How JAX primitives work

Contents

necula@google.com, October 2019.

JAX implements certain transformations of Python functions, e.g., `jit`, `grad`, `vmap`, or `pmap`. The Python functions to be transformed must be JAX-traceable, which means that as the Python function executes the only operations it applies to the data are either inspections of data attributes such as shape or type, or special operations called JAX primitives. In particular, a JAX-traceable function is sometimes invoked by JAX with abstract arguments. An example of a JAX abstract value is `ShapedArray(float32[2,2])`, which captures the type and the shape of values, but not the concrete data values. JAX primitives know how to operate on both concrete data values and on the JAX abstract values.

The JAX-transformed functions must themselves be JAX-traceable functions, to ensure that these transformations can be composed, e.g., `jit(jacfwd(grad(f)))`.

There are pre-defined JAX primitives corresponding to most XLA operations, e.g., add, matmul, sin, cos, indexing. JAX comes with an implementation of numpy functions in terms of JAX primitives, which means that Python programs using JAX's implementation of numpy are JAX-traceable and therefore transformable. Other libraries can be made JAX-traceable by implementing them in terms of JAX primitives.

The set of JAX primitives is extensible. Instead of reimplementing a function in terms of pre-defined JAX primitives, one can define a new primitive that encapsulates the behavior of the function.

The goal of this document is to explain the interface that a JAX primitive must support in order to allow JAX to perform all its transformations.

Consider that we want to add to JAX support for a multiply-add function with three arguments, defined mathematically as "multiply_add(x, y, z) = x * y + z". This function operates on 3 identically-shaped tensors of floating point values and performs the operations pointwise.

Using existing primitives

The easiest way to define new functions is to write them in terms of JAX primitives, or in terms of other functions that are themselves written using JAX primitives, e.g., those defined in the `jax.lax` module:

```
from jax import lax
from jax._src import api

def multiply_add_lax(x, y, z):
  """Implementation of multiply-add using the jax.lax primitives."""
  return lax.add(lax.mul(x, y), z)


def square_add_lax(a, b):
  """A square-add function using the newly defined multiply-add."""
  return multiply_add_lax(a, a, b)

print("square_add_lax = ", square_add_lax(2., 10.))
# Differentiate w.r.t. the first argument
print("grad(square_add_lax) = ", api.grad(square_add_lax, argnums=0)(2.0,
10.))
```

```
square_add_lax =  14.0
grad(square_add_lax) =  4.0
```

```
No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun
for more info.)
```

In order to understand how JAX is internally using the primitives, we add some helpers for tracing function calls.

```
#@title Helper functions (execute this cell)
import functools
import traceback

_indentation = 0
def _trace(msg=None):
    """Print a message at current indentation."""
    if msg is not None:
        print("  " * _indentation + msg)

def _trace_indent(msg=None):
    """Print a message and then indent the rest."""
    global _indentation
    _trace(msg)
    _indentation = 1 + _indentation

def _trace_unindent(msg=None):
    """Unindent then print a message."""
    global _indentation
    _indentation = _indentation - 1
    _trace(msg)

def trace(name):
  """A decorator for functions to trace arguments and results."""

  def trace_func(func):  # pylint: disable=missing-docstring
    def pp(v):
        """Print certain values more succinctly"""
        vtype = str(type(v))
        if "jax._src.lib.xla_bridge._JaxComputationBuilder" in vtype:
            return "<JaxComputationBuilder>"
        elif "jaxlib.xla_extension.XlaOp" in vtype:
            return "<XlaOp at 0x{:x}>".format(id(v))
        elif ("partial_eval.JaxprTracer" in vtype or
              "batching.BatchTracer" in vtype or
              "ad.JVPTracer" in vtype):
            return "Traced<{}>".format(v.aval)
        elif isinstance(v, tuple):
            return "({})".format(pp_values(v))
        else:
            return str(v)
    def pp_values(args):
        return ", ".join([pp(arg) for arg in args])

    @functools.wraps(func)
    def func_wrapper(*args):
      _trace_indent("call {}({})".format(name, pp_values(args)))
      res = func(*args)
      _trace_unindent("|<- {} = {}".format(name, pp(res)))
      return res

    return func_wrapper

  return trace_func

class expectNotImplementedError(object):
  """Context manager to check for NotImplementedError."""
  def __enter__(self): pass
  def __exit__(self, type, value, tb):
    global _indentation
    _indentation = 0
    if type is NotImplementedError:
      print("\nFound expected exception:")
```

```
      traceback.print_exc(limit=3)
      return True
    elif type is None:  # No exception
      assert False, "Expected NotImplementedError"
    else:
      return False
```

