# Table of Contents

- [Contributing to PyTorch](#contributing-to-pytorch)

- [Developing PyTorch](#developing-pytorch)

- [Codebase structure](#codebase-structure)

- [Unit testing](#unit-testing)

- [Better local unit tests with pytest](#better-local-unit-tests-with-pytest)

- [Writing documentation](#writing-documentation)

- [Building documentation](#building-documentation)

- [Tips](#tips)

- [Building C++ Documentation](#building-c-documentation)

- [Previewing changes](#previewing-changes)

- [Submitting changes for review](#submitting-changes-for-review)

- [Adding documentation tests](#adding-documentation-tests)

- [Profiling with `py-spy`](#profiling-with-py-spy)

- [Managing multiple build trees](#managing-multiple-build-trees)

- [C++ development tips](#c-development-tips)

- [Build only what you need](#build-only-what-you-need)

- [Code completion and IDE support](#code-completion-and-ide-support)

- [Make no-op build fast](#make-no-op-build-fast)

- [Use Ninja](#use-ninja)

- [Use CCache](#use-ccache)

- [Use a faster linker](#use-a-faster-linker)

- [C++ frontend development tips](#c-frontend-development-tips)

- [CUDA development tips](#cuda-development-tips)

- [Windows development tips](#windows-development-tips)

- [Known MSVC (and MSVC with NVCC) bugs](#known-msvc-and-msvc-with-nvcc-bugs)

- [Running clang-tidy](#running-clang-tidy)

- [Pre-commit tidy/linting hook](#pre-commit-tidylinting-hook)

- [Building PyTorch with ASAN](#building-pytorch-with-asan)

- [Getting `ccache` to work](#getting-ccache-to-work)

- [Why this stuff with `LD\_PRELOAD` and `LIBASAN\_RT`?](#why-this-stuff-with-ld\_preload-and-libasan\_rt)

- [Why LD\_PRELOAD in the build function?](#why-ld\_preload-in-the-build-function)

- [Why no leak detection?](#why-no-leak-detection)

- [Caffe2 notes](#caffe2-notes)

- [CI failure tips](#ci-failure-tips)

## Contributing to PyTorch

Thank you for your interest in contributing to PyTorch! Before you begin writing code, it is important

that you share your intention to contribute with the team, based on the type of contribution:

1. You want to propose a new feature and implement it.

- Post about your intended feature in an [issue](https://github.com/pytorch/pytorch/issues),

and we shall discuss the design and implementation. Once we agree that the plan looks good,

go ahead and implement it.

2. You want to implement a feature or bug-fix for an outstanding issue.

- Search for your issue in the [PyTorch issue list](https://github.com/pytorch/pytorch/issues).

- Pick an issue and comment that you'd like to work on the feature or bug-fix.

- If you need more context on a particular issue, please ask and we shall provide.

Once you implement and test your feature or bug-fix, please submit a Pull Request to

https://github.com/pytorch/pytorch.

This document covers some of the more technical aspects of contributing

to PyTorch. For more non-technical guidance about how to contribute to

PyTorch, see the [Contributing Guide](docs/source/community/contribution\_guide.rst).

## Developing PyTorch

To develop PyTorch on your machine, here are some tips:

1. Uninstall all existing PyTorch installs:

```bash

conda uninstall pytorch

pip uninstall torch

pip uninstall torch # run this command twice

```

2. Clone a copy of PyTorch from source:

```bash

git clone https://github.com/pytorch/pytorch

cd pytorch

```

2.1. If you already have PyTorch from source, update it:

```bash

git pull --rebase

git submodule sync --recursive

git submodule update --init --recursive

```

If you want to have no-op incremental rebuilds (which are fast), see the section below titled "Make no-op build fast."

3. Install PyTorch in `develop` mode:

A full set of instructions on installing PyTorch from source is here:

https://github.com/pytorch/pytorch#from-source

The change you have to make is to replace

```bash

python setup.py install

```

with

```bash

python setup.py develop

```

This mode will symlink the Python files from the current local source

tree into the Python install. Hence, if you modify a Python file, you

do not need to reinstall PyTorch again and again. This is especially

useful if you are only changing Python files.

For example:

- Install local PyTorch in `develop` mode

- modify your Python file `torch/\_\_init\_\_.py` (for example)

- test functionality

- modify your Python file `torch/\_\_init\_\_.py`

- test functionality

- modify your Python file `torch/\_\_init\_\_.py`

- test functionality

You do not need to repeatedly install after modifying Python files.

In case you want to reinstall, make sure that you uninstall PyTorch first by running `pip uninstall torch`

and `python setup.py clean`. Then you can install in `develop` mode again.

## Codebase structure

\* [c10](c10) - Core library files that work everywhere, both server

and mobile. We are slowly moving pieces from [ATen/core](aten/src/ATen/core)

here. This library is intended only to contain essential functionality,

and appropriate to use in settings where binary size matters. (But

you'll have a lot of missing functionality if you try to use it

directly.)

