TECHNISCHE UNIVERSITÄT DRESDEN

DRESDEN concept

Alexander Grund
Center for Information Services and High Performance Computing (ZIH)

EasyBuilding PyTorch

9th EasyBuild User Meeting, Umeå
24 April 2024

Timeline of PyTorch in EasyBuild

June	**2018:**	PyTorch **0.3.1**	→ EasyBuild **3.6.2**
August	**2018:**	PyTorch **0.4.1**	→ EasyBuild **3.7.0**
Feb	**2019:**	PyTorch **1.0.1**	→ EasyBuild **3.9.1**
July	**2019:**	PyTorch **1.1.0**	→ EasyBuild **3.9.4**
August	**2019:**	PyTorch **1.2.0**	→ EasyBuild **4.0.0**
Nov	**2019:**	PyTorch **1.3.1**	→ EasyBuild **4.1.0**
Feb	**2020:**	PyTorch **1.4.0**	→ EasyBuild **4.2.0**
Sep	**2020:**	PyTorch **1.6.0**	→ EasyBuild **4.3.1**
January	**2021:**	PyTorch **1.7.1**	→ EasyBuild **4.3.3**
March	**2021:**	PyTorch **1.8.1**	→ EasyBuild **4.4.0**
June	**2021:**	PyTorch **1.9.0**	→ EasyBuild **4.4.1**
Nov	**2021:**	PyTorch **1.10.0**	→ EasyBuild **4.5.1**
March	**2022:**	PyTorch **1.11.0**	→ EasyBuild **4.6.0**
Oct	**2022:**	PyTorch **1.12.0**	→ EasyBuild **4.6.2**
Oct	**2022:**	PyTorch **1.12.1**	→ EasyBuild **4.7.0**
Jan	**2023:**	PyTorch **1.13.1**	→ EasyBuild **4.8.0**
July	**2023:**	+ CUDA **1.13.1**	→ EasyBuild **4.8.2**
Oct	**2023:**	PyTorch **2.0.1**	→ EasyBuild **4.8.2**
Dec	**2023:**	PyTorch **2.1.2**	→ EasyBuild **4.9.0**

Testing PyTorch

Extensive test suite

— Python UnitTest module

— Shell/Python script starting each test file

— Few failures with PyTorch 0.x

Testing PyTorch

PyTorch 1.x: Major changes & extended testing by different sites

➔ Compiler versions

➔ CPU architectures

➔ GPUs architecture & count

Testing PyTorch

Dependency conflicts

➔ Hardcoded version ranges

➔ Changed interfaces in NumPy, Python, CUDA, NCCL & Co.

➔ Changed semantics

(Meta-)Testing PyTorch

Fixes

“Obvious” issues → Look carefully at failure & code

- Unexpected C++ preprocessor defines
- Hard coded CPU affinity
- Wrong expected value

- Optional dependencies assumed in tests

- CUDA calls in CPU builds
- CUPTI calls on CUDA 10.1

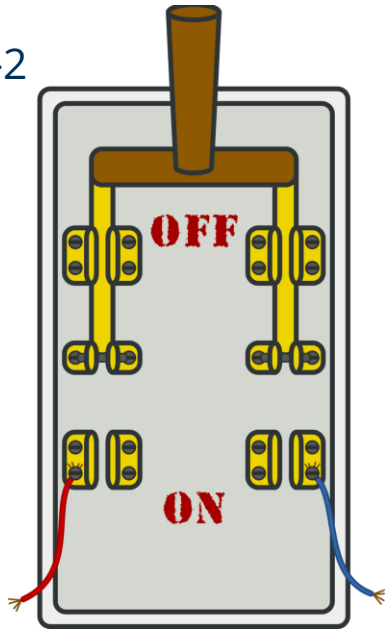
- Number of GPUs for test

(Meta-)Testing PyTorch

Fixes

– Apply upstream patches

– Derandomize inputs (e.g. singular matrices)
– Fixed input data version downloaded during test
– Relax tolerances
  Fix for A100 GPUs: 1e-4 → 1e-2
  Disable TF32



Disable “strange” tests

Reduce testing PyTorch

Disable tests

— Known bug / disabled in PyTorch CI
  – Also fails with PIP package

— Doesn't fail when run separately

— Fail randomly due to randomized input
  ➔Consistent output for same inputs?

— Random timeouts
  ➔Patch?

