PyTorch Internals

Edward Z. Yang

Who's this talk for?

You want to contribute to PyTorch,
but the codebase seems daunting

Notable omissions: JIT, distributed

Concepts

Tensor/Storage/Strides

Layout/Device/Dtype

Autograd

Mechanics

Operator call stack

Tools for writing kernels

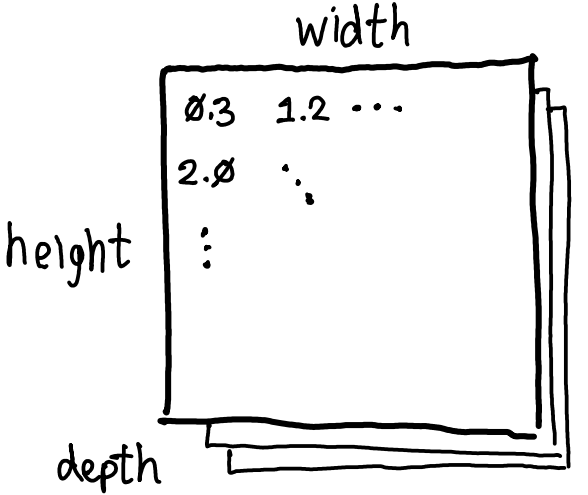
Legacy code

Efficient workflow

Concepts

Tensor

Tensor



sizes	(D, H, W)
strides	(H*W, W, 1)
dtype	float
device	cuda:0
layout	strided

contiguous

Tensor: Strided Representation

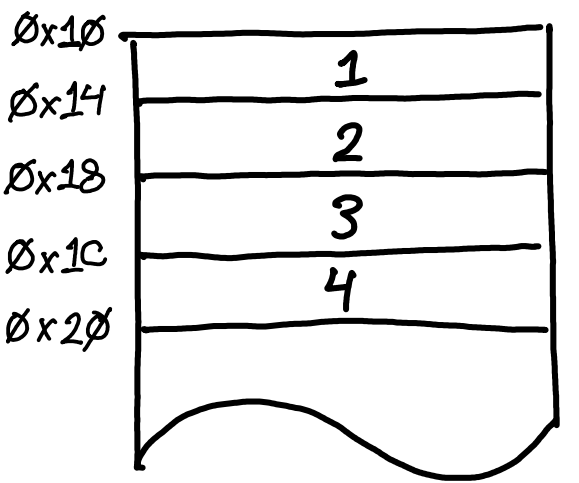
logical

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

dtype=torch.int32

maps to

physical



sizes [2, 2]
strides [2, 1]

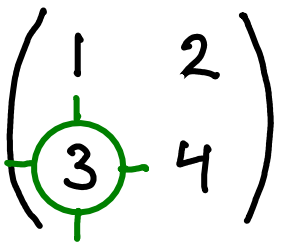
Tensor: Strided Representation

index stride index stride

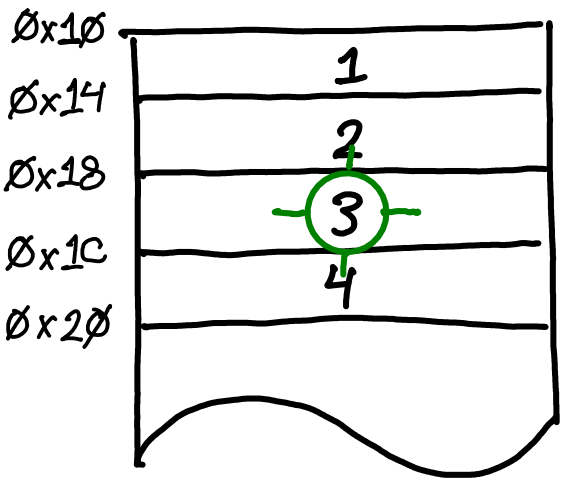
logical

$1 \times 2 + 2 \times 0 = 2$

physical



tensor[1,0]



sizes [2,2]

strides [2,1]

Upcoming: TensorAccessor

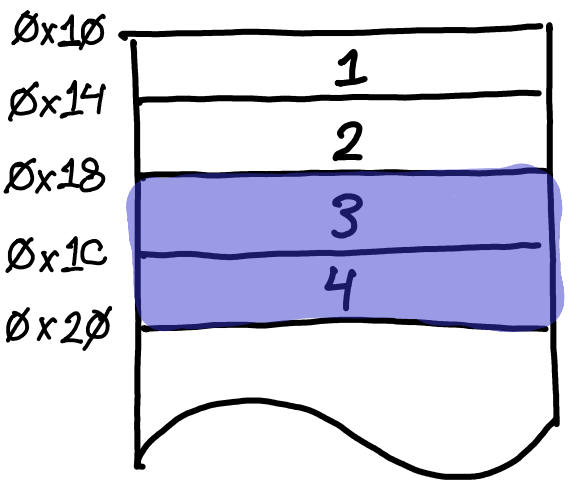
Tensor: Strided Representation

logical

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

tensor[1,:]

physical



sizes [2]
strides [1]
offset 2

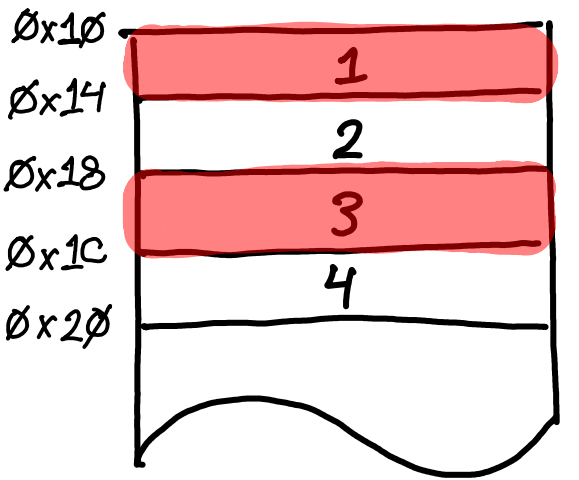
Tensor: Strided Representation

logical

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

tensor[:, 0]

physical

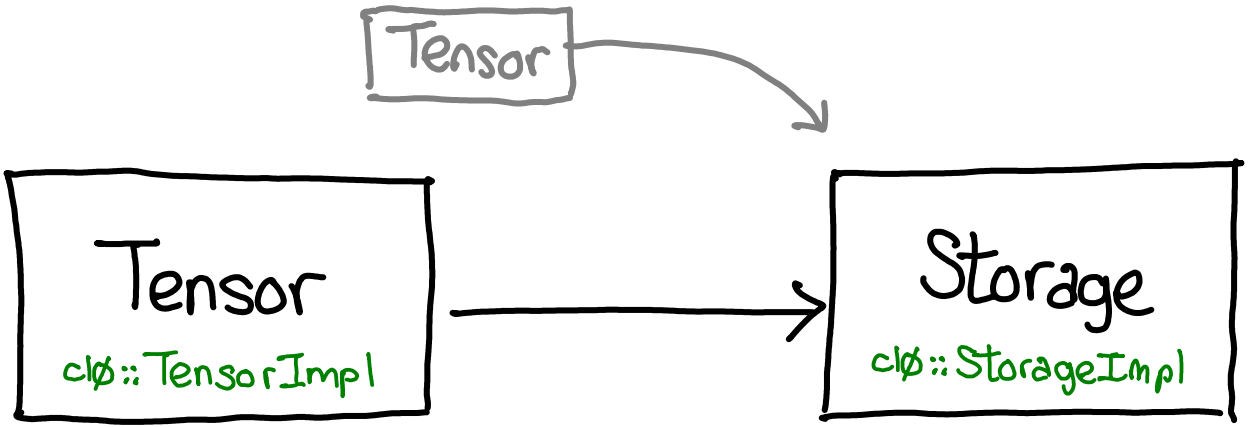


sizes [2]
strides [2]