\* [aten](aten) - C++ tensor library for PyTorch (no autograd support)

\* [src](aten/src) - [README](aten/src/README.md)

\* [TH](aten/src/TH)

[THC](aten/src/THC)

[THCUNN](aten/src/THCUNN) - Legacy library code from the original

Torch. Try not to add things here; we're slowly porting these to

[native](aten/src/ATen/native).

\* generic - Contains actual implementations of operators,

parametrized over `scalar\_t`. Files here get compiled N times

per supported scalar type in PyTorch.

\* [ATen](aten/src/ATen)

\* [core](aten/src/ATen/core) - Core functionality of ATen. This

is migrating to top-level c10 folder.

\* [native](aten/src/ATen/native) - Modern implementations of

operators. If you want to write a new operator, here is where

it should go. Most CPU operators go in the top level directory,

except for operators which need to be compiled specially; see

cpu below.

\* [cpu](aten/src/ATen/native/cpu) - Not actually CPU

implementations of operators, but specifically implementations

which are compiled with processor-specific instructions, like

AVX. See the [README](aten/src/ATen/native/cpu/README.md) for more

details.

\* [cuda](aten/src/ATen/native/cuda) - CUDA implementations of

operators.

\* [sparse](aten/src/ATen/native/sparse) - CPU and CUDA

implementations of COO sparse tensor operations

\* [mkl](aten/src/ATen/native/mkl) [mkldnn](aten/src/ATen/native/mkldnn)

[miopen](aten/src/ATen/native/miopen) [cudnn](aten/src/ATen/native/cudnn)

- implementations of operators which simply bind to some

backend library.

\* [quantized](aten/src/ATen/native/quantized/) - Quantized tensor (i.e. QTensor) operation implementations. [README](aten/src/ATen/native/quantized/README.md) contains details including how to implement native quantized operations.

\* [torch](torch) - The actual PyTorch library. Everything that is not

in [csrc](torch/csrc) is a Python module, following the PyTorch Python

frontend module structure.

\* [csrc](torch/csrc) - C++ files composing the PyTorch library. Files

in this directory tree are a mix of Python binding code, and C++

heavy lifting. Consult `setup.py` for the canonical list of Python

binding files; conventionally, they are often prefixed with

`python\_`. [README](torch/csrc/README.md)

\* [jit](torch/csrc/jit) - Compiler and frontend for TorchScript JIT

frontend. [README](torch/csrc/jit/README.md)

\* [autograd](torch/csrc/autograd) - Implementation of reverse-mode automatic differentiation. [README](torch/csrc/autograd/README.md)

\* [api](torch/csrc/api) - The PyTorch C++ frontend.

\* [distributed](torch/csrc/distributed) - Distributed training

support for PyTorch.

\* [tools](tools) - Code generation scripts for the PyTorch library.

See [README](tools/README.md) of this directory for more details.

\* [test](test) - Python unit tests for PyTorch Python frontend.

\* [test\_torch.py](test/test\_torch.py) - Basic tests for PyTorch

functionality.

\* [test\_autograd.py](test/test\_autograd.py) - Tests for non-NN

automatic differentiation support.

\* [test\_nn.py](test/test\_nn.py) - Tests for NN operators and

their automatic differentiation.

\* [test\_jit.py](test/test\_jit.py) - Tests for the JIT compiler

and TorchScript.

\* ...

\* [cpp](test/cpp) - C++ unit tests for PyTorch C++ frontend.

\* [api](test/cpp/api) - [README](test/cpp/api/README.md)

\* [jit](test/cpp/jit) - [README](test/cpp/jit/README.md)

\* [tensorexpr](test/cpp/tensorexpr) - [README](test/cpp/tensorexpr/README.md)

\* [expect](test/expect) - Automatically generated "expect" files

which are used to compare against expected output.

\* [onnx](test/onnx) - Tests for ONNX export functionality,

using both PyTorch and Caffe2.

\* [caffe2](caffe2) - The Caffe2 library.

\* [core](caffe2/core) - Core files of Caffe2, e.g., tensor, workspace,

blobs, etc.

\* [operators](caffe2/operators) - Operators of Caffe2.

\* [python](caffe2/python) - Python bindings to Caffe2.

\* ...

\* [.circleci](.circleci) - CircleCI configuration management. [README](.circleci/README.md)

## Unit testing

`hypothesis` is required to run the tests, `mypy` is an optional dependency,

and `pytest` may help run tests more selectively. All these packages can be

installed with `conda` or `pip`.

PyTorch's testing is located under `test/`. Run the entire test suite with

```bash

python test/run\_test.py

```

or run individual test files, like `python test/test\_nn.py`, for individual test suites.

### Better local unit tests with pytest

We don't officially support `pytest`, but it works well with our `unittest` tests and offers

a number of useful features for local developing. Install it via `pip install pytest`.

