 2017. DyNet: The Dynamic Neural Network Toolkit. *CoRR* abs/1701.03980 (2017). https://arxiv.org/abs/1701.03980

[39] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. 2021. DNNFusion: Accelerating Deep Neural Networks Execution with Advanced Operator Fusion. In *PLDI*. https://doi.org/10.1145/3453483.3454083

[40] Shoumik Palkar, James J. Thomas, Anil Shanbhagy, Deepak Narayanan, Holger Pirky, Malte Schwarzkopfy, Saman Amarasinghey, and Matei Zaharia. 2017. Weld: A Common Runtime for High Performance Data Analytics. In *CIDR*.

[41] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NeurIPS*. https://papers.nips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf

[42] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep contextualized word representations. In *NAACL*.

[43] Phitchaya Mangpo Phothilimthana, Archibald Samuel Elliott, An Wang, Abhinav Jangda, Bastian Hagedorn, Henrik Barthels, Samuel J. Kaufman, Vinod Grover, Emina Torlak, and Rastislav Bodik. 2019. Swizzle Inventor: Data Movement Synthesis for GPU Kernels. In *ASPLOS*. https://doi.org/10.1145/3297858.3304059

[44] Phitchaya Mangpo Phothilimthana, Amit Sabne, Nikhil Sarda, Karthik Srinivasa Murthy, Yanqi Zhou, Christof Angermueller, Mike Burrows, Sudip Roy, Ketan Mandke, Rezsa Farahani, Yu Emma Wang, Berkin Ilbeyi, Blake Hechtman, Bjarke Roune, Shen Wang, Yuanzhong Xu, and Samuel J. Kaufman. 2021. A Flexible Approach to Autotuning Multi-Pass Machine Learning Compilers. In *PACT*. https://doi.org/10.1109/PACT52795.2021.00008

[45] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. (2019). https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf

[46] Ilija Radosavovic, Raj Prateek Kosaraju, Ross Girshick, Kaiming He, and Piotr Dollár. 2020. Designing Network Design Spaces. In *CVPR*. https://doi.org/10.1109/CVPR42600.2020.01044

[47] 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 *PLDI*. https://doi.org/10.1145/2491956.2462176

[48] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, James Hegeman, Roman Levenstein, Bert Maher, Nadathur Satish, Jakob Olesen, Jongsoo Park, Artem Rakhov, and Misha Smelyanskiy. 2018. Glow: Graph Lowering Compiler Techniques for Neural Networks. *CoRR* abs/1805.00907 (2018). http://arxiv.org/abs/1805.00907

[49] Muthian Sivathanu, Tapan Chugh, Sanjay S. Singapuram, and Lidong Zhou. 2019. Astra: Exploiting Predictability to Optimize Deep Learning. In *ASPLOS*. https://doi.org/10.1145/3297858.3304072

[50] 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. *CoRR* abs/2102.13267 (2021). https://arxiv.org/abs/2102.13267

[51] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary Devito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2019. The Next 700 Accelerated Layers: From Mathematical Expressions of Network Computation Graphs to Accelerated GPU Kernels, Automatically. *ACM Trans. Archit. Code Optim.* 16, 4, Article 38 (Oct. 2019). https://doi.org/10.1145/3355606

[52] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. 2017. Aggregated Residual Transformations for Deep Neural Networks. In *CVPR*. https://doi.org/10.1109/CVPR.2017.634

[53] Yuan Yu, Martín Abadi, Paul Barham, Eugene Brevdo, Mike Burrows, Andy Davis, Jeff Dean, Sanjay Ghemawat, Tim Harley, Peter Hawkins, Michael Isard, Manjunath Kudlur, Rajat Monga, Derek Murray, and Xiaoqiang Zheng. 2018. Dynamic Control Flow in Large-Scale Machine Learning. In *EuroSys*. https://doi.org/10.1145/3190508.3190551

[54] Guoqiang Zhang and Xipeng Shen. 2021. Best-Effort Lazy Evaluation for Python Software Built on APIs. In *ECOOP*. https://doi.org/10.4230/LIPIcs.ECOOP.2021.15

[55] Jie Zhao, Bojie Li, Wang Nie, Zhen Geng, Renwei Zhang, Xiong Gao, Bin Cheng, Chen Wu, Yun Cheng, Zheng Li, Peng Di, Kun Zhang, and Xuefeng Jin. 2021. AKG: Automatic Kernel Generation for Neural Processing Units Using Polyhedral Transformations. In *PLDI*. https://doi.org/10.1145/3453483.3454106