stride ≠ 1 means we skip elements

https://ezyang.github.io/stride-visualizer

Tensor: Strided Representation



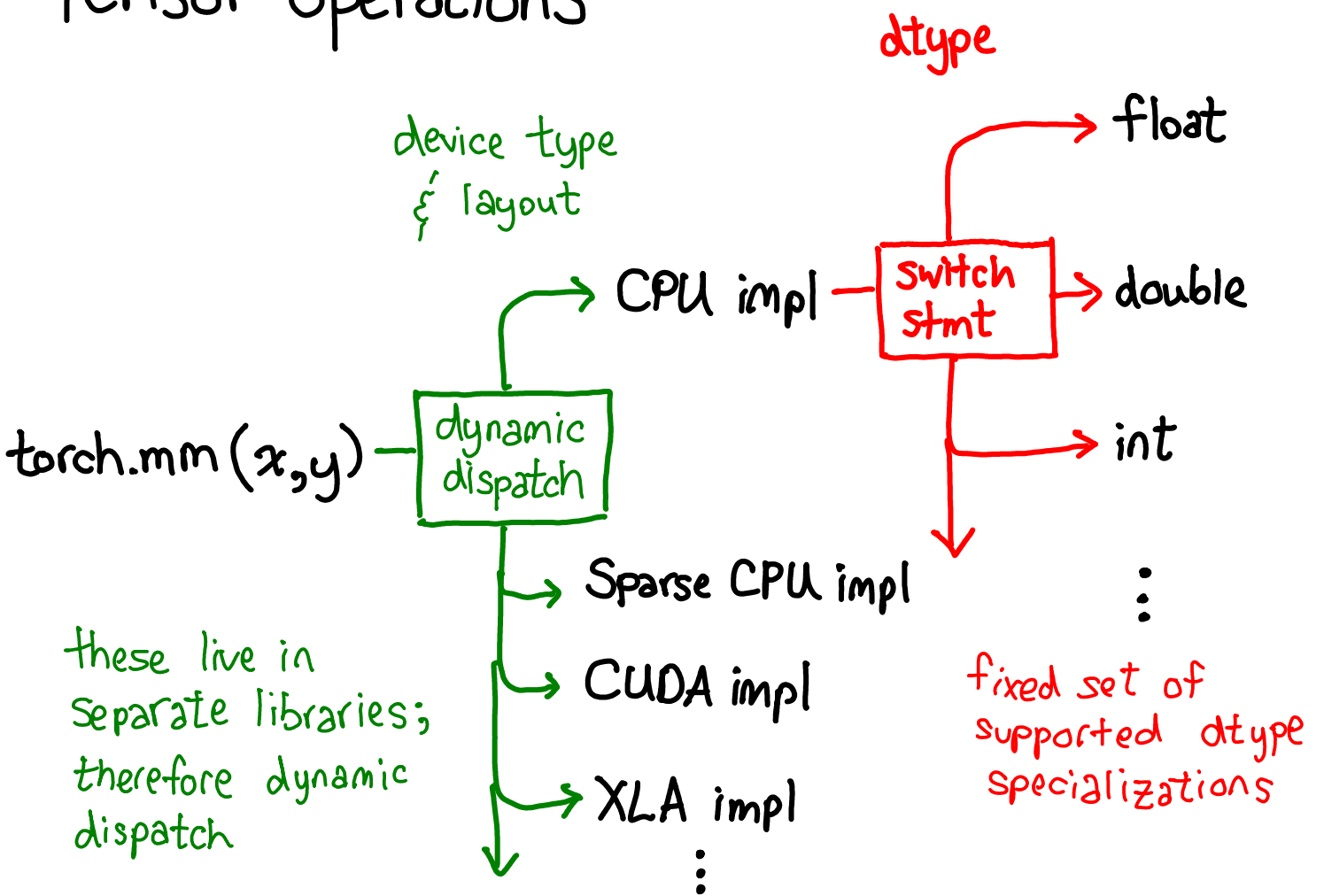
the logical view
sizes, strides, offset

the actual physical data
physical size, dtype

there is always a Tensor + Storage, even for "simple" cases

Tensor operations

(slight simplification)