If you want to just run tests that contain a specific substring, you can use the `-k` flag:

```bash

pytest test/test\_nn.py -k Loss -v

```

The above is an example of testing a change to Loss functions: this command runs tests such as

`TestNN.test\_BCELoss` and `TestNN.test\_MSELoss` and can be useful to save keystrokes.

## Writing documentation

PyTorch uses [Google style](http://sphinxcontrib-napoleon.readthedocs.io/en/latest/example\_google.html)

for formatting docstrings. Length of line inside docstrings block must be limited to 80 characters to

fit into Jupyter documentation popups.

### Building documentation

To build the documentation:

1. Build and install PyTorch

2. Install the prerequisites

```bash

cd docs

pip install -r requirements.txt

# `katex` must also be available in your PATH.

# You can either install katex globally if you have properly configured npm:

# npm install -g katex

# Or if you prefer an uncontaminated global executable environment or do not want to go through the node configuration:

# npm install katex && export PATH="$PATH:$(pwd)/node\_modules/.bin"

```

3. Generate the documentation HTML files. The generated files will be in `docs/build/html`.

```bash

cd docs

make html

```

#### Tips

The `.rst` source files live in [docs/source](docs/source). Some of the `.rst`

files pull in docstrings from PyTorch Python code (for example, via

the `autofunction` or `autoclass` directives). To vastly shorten doc build times,

it is helpful to remove the files you are not working on, only keeping the base

`index.rst` file and the files you are editing. The Sphinx build will produce

missing file warnings but will still complete. For example, to work on `jit.rst`:

```bash

cd docs/source

ls | grep rst | grep -v index | grep -v jit | xargs rm

# Make your changes, build the docs, etc.

# Don't commit the deletions!

git add index.rst jit.rst

...

```

#### Building C++ Documentation

For C++ documentation (https://pytorch.org/cppdocs), we use

[Doxygen](http://www.doxygen.nl/) and then convert it to

[Sphinx](http://www.sphinx-doc.org/) via

[Breathe](https://github.com/michaeljones/breathe) and

[Exhale](https://github.com/svenevs/exhale). Check the [Doxygen

reference](http://www.stack.nl/~dimitri/doxygen/manual/index.html) for more

information on the documentation syntax.

We run Doxygen in CI (Travis) to verify that you do not use invalid Doxygen

commands. To run this check locally, run `./check-doxygen.sh` from inside

`docs/cpp`.

To build the documentation, follow the same steps as above, but run them from

`docs/cpp` instead of `docs`.

### Previewing changes

To view HTML files locally, you can open the files in your web browser. For example,

navigate to `file:///your\_pytorch\_folder/docs/build/html/index.html` in a web

browser.

If you are developing on a remote machine, you can set up an SSH tunnel so that

you can access the HTTP server on the remote machine from your local machine. To map

remote port 8000 to local port 8000, use either of the following commands.

```bash

# For SSH

ssh my\_machine -L 8000:my\_machine:8000

# For Eternal Terminal

et my\_machine -t="8000:8000"

```

Then navigate to `localhost:8000` in your web browser.

#### Submitting changes for review

It is helpful when submitting a PR that changes the docs to provide a rendered

version of the result. If your change is small, you can add a screenshot of the

changed docs to your PR.

If your change to the docs is large and affects multiple pages, you can host

the docs yourself with the following steps, then add a link to the output in your

PR. These instructions use GitHub pages to host the docs

you have built. To do so, follow [these steps](https://guides.github.com/features/pages/)

to make a repo to host your changed documentation.

GitHub pages expects to be hosting a Jekyll generated website which does not work

well with the static resource paths used in the PyTorch documentation. To get around

this, you must add an empty file called `.nojekyll` to your repo.

```bash

cd your\_github\_pages\_repo

touch .nojekyll

git add .

git commit

git push

```

Then, copy built documentation and push the changes:

```bash

cd your\_github\_pages\_repo

cp -r ~/my\_pytorch\_path/docs/build/html/\* .

git add .

git commit

git push

```

Then you should be able to see the changes at your\_github\_username.github.com/your\_github\_pages\_repo.

### Adding documentation tests

It is easy for code snippets in docstrings and `.rst` files to get out of date. The docs

build includes the [Sphinx Doctest Extension](https://www.sphinx-doc.org/en/master/usage/extensions/doctest.html),

which can run code in documentation as a unit test. To use the extension, use

the `.. testcode::` directive in your `.rst` and docstrings.

To manually run these tests, follow steps 1 and 2 above, then run:

```bash

cd docs

make doctest

```

## Profiling with `py-spy`

Evaluating the performance impact of code changes in PyTorch can be complicated,

particularly if code changes happen in compiled code. One simple way to profile

both Python and C++ code in PyTorch is to use

[`py-spy`](https://github.com/benfred/py-spy), a sampling profiler for Python

that has the ability to profile native code and Python code in the same session.