— Assumptions incompatible with EasyBuild
  - Specific CUDA/GPU version
  - Network access
  - Filesystem (/dev/shm, $HOME)

Reduce testing PyTorch

The Challenge

1. Output (in EB logfile) **VERY** long

2. Random failures

➔ Need to allow some failures

3. Test file may contain many (100+) tests

➔ Skipping full file might miss important issues
E.g. CUDA with PyBind11

Reduce testing PyTorch

Reduce the amount of skipped tests

1. Test files/suites → **test_cuda**(.py)
2. Test functions → def **test_memory_snapshot**(self)
3. Test cases → @**dtypes**(torch.**float**, torch.**double**)

Reduce testing PyTorch

The EasyBlock

Parses test output
- Count number of executed and failed tests
- Allow (configurable) number of tests to fail
- Assemble readable summary / error

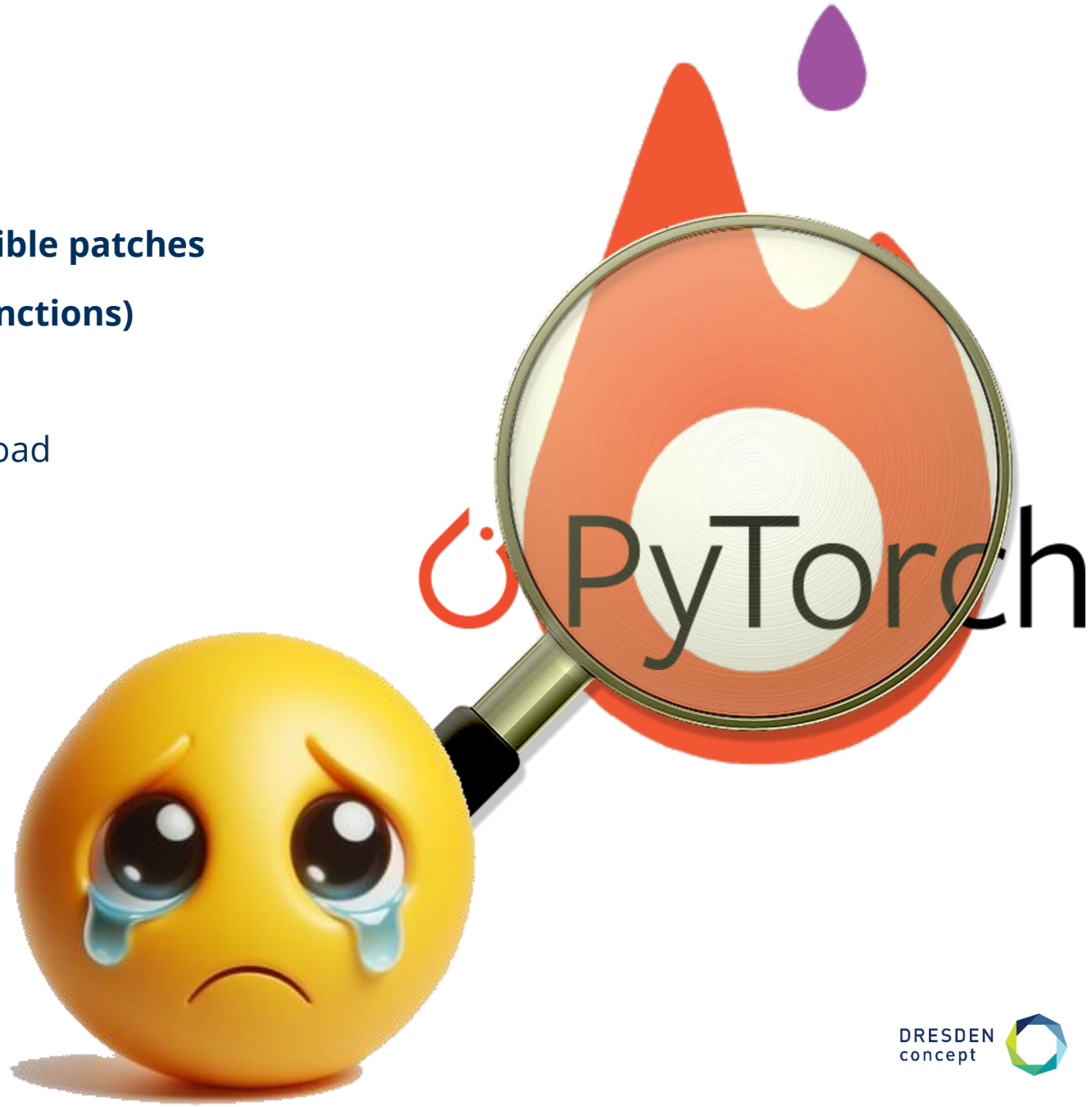
‼ **Non-trivial parsing (Regular Expressions)**