Tensor extensions

quantized tensors

sparse tensors

encrypted tensors

There's more to life than strided tensors

MKLDNN tensors

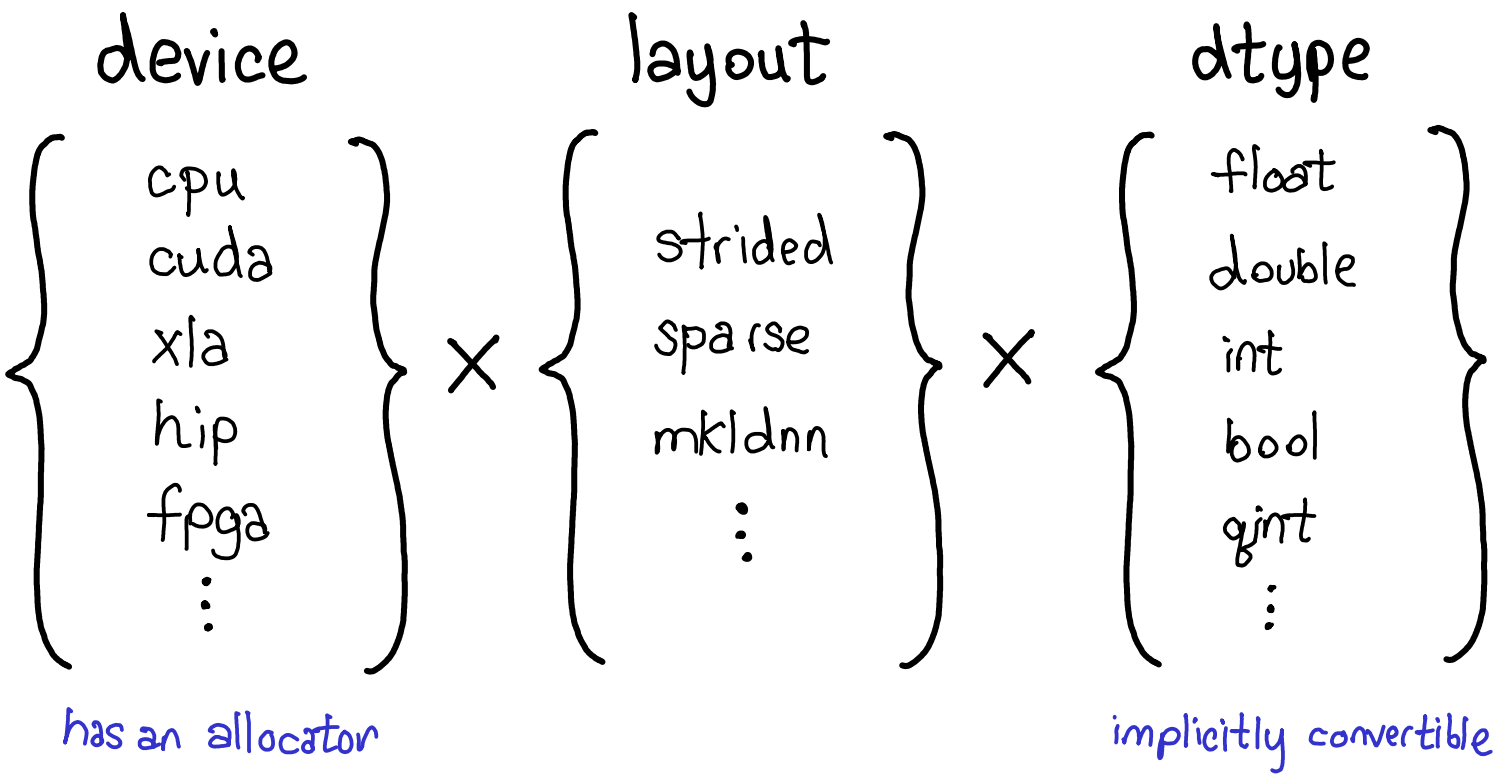
TPU tensors

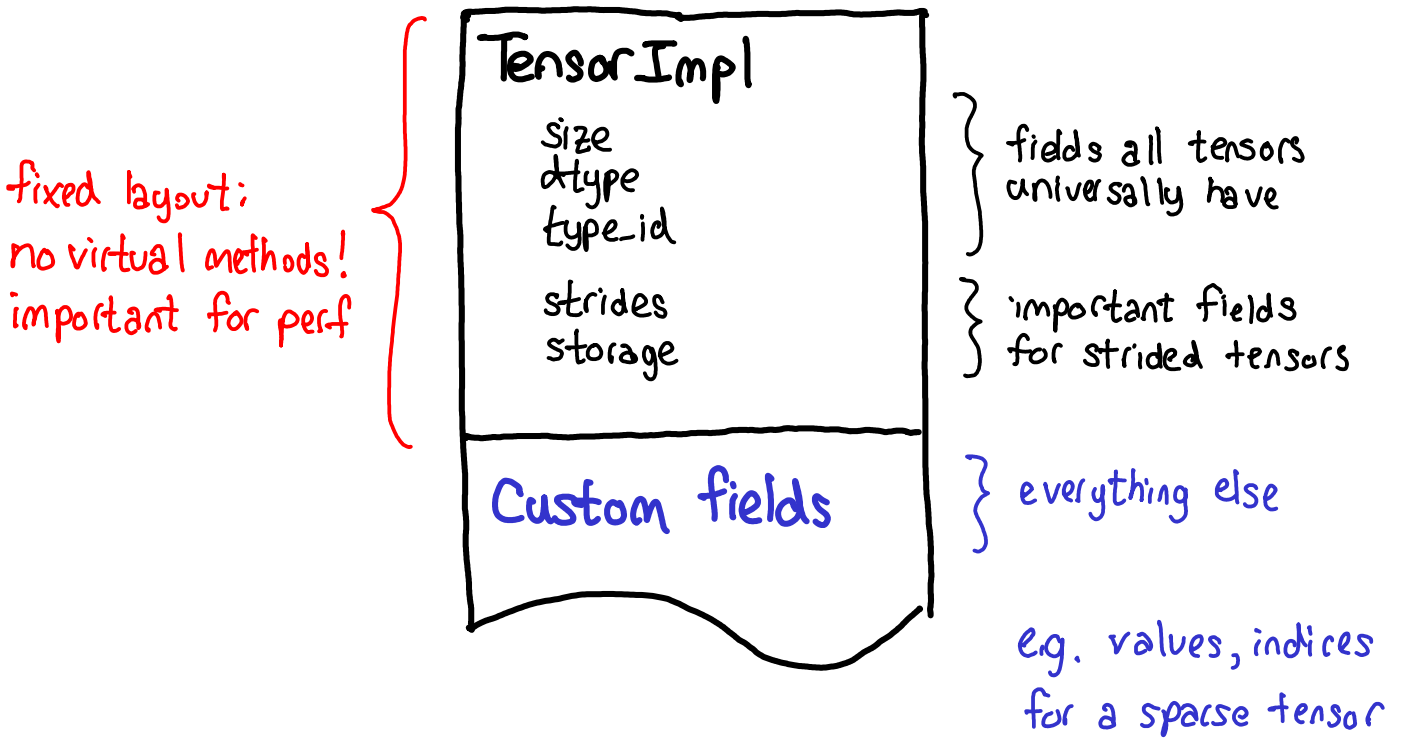
batch tensors

tensor wrapper

e.g., batch tensors

(things that don't flow through autograd)




fixed layout:
no virtual methods!
important for perf
TensorImpl
size
dtype
type_id
strides
storage
Custom fields
fields all tensors universally have
important fields for strided tensors
everything else
e.g. values, indices for a sparse tensor

Autograd

```
i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
next_h = next_h.tanh()
loss = next_h.sum()
loss.backward()
```

W_h, W_x, x, prev_h

requires grad

```
i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
next_h2 = next_h.tanh()
loss = next_h2.sum()
```

W_h, W_x, x, prev_h

requires grad

swap inputs and outputs!

```
grad_loss = torch.tensor(1, dtype=loss.dtype())
grad_next_h2 += grad_loss.expand(next_h2.size())
grad_next_h += tanh_backward(grad_next_h2, next_h2)
grad_i2h, grad_h2h += grad_next_h, grad_next_h
W_h.grad += mm_mat1_backward(grad_h2h, prev_h.t(), W_h, 1)
W_x.grad += mm_mat1_backward(grad_i2h, x.t(), W_x, 1)
```