`py-spy` can be installed via `pip`:

```bash

$ pip install py-spy

```

To use `py-spy`, first write a Python test script that exercises the

functionality you would like to profile. For example, this script profiles

`torch.add`:

```python

import torch

t1 = torch.tensor([[1, 1], [1, 1.]])

t2 = torch.tensor([[0, 0], [0, 0.]])

for \_ in range(1000000):

torch.add(t1, t2)

```

Since the `torch.add` operation happens in microseconds, we repeat it a large

number of times to get good statistics. The most straightforward way to use

`py-spy` with such a script is to generate a [flame

graph](http://www.brendangregg.com/flamegraphs.html):

```bash

$ py-spy record -o profile.svg --native -- python test\_tensor\_tensor\_add.py

```

This will output a file named `profile.svg` containing a flame graph you can

view in a web browser or SVG viewer. Individual stack frame entries in the graph

can be selected interactively with your mouse to zoom in on a particular part of

the program execution timeline. The `--native` command-line option tells

`py-spy` to record stack frame entries for PyTorch C++ code. To get line numbers

for C++ code it may be necessary to compile PyTorch in debug mode by prepending

your `setup.py develop` call to compile PyTorch with `DEBUG=1`. Depending on

your operating system it may also be necessary to run `py-spy` with root

privileges.

`py-spy` can also work in an `htop`-like "live profiling" mode and can be

tweaked to adjust the stack sampling rate, see the `py-spy` readme for more

details.

## Managing multiple build trees

One downside to using `python setup.py develop` is that your development

version of PyTorch will be installed globally on your account (e.g., if

you run `import torch` anywhere else, the development version will be

used.

If you want to manage multiple builds of PyTorch, you can make use of

[conda environments](https://conda.io/docs/using/envs.html) to maintain

separate Python package environments, each of which can be tied to a

specific build of PyTorch. To set one up:

```bash

conda create -n pytorch-myfeature

source activate pytorch-myfeature

# if you run python now, torch will NOT be installed

python setup.py develop

```

## C++ development tips

If you are working on the C++ code, there are a few important things that you

will want to keep in mind:

1. How to rebuild only the code you are working on.

2. How to make rebuilds in the absence of changes go faster.

### Build only what you need

`python setup.py build` will build everything by default, but sometimes you are

only interested in a specific component.

- Working on a test binary? Run `(cd build && ninja bin/test\_binary\_name)` to

rebuild only that test binary (without rerunning cmake). (Replace `ninja` with

`make` if you don't have ninja installed).

- Don't need Caffe2? Pass `BUILD\_CAFFE2\_OPS=0` to disable build of

Caffe2 operators.

On the initial build, you can also speed things up with the environment

variables `DEBUG`, `USE\_DISTRIBUTED`, `USE\_MKLDNN`, `USE\_CUDA`, `BUILD\_TEST`, `USE\_FBGEMM`, `USE\_NNPACK` and `USE\_QNNPACK`.

- `DEBUG=1` will enable debug builds (-g -O0)

- `REL\_WITH\_DEB\_INFO=1` will enable debug symbols with optimizations (-g -O3)

- `USE\_DISTRIBUTED=0` will disable distributed (c10d, gloo, mpi, etc.) build.

- `USE\_MKLDNN=0` will disable using MKL-DNN.

- `USE\_CUDA=0` will disable compiling CUDA (in case you are developing on something not CUDA related), to save compile time.

- `BUILD\_TEST=0` will disable building C++ test binaries.

- `USE\_FBGEMM=0` will disable using FBGEMM (quantized 8-bit server operators).

- `USE\_NNPACK=0` will disable compiling with NNPACK.

- `USE\_QNNPACK=0` will disable QNNPACK build (quantized 8-bit operators).

- `USE\_XNNPACK=0` will disable compiling with XNNPACK.

For example:

```bash

DEBUG=1 USE\_DISTRIBUTED=0 USE\_MKLDNN=0 USE\_CUDA=0 BUILD\_TEST=0 USE\_FBGEMM=0 USE\_NNPACK=0 USE\_QNNPACK=0 USE\_XNNPACK=0 python setup.py develop

```

For subsequent builds (i.e., when `build/CMakeCache.txt` exists), the build

options passed for the first time will persist; please run `ccmake build/`, run

`cmake-gui build/`, or directly edit `build/CMakeCache.txt` to adapt build

options.

### Code completion and IDE support

When using `python setup.py develop`, PyTorch will generate

a `compile\_commands.json` file that can be used by many editors

to provide command completion and error highlighting for PyTorch's

C++ code. You need to `pip install ninja` to generate accurate

information for the code in `torch/csrc`. More information at:

- https://sarcasm.github.io/notes/dev/compilation-database.html

### Make no-op build fast

#### Use Ninja

By default, cmake will use its Makefile generator to generate your build

system. You can get faster builds if you install the ninja build system

with `pip install ninja`. If PyTorch was already built, you will need

to run `python setup.py clean` once after installing ninja for builds to

succeed.

#### Use CCache

Even when dependencies are tracked with file modification,

there are many situations where files get rebuilt when a previous

compilation was exactly the same.