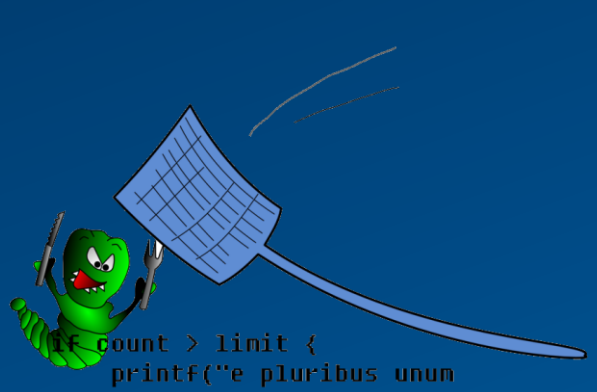
‼ **Output changes between PyTorch version**

➔ Fail when tests could not be counted

Improving PyTorch

1. **PyTorch GitHub**
   Open issue if not found → Reference in possible patches
2. **Changes in main branch (test and related functions)**
   - GitHub GUI: “blame” → commit(s)
   - Append “.patch” to commit/PR URL to download
3. **Different toolchains / dependency versions**
4. **Fix it & create pull request**
5. **Give up**





Debugging PyTorch

Examples inside

Bugs in vectorized code

Symptoms
— Many similar tests fail
— Last few values are correct
  – Tensor size not multiple of 4/8/16

> tensor**([1e+20]*10**, dtype=**float32**).**sin**()
tensor([-inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf**, -0.65, -0.65**])

test_unary_ufuncs
- test_reference_numerics_large_**cos**_**cpu**_bfloat16
- test_reference_numerics_large_**cos**_**cpu**_float32
- test_reference_numerics_large_**sin**_**cpu**_bfloat16
- test_reference_numerics_large_**sin**_**cpu**_float32

test_binary_ufuncs
- test_contig_vs_every_other_bitwise_right_**shift**_**cpu**_int16
- test_contig_vs_every_other_bitwise_right_**shift**_**cpu**_int32
- test_contig_vs_every_other_bitwise_right_**shift**_**cpu**_int64
- test_non_contig_bitwise_right_**shift**_**cpu**_int16
- test_non_contig_bitwise_right_**shift**_**cpu**_int32
- test_non_contig_bitwise_right_**shift**_**cpu**_int64
- test_**shift**_limits_**cpu**_int16
- test_**shift**_limits_**cpu**_int32
- test_**shift**_limits_**cpu**_int64

Bugs in vectorized code

```
// emulates vectorized types
template <class T>
struct Vec256 {
__at_align32__ T values[32 / sizeof(T)];
...
Vec256<float> sin()
Vec256<float> operator>>(...)
```

➔ Map to hardware instructions

➔ Fallback to auto-vectorization

test_unary_ufuncs
- test_reference_numerics_large_**cos**_**cpu**_bfloat16
- test_reference_numerics_large_**cos**_**cpu**_float32
- test_reference_numerics_large_**sin**_**cpu**_bfloat16
- test_reference_numerics_large_**sin**_**cpu**_float32

test_binary_ufuncs
- test_contig_vs_every_other_bitwise_right_**shift**_**cpu**_int16
- test_contig_vs_every_other_bitwise_right_**shift**_**cpu**_int32
- test_contig_vs_every_other_bitwise_right_**shift**_**cpu**_int64
- test_non_contig_bitwise_right_**shift**_**cpu**_int16
- test_non_contig_bitwise_right_**shift**_**cpu**_int32
- test_non_contig_bitwise_right_**shift**_**cpu**_int64
- test_**shift**_limits_**cpu**_int16
- test_**shift**_limits_**cpu**_int32
- test_**shift**_limits_**cpu**_int64

Bugs in vectorized code – Level 1

Vec256<float> sin() const {
auto x = abs(); // take the absolute value and xtract sign
auto sign_bit = (*this) & sign_mask;
auto y = x * _4div_pi; // scale by 4/Pi

```
// store the integer part of y in mm0

// j=(j+1) & (~1) (see the cephes sources)
vint32 imm0 = (vec_signed(y._vec0) + vi_1) & vi_inv1;
vint32 imm1 = (vec_signed(y._vec1) + vi_1) & vi_inv1;
y._vec0 = vec_float(imm0);
y._vec1 = vec_float(imm1);
// get the swap sign flag
Vectorized<float> swap_sign_bit, poly_mask;
```