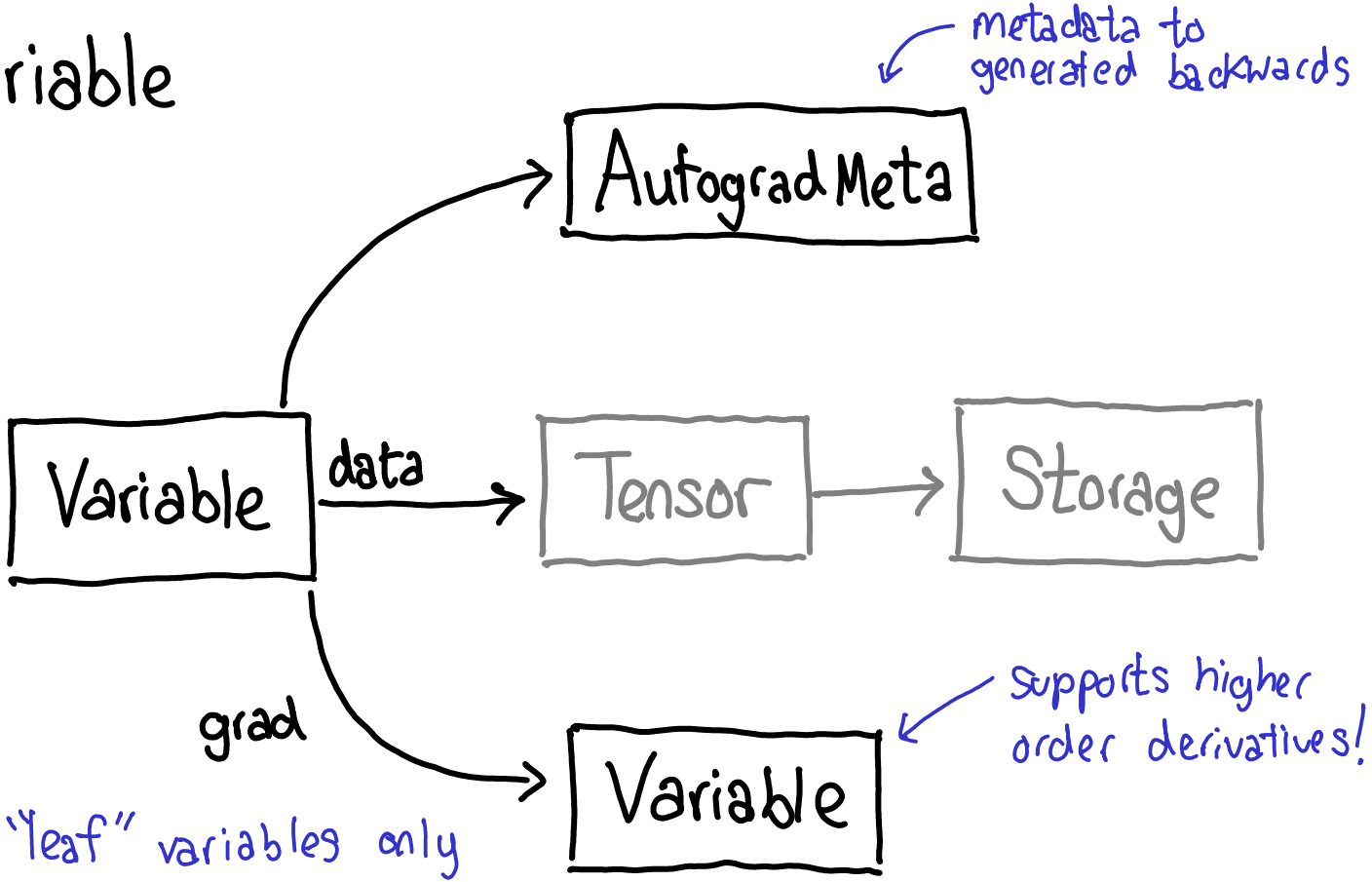
original inputs may be reused

how do we get here?

All grad variables are zero to start:

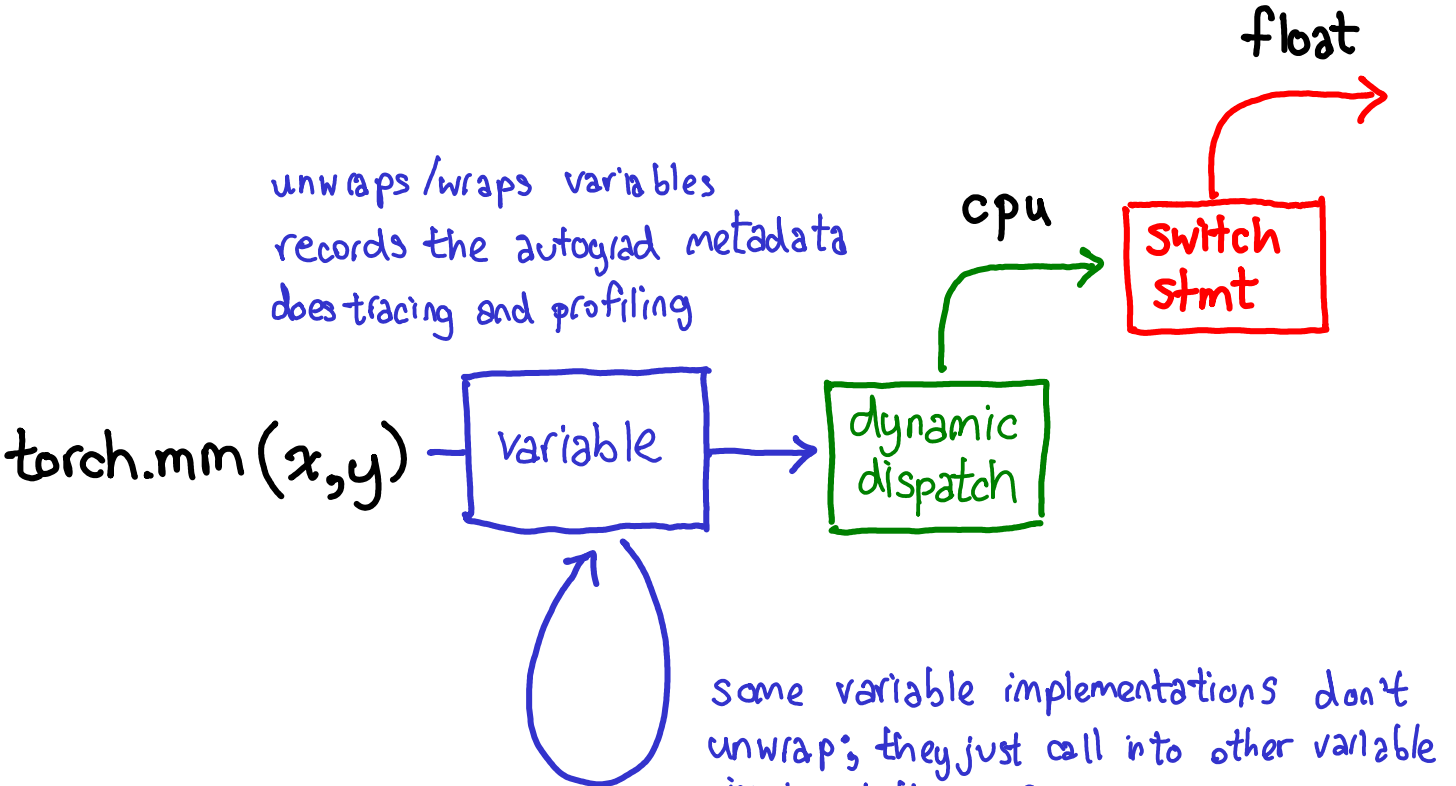
```
grad_i2h = torch.zeros_like(i2h)
```

Variable



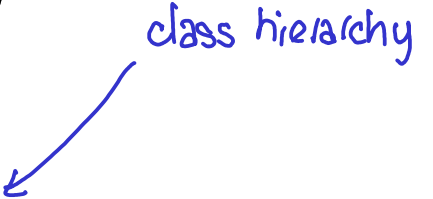
NB: This changing soon!