Using ccache in a situation like this is a real time-saver. The ccache manual

describes [two ways to use ccache](https://ccache.samba.org/manual/latest.html#\_run\_modes).

In the PyTorch project, currently only the latter method of masquerading as

the compiler via symlinks works for CUDA compilation.

Here are the instructions for installing ccache from source (tested at commit

`3c302a7` of the `ccache` repo):

```bash

#!/bin/bash

if ! ls ~/ccache/bin/ccache

then

set -ex

sudo apt-get update

sudo apt-get install -y cmake

mkdir -p ~/ccache

pushd ~/ccache

rm -rf ccache

git clone https://github.com/ccache/ccache.git

mkdir -p ccache/build

pushd ccache/build

cmake -DCMAKE\_INSTALL\_PREFIX=${HOME}/ccache -DENABLE\_TESTING=OFF -DZSTD\_FROM\_INTERNET=ON ..

make -j$(nproc) install

popd

popd

mkdir -p ~/ccache/lib

mkdir -p ~/ccache/cuda

ln -s ~/ccache/bin/ccache ~/ccache/lib/cc

ln -s ~/ccache/bin/ccache ~/ccache/lib/c++

ln -s ~/ccache/bin/ccache ~/ccache/lib/gcc

ln -s ~/ccache/bin/ccache ~/ccache/lib/g++

ln -s ~/ccache/bin/ccache ~/ccache/cuda/nvcc

~/ccache/bin/ccache -M 25Gi

fi

export PATH=~/ccache/lib:$PATH

export CUDA\_NVCC\_EXECUTABLE=~/ccache/cuda/nvcc

```

Alternatively, `ccache` provided by newer Linux distributions (e.g. Debian/sid)

also works, but the `nvcc` symlink to `ccache` as described above is still required.

Note that the original `nvcc` binary (typically at `/usr/local/cuda/bin`) must

be on your `PATH`, otherwise `ccache` will emit the following error:

ccache: error: Could not find compiler "nvcc" in PATH

For example, here is how to install/configure `ccache` on Ubuntu:

```bash

# install ccache

sudo apt install ccache

# update symlinks and create/re-create nvcc link

sudo /usr/sbin/update-ccache-symlinks

sudo ln -s /usr/bin/ccache /usr/lib/ccache/nvcc

# config: cache dir is ~/.ccache, conf file ~/.ccache/ccache.conf

# max size of cache

ccache -M 25Gi # -M 0 for unlimited

# unlimited number of files

ccache -F 0

# deploy (and add to ~/.bashrc for later)

export PATH="/usr/lib/ccache:$PATH"

```

It is also possible to install `ccache` via `conda` by installing it from the

community-maintained `conda-forge` channel. Here is how to set up `ccache` this

way:

```bash

# install ccache

conda install -c conda-forge ccache

# set up ccache compiler symlinks

mkdir ~/ccache

mkdir ~/ccache/lib

mkdir ~/ccache/cuda

ln -s $CONDA\_PREFIX/bin/ccache ~/ccache/lib/cc

ln -s $CONDA\_PREFIX/bin/ccache ~/ccache/lib/c++

ln -s $CONDA\_PREFIX/bin/ccache ~/ccache/lib/gcc

ln -s $CONDA\_PREFIX/bin/ccache ~/ccache/lib/g++

ln -s $CONDA\_PREFIX/bin/ccache ~/ccache/cuda/nvcc

# update PATH to reflect symlink locations, consider

# adding this to your .bashrc

export PATH=~/ccache/lib:$PATH

export CUDA\_NVCC\_EXECUTABLE=~/ccache/cuda/nvcc

# increase ccache cache size to 25 GiB

ccache -M 25Gi

```

To check this is working, do two clean builds of pytorch in a row. The second

build should be substantially and noticeably faster than the first build.

#### Use a faster linker

If you are editing a single file and rebuilding in a tight loop, the time spent

linking will dominate. The system linker available in most Linux distributions

(GNU `ld`) is quite slow. Use a faster linker, like [lld](https://lld.llvm.org/).

The easiest way to use `lld` this is download the

[latest LLVM binaries](http://releases.llvm.org/download.html#8.0.0) and run:

```

ln -s /path/to/downloaded/ld.lld /usr/local/bin/ld

```

### C++ frontend development tips

We have very extensive tests in the [test/cpp/api](test/cpp/api) folder. The

tests are a great way to see how certain components are intended to be used.

When compiling PyTorch from source, the test runner binary will be written to

`build/bin/test\_api`. The tests use the [GoogleTest](https://github.com/google/googletest/blob/master/googletest)

framework, which you can read up about to learn how to configure the test runner. When

submitting a new feature, we care very much that you write appropriate tests.

Please follow the lead of the other tests to see how to write a new test case.

## CUDA development tips

If you are working on the CUDA code, here are some useful CUDA debugging tips:

1. `CUDA\_DEVICE\_DEBUG=1` will enable CUDA device function debug symbols (`-g -G`).

This will be particularly helpful in debugging device code. However, it will

slow down the build process for about 50% (compared to only `DEBUG=1`), so use wisely.