Instead of using `jax.lax` primitives directly, we can use other functions that are already written in terms of those primitives, such as those in `jax.numpy`:

```
import jax.numpy as jnp
import numpy as np

@trace("multiply_add_numpy")
def multiply_add_numpy(x, y, z):
    return jnp.add(jnp.multiply(x, y), z)

@trace("square_add_numpy")
def square_add_numpy(a, b):
    return multiply_add_numpy(a, a, b)

print("\nNormal evaluation:")
print("square_add_numpy = ", square_add_numpy(2., 10.))
print("\nGradient evaluation:")
print("grad(square_add_numpy) = ", api.grad(square_add_numpy)(2.0, 10.))
```

```
Normal evaluation:
call square_add_numpy(2.0, 10.0)
  call multiply_add_numpy(2.0, 2.0, 10.0)
  |<- multiply_add_numpy = 14.0
|<- square_add_numpy = 14.0
square_add_numpy =  14.0

Gradient evaluation:
call square_add_numpy(Traced<ConcreteArray(2.0, dtype=float32,
weak_type=True)>, 10.0)
  call multiply_add_numpy(Traced<ConcreteArray(2.0, dtype=float32,
weak_type=True)>, Traced<ConcreteArray(2.0, dtype=float32, weak_type=True)>,
10.0)
  |<- multiply_add_numpy = Traced<ConcreteArray(14.0, dtype=float32,
weak_type=True)>
|<- square_add_numpy = Traced<ConcreteArray(14.0, dtype=float32,
weak_type=True)>
grad(square_add_numpy) =  4.0
```

Notice that in the process of computing `grad`, JAX invokes `square_add_numpy` and `multiply_add_numpy` with special arguments `ConcreteArray(...)` (described further below in this colab). It is important to remember that a JAX-traceable function must be able to operate not only on concrete arguments but also on special abstract arguments that JAX may use to abstract the function execution.

The JAX traceability property is satisfied as long as the function is written in terms of JAX primitives.

Defining new JAX primitives

The right way to add support for multiply-add is in terms of existing JAX primitives, as shown above. However, in order to demonstrate how JAX primitives work let us pretend that we want to add a new primitive to JAX for the multiply-add functionality.

```
from jax import core
multiply_add_p = core.Primitive("multiply_add")  # Create the primitive

@trace("multiply_add_prim")
def multiply_add_prim(x, y, z):
  """The JAX-traceable way to use the JAX primitive.

  Note that the traced arguments must be passed as positional arguments
  to `bind`.
  """
  return multiply_add_p.bind(x, y, z)

@trace("square_add_prim")
def square_add_prim(a, b):
  """A square-add function implemented using the new JAX-primitive."""
  return multiply_add_prim(a, a, b)
```

If we try to call the newly defined functions we get an error, because we have not yet told JAX anything about the semantics of the new primitive.

```
with expectNotImplementedError():
  square_add_prim(2., 10.)
```

```
call square_add_prim(2.0, 10.0)
  call multiply_add_prim(2.0, 2.0, 10.0)

Found expected exception:
```

```
Traceback (most recent call last):
  File "/tmp/ipykernel_893/2844449444.py", line 2, in <module>
    square_add_prim(2., 10.)
  File "/tmp/ipykernel_893/2936509082.py", line 48, in func_wrapper
    res = func(*args)
  File "/tmp/ipykernel_893/1308506715.py", line 16, in square_add_prim
    return multiply_add_prim(a, a, b)
NotImplementedError: Evaluation rule for 'multiply_add' not implemented
```