Dispatch with autograd



AutogradMeta

```
Variable grad_;
shared_ptr<Function> grad_fn_;
weak_ptr<Function> grad_accumulator_;
bool requires_grad_;  // leaf only
uint32_t output_nr_;
```



Function

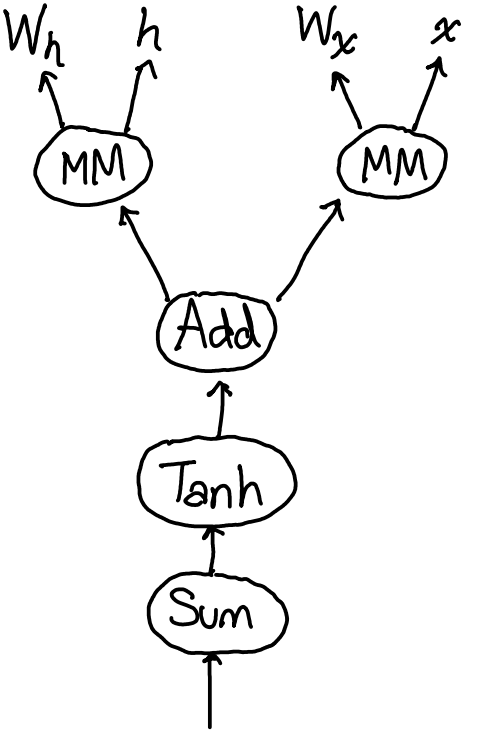
```
edge_list next_edges_;
```

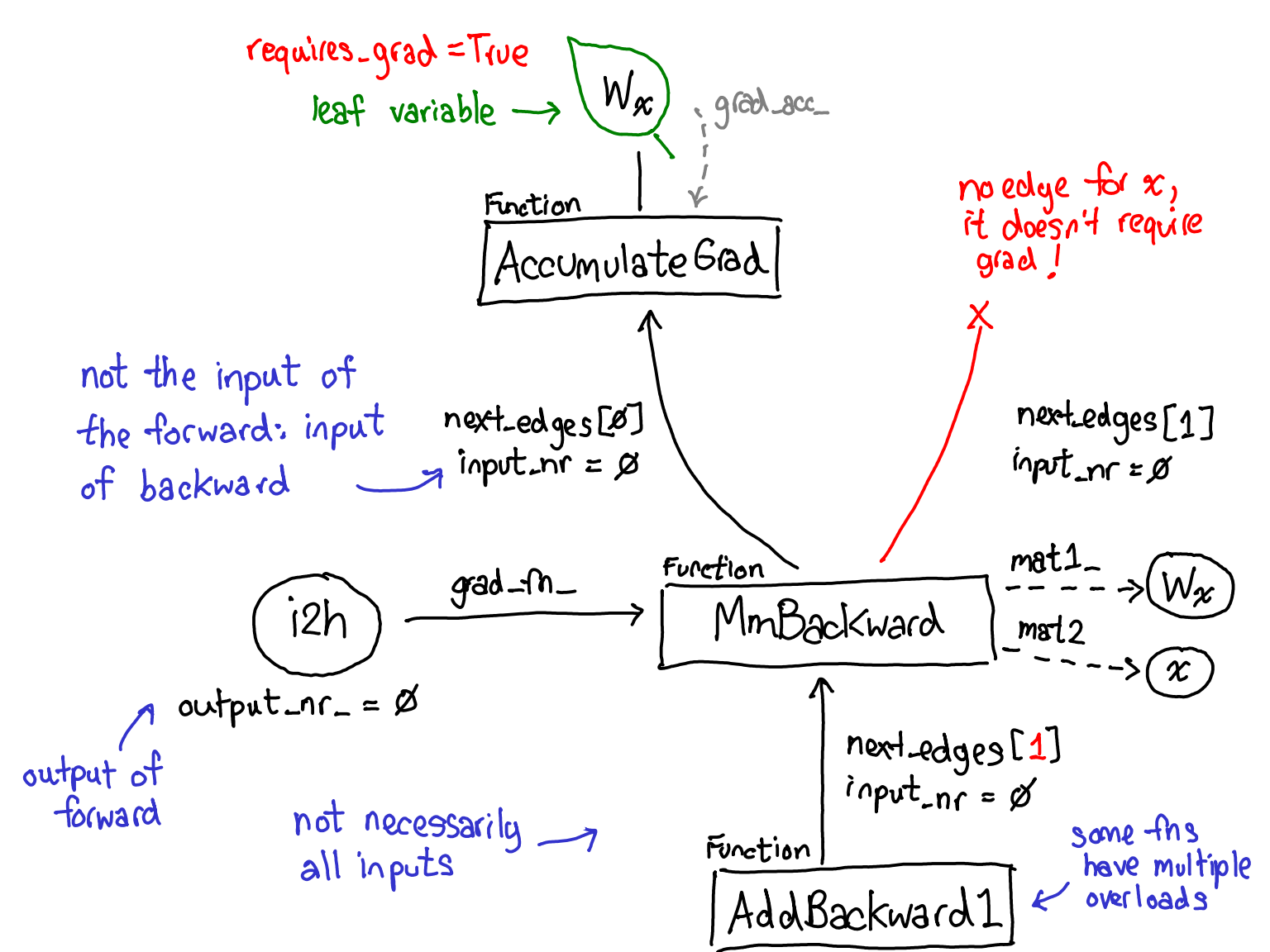
(e.g.,) `SavedVariable x;`

Edge

```
shared_ptr<Function> function;
uint32_t input_nr;
```

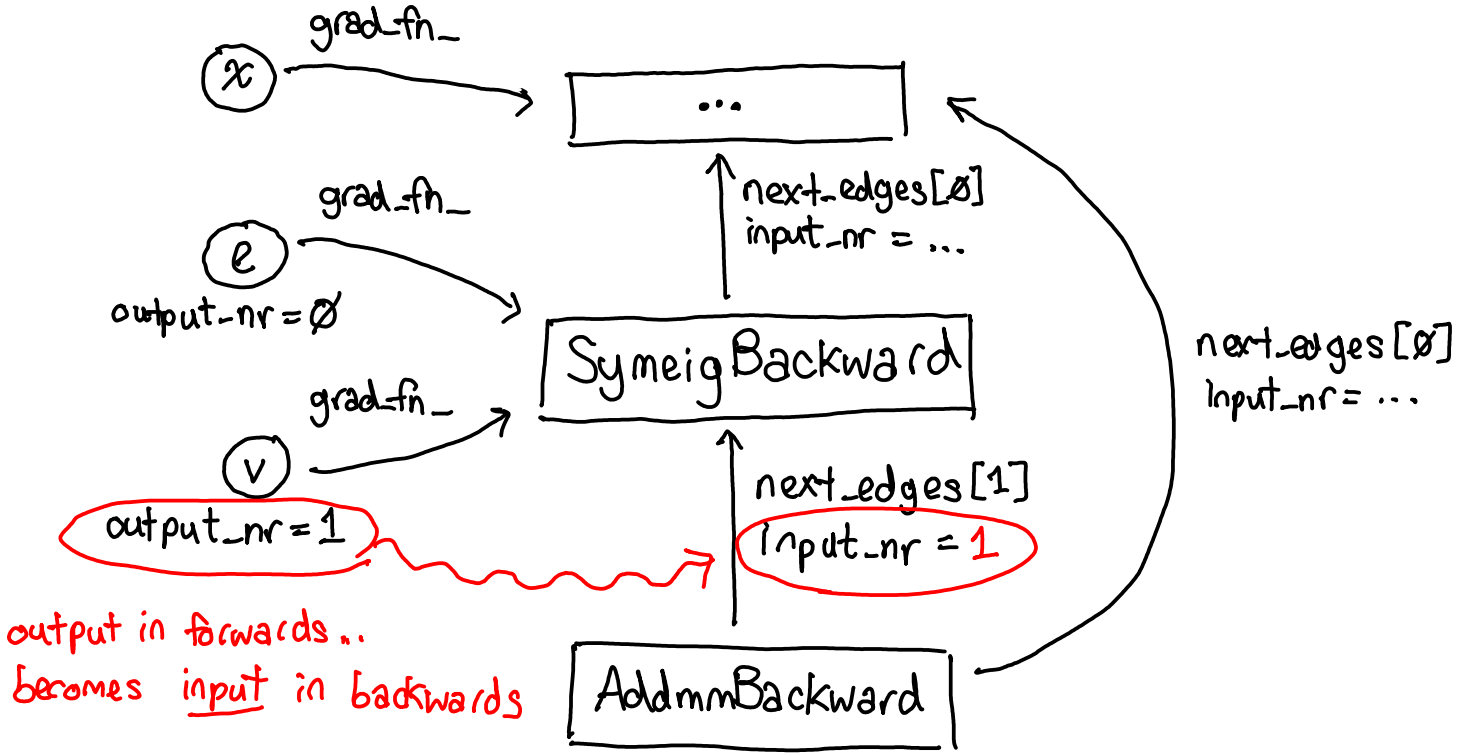
```
i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
next_h = next_h.tanh()
loss = next_h.sum()
loss.backward()
```