2. `cuda-gdb` and `cuda-memcheck` are your best CUDA debugging friends. Unlike`gdb`,

`cuda-gdb` can display actual values in a CUDA tensor (rather than all zeros).

3. CUDA supports a lot of C++11/14 features such as, `std::numeric\_limits`, `std::nextafter`,

`std::tuple` etc. in device code. Many of such features are possible because of the

[--expt-relaxed-constexpr](https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#constexpr-functions)

nvcc flag. There is a known [issue](https://github.com/ROCm-Developer-Tools/HIP/issues/374)

that ROCm errors out on device code, which uses such stl functions.

4. A good performance metric for a CUDA kernel is the

[Effective Memory Bandwidth](https://devblogs.nvidia.com/how-implement-performance-metrics-cuda-cc/).

It is useful for you to measure this metric whenever you are writing/optimizing a CUDA

kernel. Following script shows how we can measure the effective bandwidth of CUDA `uniform\_`

kernel.

```python

import torch

import time

size = 128\*512

nrep = 100

nbytes\_read\_write = 4 # this is number of bytes read + written by a kernel. Change this to fit your kernel.

for i in range(10):

a=torch.Tensor(size).cuda().uniform\_()

torch.cuda.synchronize()

start = time.time()

# dry run to alloc

out = a.uniform\_()

torch.cuda.synchronize()

start = time.time()

for i in range(nrep):

out = a.uniform\_()

torch.cuda.synchronize()

end = time.time()

timec = (end-start)/nrep

print("uniform, size, elements", size, "forward", timec, "bandwidth (GB/s)", size\*(nbytes\_read\_write)\*1e-9/timec)

size \*=2

```

## Windows development tips

For building from source on Windows, consult

[our documentation](https://pytorch.org/docs/stable/notes/windows.html) on it.

Occasionally, you will write a patch which works on Linux, but fails CI on Windows.

There are a few aspects in which MSVC (the Windows compiler toolchain we use) is stricter

than Linux, which are worth keeping in mind when fixing these problems.

1. Symbols are NOT exported by default on Windows; instead, you have to explicitly

mark a symbol as exported/imported in a header file with `\_\_declspec(dllexport)` /

`\_\_declspec(dllimport)`. We have codified this pattern into a set of macros

which follow the convention `\*\_API`, e.g., `CAFFE2\_API` inside Caffe2 and ATen.

(Every separate shared library needs a unique macro name, because symbol visibility

is on a per shared library basis. See c10/macros/Macros.h for more details.)

The upshot is if you see an "unresolved external" error in your Windows build, this

is probably because you forgot to mark a function with `\*\_API`. However, there is

one important counterexample to this principle: if you want a \*templated\* function

to be instantiated at the call site, do NOT mark it with `\*\_API` (if you do mark it,

you'll have to explicitly instantiate all of the specializations used by the call

sites.)

2. If you link against a library, this does not make its dependencies transitively

visible. You must explicitly specify a link dependency against every library whose

symbols you use. (This is different from Linux where in most environments,

transitive dependencies can be used to fulfill unresolved symbols.)

3. If you have a Windows box (we have a few on EC2 which you can request access to) and

you want to run the build, the easiest way is to just run `.jenkins/pytorch/win-build.sh`.

If you need to rebuild, run `REBUILD=1 .jenkins/pytorch/win-build.sh` (this will avoid

blowing away your Conda environment.)

Even if you don't know anything about MSVC, you can use cmake to build simple programs on

Windows; this can be helpful if you want to learn more about some peculiar linking behavior

by reproducing it on a small example. Here's a simple example cmake file that defines

two dynamic libraries, one linking with the other:

```CMake

project(myproject CXX)

set(CMAKE\_CXX\_STANDARD 14)

add\_library(foo SHARED foo.cpp)

add\_library(bar SHARED bar.cpp)

# NB: don't forget to \_\_declspec(dllexport) at least one symbol from foo,

# otherwise foo.lib will not be created.

target\_link\_libraries(bar PUBLIC foo)

```

You can build it with:

```bash

mkdir build

cd build

cmake ..

cmake --build .

```

### Known MSVC (and MSVC with NVCC) bugs

The PyTorch codebase sometimes likes to use exciting C++ features, and

these exciting features lead to exciting bugs in Windows compilers.

To add insult to injury, the error messages will often not tell you

which line of code actually induced the erroring template instantiation.

We've found the most effective way to debug these problems is to

carefully read over diffs, keeping in mind known bugs in MSVC/NVCC.

Here are a few well known pitfalls and workarounds:

\* This is not actually a bug per se, but in general, code generated by MSVC

is more sensitive to memory errors; you may have written some code

that does a use-after-free or stack overflows; on Linux the code

might work, but on Windows your program will crash. ASAN may not

catch all of these problems: stay vigilant to the possibility that

your crash is due to a real memory problem.