swap_sign_bit.**_vecb0** = (vbool32)**vec_sl**(imm0 & vi_4, vu_29);

```
swap_sign_bit._vecb1 = (vbool32)vec_sl(imm1 & vi_4, vu_29);
// get the polynom selection mask
// there is one polynom for 0 <= x <= Pi/4
// and another one for Pi/4<x<=Pi/2
// Both branches will be computed.

poly_mask._vecb0 = vec_cmpeq((imm0 & vi_2), vi_0);
poly_mask._vecb1 = vec_cmpeq((imm1 & vi_2), vi_0);
sign_bit = sign_bit ^ swap_sign_bit; // xor operation

// The magic pass: "Extended precision modular arithmetic"
//  x = ((x - y * DP1) - y * DP2) - y * DP3;
x = y.madd(minus_cephes_dp1, x);
x = y.madd(minus_cephes_dp2, x);
x = y.madd(minus_cephes_dp3, x);

// Evaluate the first polynom  (0 <= x <= Pi/4)
auto z = x * x;
y = z.madd(coscof_p0, coscof_p1);
y = y.madd(z, coscof_p2);
y = y * z * z;
y = y - z * half + one;

// Evaluate the second polynom  (Pi/4 <= x <= 0)
auto y2 = z.madd(sincof_p0, sincof_p1);
y2 = y2.madd(z, sincof_p2);
y2 = y2 * z;
y2 = y2.madd(x, x);
// select the correct result from the two polynoms
y = blendv(y, y2, poly_mask);
y = y ^ sign_bit;
```

return y;

Vec256<float> sin() const {
return {**Sleef_sinf4_u10vsx**(_vec0),
Sleef_sinf4_u10vsx(_vec1)};

Bugs in vectorized code – Level 2

```
> torch.ones(16) | torch.ones(16)
> torch.ones(16) & torch.ones(16)

tensor(18939904,   8192, 0, 0, 0, 0, 0, 0,
       18939904,   8192, 0, 0, 0, 0, 0, 0)
```

```
Vec256 binary_op(Vec256 a, Vec256 b, Op op) {
  a_ptr = reinterpret_cast<intmax_t*>((T*) a);
  b_ptr = reinterpret_cast<intmax_t*>((T*) b);
  for (i = 0...n)
      result[i] = op(a_ptr[i], b_ptr[i]);
```

Bugs in vectorized code – Level 2


```
> torch.ones(16) | torch.ones(16)
> torch.ones(16) & torch.ones(16)

tensor(18939904,   8192, 0, 0, 0, 0, 0, 0,
       18939904,   8192, 0, 0, 0, 0, 0, 0)
```

```
Vec256 binary_op(Vec256 a, Vec256 b, Op op) {
  a_ptr = reinterpret_cast<intmax_t*>((T*) a);
  b_ptr = reinterpret_cast<intmax_t*>((T*) b);
  for (i = 0...n)
      result[i] = op(a_ptr[i], b_ptr[i]);
```

```
Vec256 binary_op(Vec256 a, Vec256 b, Op op) {
  a_ptr = load_as<intmax_t*>(a);
  b_ptr = load_as<intmax_t*>(b);
  for (i = 0...n)
      result[i] = op(a_ptr[i], b_ptr[i]);
```

Bugs in vectorized code – Level 3

– Same input values
– Different memory layout

➔

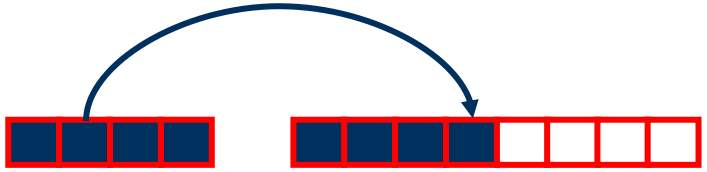
tensor([nan, **nan**, nan, 13.8, 13.8, nan, **13.8, 13.8**])
tensor([nan, **13,8**, nan, 13.8, 13.8, nan, **nan, nan**])

– Disappears on **–O0**

➔ **Compiler Bug**
– Fixed in GCC 9.4 / 10.1 after report

Level 4 Bugs

1. **Invalid memory access**
   - Use after free
   - Uninitialized loads



2. **Exception handling**
   - Static CUPTI → Crash
   - Double free → Compiler bug

Useful tools

1. **EasyBuild**
   – Logs & --dump-env-script ➔ Manual build & test
2. **GDB**
3. **Valgrind**
4. **Git**
   – Blame
   – Bisect
5. **Print**