requires_grad = True
leaf variable
W_x
grad_acc_
Function
AccumulateGrad
no edge for x, it doesn't require grad!
x
not the input of the forward: input of backward
next_edges[0]
input_nr = 0
next_edges[1]
input_nr = 0
i2h
grad_fn_
Function
MmBackward
mat1_
mat2
W_x
x
output_nr_ = 0
output of forward
not necessarily all inputs
next_edges[1]
input_nr = 0
Function
AddBackward1
some fns have multiple overloads

Multiple outputs

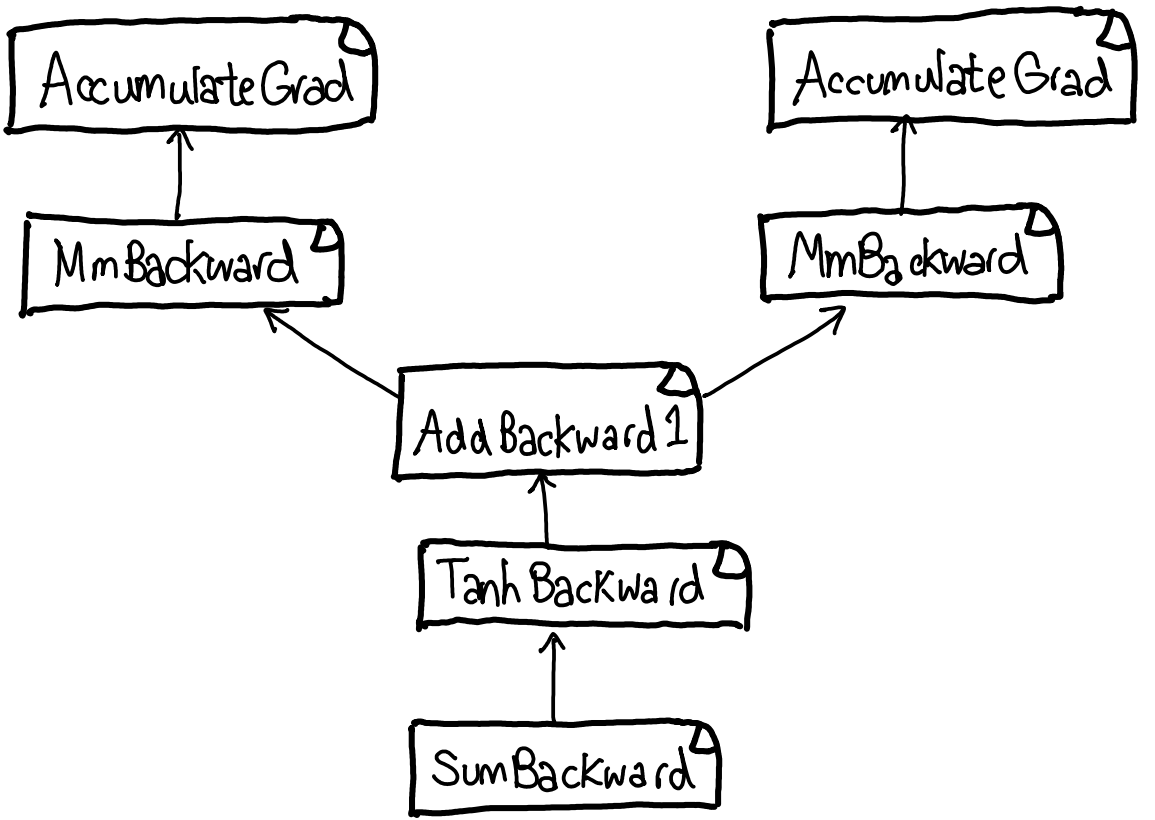
```
e, v = torch.symeig(x, eigenvectors=True)
a = torch.mm(x, v)
```



Autograd engine

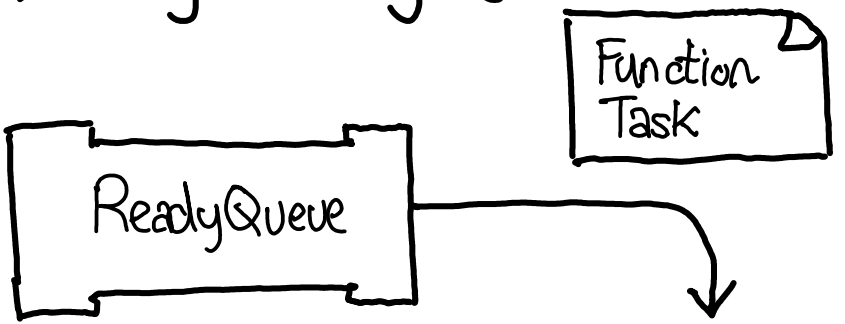
"just a parallel graph executor"

FunctionTasks:



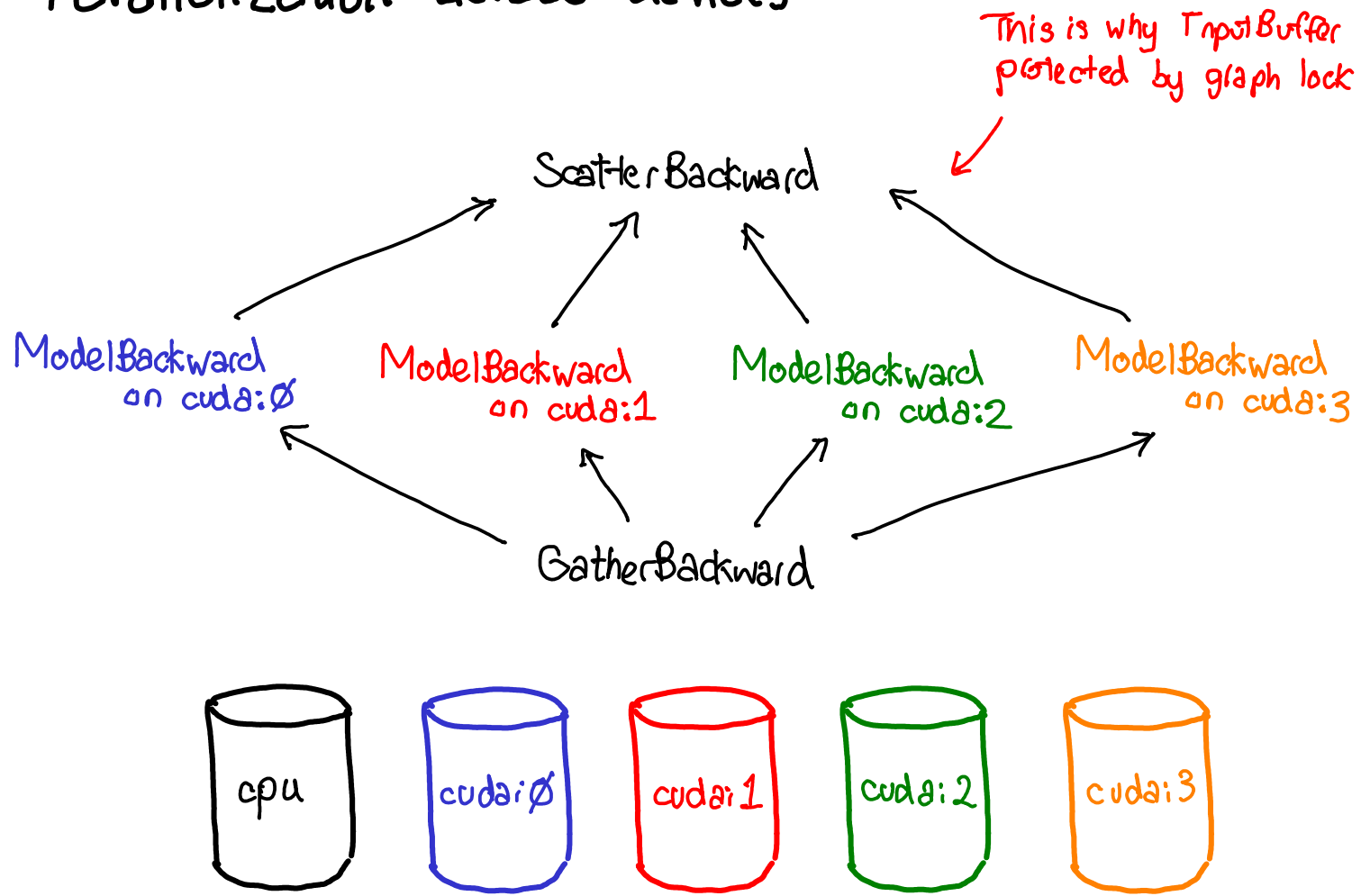
Autograd engine

There's a preprocessing scan of the graph to determine how many dependencies each FunctionTask has



1. Evaluate function on InputBuffer
   a. Call pre-hooks on inputs
   b. Call the function for real
   c. Call post-hooks on outputs
2. Release saved variables
3. With graph lock, accumulate grad into InputBuffers of next functions, and add newly ready FunctionTasks to ReadyQueue

Parallelization across devices



Mechanics

Finding your way around

Source structure

tests & code generation

test/
tools/

torch/
Python; the frontend!

torch/csrc/
Python bindings, this functionality

- api/ C++ API
- autograd/
- jit/ TorchScript

aten/src/ATen/
Tensor operator implementations

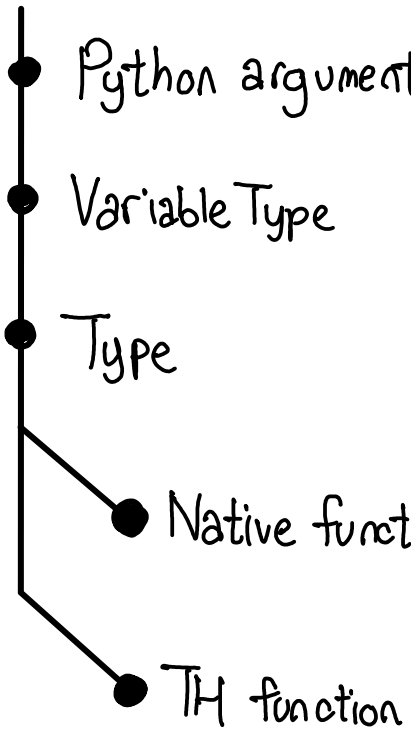
native/ Modern
../TH, THC, ... Legacy

c10/
Core abstractions (e.g. TensorImpl)

Anatomy of an operator call

How to figure this out?
Build PyTorch with DEBUG=1, set a breakpoint on at::native::add, and look at the backtrace!

torch.add(x, y)



THPVariable_add
python_variable_methods.cpp

VariableType::add

TypeDefault::add (or, CPUFloatType)

at::native::add
BinaryOps.cpp

THTensor_(add)
generic/THTensorMath.cpp, e.g.

```
static PyObject * THPVariable_add(PyObject* self_, PyObject* args, PyObject* kwargs)
{
  HANDLE_TH_ERRORS
  static PythonArgParser parser({
    "add(Tensor input, Scalar alpha, Tensor other, *, Tensor out=None)|deprecated",
    "add(Tensor input, Tensor other, *, Scalar alpha=1, Tensor out=None)",
  }, /*traceable=*/true);

  ParsedArgs<4> parsed_args;
  auto r = parser.parse(args, kwargs, parsed_args);

  if (r.idx == 0) {
    if (r.isNone(3)) {
      return wrap(dispatch_add(r.tensor(0), r.scalar(1), r.tensor(2)));
    } else {
      return wrap(dispatch_add(r.tensor(0), r.scalar(1), r.tensor(2), r.tensor(3)));
    }
  } else if (r.idx == 1) {
    if (r.isNone(3)) {
      return wrap(dispatch_add(r.tensor(0), r.tensor(1), r.scalar(2)));
    } else {
      return wrap(dispatch_add(r.tensor(0), r.tensor(1), r.scalar(2), r.tensor(3)));
    }
  }
  Py_RETURN_NONE;
  END_HANDLE_TH_ERRORS
}

static PyMethodDef torch_functions[] = {
  ...
  {"add", (PyCFunction)THPVariable_add, METH_VARARGS | METH_KEYWORDS | METH_STATIC, N
  ...
```

auto-generated! just skim

argument parser

release the GIL

```
AutoNoGIL no_gil;
return self.add(other, alpha);
```

rewrap into PyObject

actual binding
on torch._C.VariableFunctions

File: torch/csrc/autograd/generated/python_torch_functions_dispatch.h

```
inline Tensor Tensor::add(const Tensor & other, Scalar alpha) const {
    return type().add(*this, other, alpha);
}
```

File: aten/src/ATen/core/TensorMethods.h

CPUFloatType

virtual method

might be overriden by subclass

Switch (e.g. CUDA) device

```
Tensor TypeDefault::add(const Tensor & self, const Tensor & other, Scalar alpha) cons
    const OptionalDeviceGuard device_guard(device_of(self));
    return at::native::add(/* native_actuals */ self, other, alpha);
}
```