\* (NVCC) `c10::optional` does not work when used from device code. Don't use

it from kernels. Upstream issue: https://github.com/akrzemi1/Optional/issues/58

and our local issue #10329.

\* `constexpr` generally works less well on MSVC.

\* The idiom `static\_assert(f() == f())` to test if `f` is constexpr

does not work; you'll get "error C2131: expression did not evaluate

to a constant". Don't use these asserts on Windows.

(Example: `c10/util/intrusive\_ptr.h`)

\* (NVCC) Code you access inside a `static\_assert` will eagerly be

evaluated as if it were device code, and so you might get an error

that the code is "not accessible".

```cpp

class A {

static A singleton\_;

static constexpr inline A\* singleton() {

return &singleton\_;

}

};

static\_assert(std::is\_same(A\*, decltype(A::singleton()))::value, "hmm");

```

\* The compiler will run out of heap space if you attempt to compile files that

are too large. Splitting such files into separate files helps.

(Example: `THTensorMath`, `THTensorMoreMath`, `THTensorEvenMoreMath`.)

\* MSVC's preprocessor (but not the standard compiler) has a bug

where it incorrectly tokenizes raw string literals, ending when it sees a `"`.

This causes preprocessor tokens inside the literal like an`#endif` to be incorrectly

treated as preprocessor directives. See https://godbolt.org/z/eVTIJq as an example.

\* Either MSVC or the Windows headers have a PURE macro defined and will replace

any occurrences of the PURE token in code with an empty string. This is why

we have AliasAnalysisKind::PURE\_FUNCTION and not AliasAnalysisKind::PURE.

The same is likely true for other identifiers that we just didn't try to use yet.

## Running clang-tidy

[Clang-Tidy](https://clang.llvm.org/extra/clang-tidy/index.html) is a C++

linter and static analysis tool based on the clang compiler. We run clang-tidy

in our CI to make sure that new C++ code is safe, sane and efficient. See our

[.travis.yml](https://github.com/pytorch/pytorch/blob/master/.travis.yml) file

for the simple commands we use for this.

To run clang-tidy locally, follow these steps:

1. Install clang-tidy. First, check if you already have clang-tidy by simply

writing `clang-tidy` in your terminal. If you don't yet have clang-tidy, you

should be able to install it easily with your package manager, e.g. by writing

`apt-get install clang-tidy` on Ubuntu. See https://apt.llvm.org for details on

how to install the latest version. Note that newer versions of clang-tidy will

have more checks than older versions. In our CI, we run clang-tidy-6.0.

2. Use our driver script to run clang-tidy over any changes relative to some

git revision (you may want to replace `HEAD~1` with `HEAD` to pick up

uncommitted changes). Changes are picked up based on a `git diff` with the

given revision:

```bash

python tools/clang\_tidy.py -d build -p torch/csrc --diff 'HEAD~1'

```

Above, it is assumed you are in the PyTorch root folder. `path/to/build` should

be the path to where you built PyTorch from source, e.g. `build` in the PyTorch

root folder if you used `setup.py build`. You can use `-c <clang-tidy-binary>`

to change the clang-tidy this script uses. Make sure you have PyYaml installed,

which is in PyTorch's `requirements.txt`.

## Pre-commit tidy/linting hook

We use clang-tidy and flake8 (installed with flake8-bugbear,

flake8-comprehensions, flake8-mypy, and flake8-pyi) to perform additional

formatting and semantic checking of code. We provide a pre-commit git hook for

performing these checks, before a commit is created:

```bash

ln -s ../../tools/git-pre-commit .git/hooks/pre-commit

```

You'll need to install an appropriately configured flake8; see

[Lint as you type](https://github.com/pytorch/pytorch/wiki/Lint-as-you-type)

for documentation on how to do this.

## Building PyTorch with ASAN

[ASAN](https://github.com/google/sanitizers/wiki/AddressSanitizer) is very

useful for debugging memory errors in C++. We run it in CI, but here's how to

get the same thing to run on your local machine.

First, install LLVM 8. The easiest way is to get [prebuilt

binaries](http://releases.llvm.org/download.html#8.0.0) and extract them to

folder (later called `$LLVM\_ROOT`).

Then set up the appropriate scripts. You can put this in your `.bashrc`:

```

LLVM\_ROOT=<wherever your llvm install is>

PYTORCH\_ROOT=<wherever your pytorch checkout is>

LIBASAN\_RT="$LLVM\_ROOT/lib/clang/8.0.0/lib/linux/libclang\_rt.asan-x86\_64.so"

build\_with\_asan()

{

LD\_PRELOAD=${LIBASAN\_RT} \

CC="$LLVM\_ROOT/bin/clang" \

CXX="$LLVM\_ROOT/bin/clang++" \

LDSHARED="clang --shared" \

LDFLAGS="-stdlib=libstdc++" \

CFLAGS="-fsanitize=address -fno-sanitize-recover=all -shared-libasan -pthread" \

CXX\_FLAGS="-pthread" \

USE\_CUDA=0 USE\_OPENMP=0 BUILD\_CAFFE2\_OPS=0 USE\_DISTRIBUTED=0 DEBUG=1 \

python setup.py develop

}

run\_with\_asan()

{

LD\_PRELOAD=${LIBASAN\_RT} $@

}

# you can look at build-asan.sh to find the latest options the CI uses

export ASAN\_OPTIONS=detect\_leaks=0:symbolize=1:strict\_init\_order=true

export UBSAN\_OPTIONS=print\_stacktrace=1:suppressions=$PYTORCH\_ROOT/ubsan.supp

export ASAN\_SYMBOLIZER\_PATH=$LLVM\_ROOT/bin/llvm-symbolizer

```

Then you can use the scripts like:

```

suo-devfair ~/pytorch ? build\_with\_asan

suo-devfair ~/pytorch ? run\_with\_asan python test/test\_jit.py

```

### Getting `ccache` to work

The scripts above specify the `clang` and `clang++` binaries directly, which

bypasses `ccache`. Here's how to get `ccache` to work:

1. Make sure the ccache symlinks for `clang` and `clang++` are set up (see

CONTRIBUTING.md)

2. Make sure `$LLVM\_ROOT/bin` is available on your `$PATH`.

3. Change the `CC` and `CXX` variables in `build\_with\_asan()` to point

directly to `clang` and `clang++`.

### Why this stuff with `LD\_PRELOAD` and `LIBASAN\_RT`?

The standard workflow for ASAN assumes you have a standalone binary:

1. Recompile your binary with `-fsanitize=address`.

2. Run the binary, and ASAN will report whatever errors it find.

Unfortunately, PyTorch is a distributed as a shared library that is loaded by

a third-party executable (Python). Its too much of a hassle to recompile all

of Python every time we want to use ASAN. Luckily, the ASAN folks have a

workaround for cases like this:

1. Recompile your library with `-fsanitize=address -shared-libasan`. The

extra `-shared-libasan` tells the compiler to ask for the shared ASAN

runtime library.

2. Use `LD\_PRELOAD` to tell the dynamic linker to load the ASAN runtime

library before anything else.

More information can be found

[here](https://github.com/google/sanitizers/wiki/AddressSanitizerAsDso).

### Why LD\_PRELOAD in the build function?

We need `LD\_PRELOAD` because there is a cmake check that ensures that a

simple program builds and runs. If we are building with ASAN as a shared

library, we need to `LD\_PRELOAD` the runtime library, otherwise there will

dynamic linker errors and the check will fail.

We dont actually need either of these if we fix the cmake checks.

### Why no leak detection?

Python leaks a lot of memory. Possibly we could configure a suppression file,

but we havent gotten around to it.

## Caffe2 notes

In 2018, we merged Caffe2 into the PyTorch source repository. While the

steady state aspiration is that Caffe2 and PyTorch share code freely,

in the meantime there will be some separation.

If you submit a PR to only PyTorch or only Caffe2 code, CI will only

run for the project you edited. The logic for this is implemented

in `.jenkins/pytorch/dirty.sh` and `.jenkins/caffe2/dirty.sh`; you

can look at this to see what path prefixes constitute changes.

This also means if you ADD a new top-level path, or you start

sharing code between projects, you need to modify these files.

There are a few "unusual" directories which, for historical reasons,

are Caffe2/PyTorch specific. Here they are:

- `CMakeLists.txt`, `Makefile`, `binaries`, `cmake`, `conda`, `modules`,

`scripts` are Caffe2-specific. Don't put PyTorch code in them without

extra coordination.

- `mypy\*`, `requirements.txt`, `setup.py`, `test`, `tools` are

PyTorch-specific. Don't put Caffe2 code in them without extra

coordination.

## CI failure tips

Once you submit a PR or push a new commit to a branch that is in

an active PR, CI jobs will be run automatically. Some of these may

fail and you will need to find out why, by looking at the logs.

Fairly often, a CI failure might be unrelated to your changes. In this

case, you can usually ignore the failure.

Some failures might be related to specific hardware or environment

configurations. In this case, if the job is run by CircleCI, you can

ssh into the job's session to perform manual debugging using the

following steps:

1. In the CircleCI page for the failed job, make sure you are logged in

and then click the `Rerun` actions dropdown button on the top right.

Click `Rerun Job with SSH`.

2. When the job reruns, a new step will be added in the `STEPS` tab

labelled `Set up SSH`. Inside that tab will be an ssh command that

you can execute in a shell.

3. Once you are connected through ssh, you may need to enter a docker

container. Run `docker ps` to check if there are any docker

containers running. Note that your CI job might be in the process

of initiating a docker container, which means it will not show up

yet. It is best to wait until the CI job reaches a step where it is

building pytorch or running pytorch tests. If the job does have a

docker container, run `docker exec -it IMAGE\_ID /bin/bash` to

connect to it.

4. Now you can find the pytorch working directory, which could be

`~/workspace` or `~/project`, and run commands locally to debug

the failure.