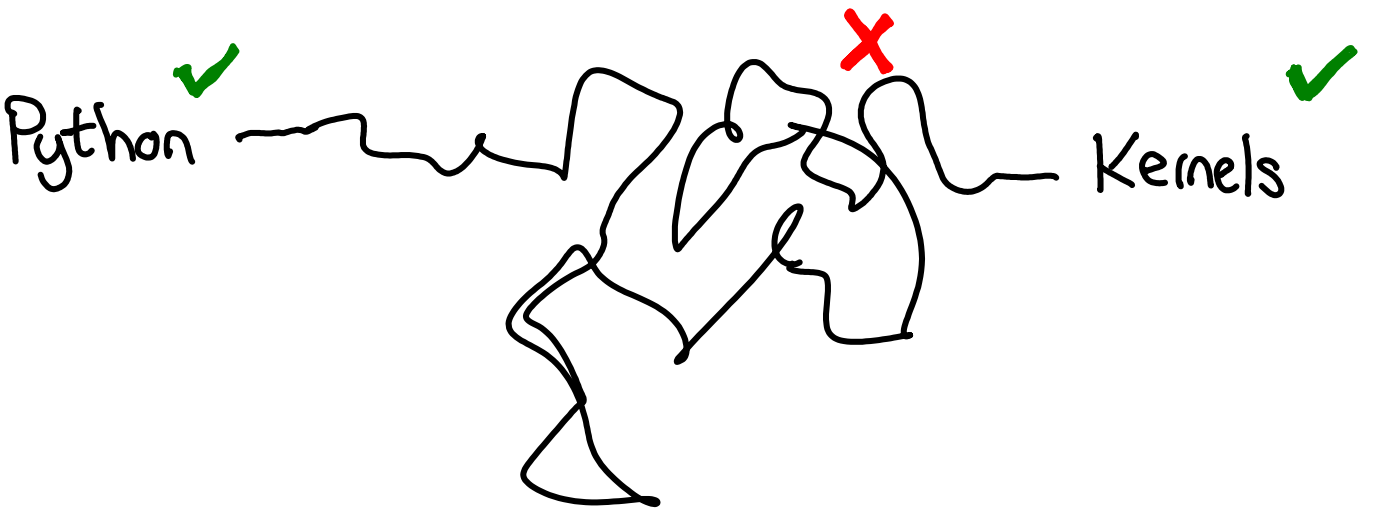
File: build/aten/src/ATen/TypeDefault.cpp

```
Tensor add(const Tensor& self, const Tensor& other, Scalar alpha) {
  Tensor result;
  if (other.is_sparse()) {
    result = at::empty({0}, self.options());
    return native::add_out(result, self, other, alpha);
  }
  auto iter = TensorIterator::binary_op(result, self, other);
  add_stub(iter->device_type(), *iter, alpha);
  return iter->output();
}
```

the kernel proper

File: aten/src/ATen/native/BinaryOps.cpp

I don't recommend spending too much time mucking around backtraces



Writing kernels

Anatomy of a kernel

(+ Infra wiring up!)

```
Tensor my_op_out_cpu(Tensor& result, const Tensor& self, const Tensor& other) {

  TORCH_CHECK(result.is_cpu() && self.is_cpu() && other.is_cpu());
  TORCH_CHECK(self.dim() == 1);
  TORCH_CHECK(self.sizes() == other.sizes());

  result.resize_(self.sizes());

  AT_DISPATCH_FORALL_TYPES(
    self.scalar_type(), "my_op_cpu", [&] {
      my_op_cpu_kernel<scalar_t>(result, self, other);
    }
  );
}


template <typename scalar_t>
void my_op_cpu_kernel(Tensor& result, const Tensor& self, const Tensor& other) {
  auto result_accessor = result.accessor<scalar_t, 1>();
  auto self_accessor = self.accessor<scalar_t, 1>();
  auto other_accessor = other.accessor<scalar_t, 1>();

  parallel_for(0, self.size(0), 0, [&](int64_t start, int64_t end) {
    ... self_accessor[i] ...
  });
}
```

Error checking

Output allocation

Dtype dispatch

Parallelization

Data access (vectorized?!)

Scaffolding

Every operator has a schema

signature

```
- func: add(Tensor self, Tensor other, *, Scalar alpha=1) -> Tensor
  variants: function, method
  dispatch:
    CPU: add
    CUDA: add
    SparseCPU: add
    SparseCUDA: add
    MkldnnCPU: mkldnn_add
```

control Python/C++ API

dispatch

aten/src/ATen/native/native_functions.yaml

This schema controls our code generation

https://github.com/pytorch/pytorch/blob/master/aten/src/ATen/native/README.md

Scaffolding

Also a derivative:

signature

```
- name: add(Tensor self, Tensor other, *, Scalar alpha)
  self: grad
  other: maybe_multiply(grad, alpha)
```

one grad per Tensor argument

the gradient flowing into add

grad_self = grad_add

tools/autograd/derivatives.yaml

Error checking

Low level

Hand-write your error message.

```
#include <c10/util/Exception.h>

TORCH_CHECK(self.dim() == 1, "Expected self to be 1-D tensor, but "
                             "was ", self.dim(), "-D tensor")
```

Condition

Intersperse values directly; they are formatted with operator<<

Higher level

```
#include <ATen/TensorUtils.h>

TensorArg result_arg{result, "result", 0},
          self_arg{self, "self", 1},
          other_arg{other, "other", 2};
CheckedFrom c = "my_op_cpu";
checkDim(c, self_arg, 1);
```

Metadata about each argument

Pre-canned error message

Not complete!

Output allocation

no-op if already the right size

```
Tensor& abs_out(Tensor& result, const Tensor& self) {
  result.resize_(self.sizes());
  // ... the real implementation ...
}

Tensor abs(const Tensor& self) {
  Tensor result = at::empty({0}, self.options());
  abs_out(result, self);
  return result;
}

Tensor& abs_(Tensor& self) {
  return abs_out(self, self);
}
```

preallocated output

generate for me

inplace

assume _out will resize

assumes _out kernel handles aliasing inputs

Dtype dispatch

not actually all types

```
AT_DISPATCH_ALL_TYPES(
  self.scalar_type(), "my_op_cpu", [&] {
    my_op_cpu_kernel<scalar_t>(result, self, other);
  }
);
```

specialize lambda for each scalar_type

```
AT_DISPATCH_ALL_TYPES(TYPE, NAME, ...)
AT_DISPATCH_FLOATING_TYPES(TYPE, NAME, ...)
AT_DISPATCH_INTEGRAL_TYPES(TYPE, NAME, ...)
AT_DISPATCH_ALL_TYPES_AND(SCALARTYPE, TYPE, NAME, ...)
```

Data access

Point access

```
auto x_accessor = x.accessor<float, 3>();
float val = x_accessor[0][0][0];
```

CUDA: packed_accessor (gotcha: 32-bit!)

Elementwise

```
#include <ATen/native/TensorIterator.h
auto iter = TensorIterator::Builder()
  .add_output(output)
  .add_input(input)
  .build();
binary_kernel(iter, [](float a, float b) {
  return a + b;
});
```

CUDA: gpu_binary_kernel

Vectorized

```
#include <ATen/native/cpu/Loops.h>
binary_kernel_vec(iter,
  [=](scalar_t a, scalar_t b) -> scalar_t {
    return a + alpha * b; },
  [=](Vec256<scalar_t> a, Vec256<scalar_t> b) {
    return vec256::fmadd(b, alpha_vec, a);
 });
```

Parallelization (CPU only)

```
#pragma omp parallel for private(i) if (input_size > grain_size)
for (i = 0; i < input_size; i++) {
  ...
}
```

```
at::parallel_for(0, input_size, grain_size, [&](int64_t begin, int64_t end) {
  for (auto i = start; i < end; i++) {
    // ...
  }
});
```

critical? implement the locks yourself!

Legacy code

TH, THC, THNN, THCUNN

- C code
- Manual refcounting
- Preprocessor shenanigans

```
THTensor_wrap()
```

Write new code in C++, consider porting old code when you can!

https://github.com/pytorch/pytorch/wiki/TH-to-ATen-porting-guide

Workflow efficiency

- Don't edit headers
- Don't test by CI
- Do setup ccache
- Get a beefy workstation for builds

How to get involved!

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

high priority

module

small

Port some code from TH to ATen!

Help us improve our documentation!

Help minimize reproducers on bugs!

Help answer questions on the forums

Help us discuss design of new features!

Scratch your own itches!