Primal evaluation rules

```
@trace("multiply_add_impl")
def multiply_add_impl(x, y, z):
  """Concrete implementation of the primitive.

  This function does not need to be JAX traceable.
  Args:
    x, y, z: the concrete arguments of the primitive. Will only be called with
      concrete values.
  Returns:
    the concrete result of the primitive.
  """
  # Note that we can use the original numpy, which is not JAX traceable
  return np.add(np.multiply(x, y), z)

# Now we register the primal implementation with JAX
multiply_add_p.def_impl(multiply_add_impl)
```

```
<function __main__.multiply_add_impl(x, y, z)>
```

```
assert square_add_prim(2., 10.) == 14.
```

```
call square_add_prim(2.0, 10.0)
  call multiply_add_prim(2.0, 2.0, 10.0)
    call multiply_add_impl(2.0, 2.0, 10.0)
    |<- multiply_add_impl = 14.0
  |<- multiply_add_prim = 14.0
|<- square_add_prim = 14.0
```

JIT

If we now try to use `jit` we get a `NotImplementedError`:

```
with expectNotImplementedError():
  api.jit(square_add_prim)(2., 10.)
```

```
call square_add_prim(Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>,
Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>)
  call multiply_add_prim(Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>,
Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>,
Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>)

Found expected exception:
```

```
Traceback (most recent call last):
  File "/tmp/ipykernel_893/1813425700.py", line 2, in <module>
    api.jit(square_add_prim)(2., 10.)
  File
"/home/docs/checkouts/readthedocs.org/user_builds/jax/envs/latest/lib/python3.
9/site-packages/jax/_src/traceback_util.py", line 163, in
reraise_with_filtered_traceback
    return fun(*args, **kwargs)
  File
"/home/docs/checkouts/readthedocs.org/user_builds/jax/envs/latest/lib/python3.
9/site-packages/jax/_src/api.py", line 567, in cache_miss
    execute = dispatch._xla_call_impl_lazy(fun_, *tracers, **params)
NotImplementedError: Abstract evaluation for 'multiply_add' not implemented
```

Abstract evaluation rules

In order to JIT the function, and for other transformations as well, JAX first evaluates it abstractly using only the shape and type of the arguments. This abstract evaluation serves multiple purposes:

- Gets the sequence of JAX primitives that are used in the computation. This sequence will be compiled.
- Computes the shape and type of all vectors and operations used in the computation.

For example, the abstraction of a vector with 3 elements may be `ShapedArray(float32[3])`, or `ConcreteArray([1., 2., 3.])`. In the latter case, JAX uses the actual concrete value wrapped as an abstract value.

```python
from jax._src import abstract_arrays
@trace("multiply_add_abstract_eval")
def multiply_add_abstract_eval(xs, ys, zs):
  """Abstract evaluation of the primitive.

  This function does not need to be JAX traceable. It will be invoked with
  abstractions of the actual arguments.
  Args:
    xs, ys, zs: abstractions of the arguments.
  Result:
    a ShapedArray for the result of the primitive.
  """
  assert xs.shape == ys.shape
  assert xs.shape == zs.shape
  return abstract_arrays.ShapedArray(xs.shape, xs.dtype)

# Now we register the abstract evaluation with JAX
multiply_add_p.def_abstract_eval(multiply_add_abstract_eval)
```

```
<function __main__.multiply_add_abstract_eval(xs, ys, zs)>
```

If we re-attempt to JIT, we see how the abstract evaluation proceeds, but we get another error, about missing the actual XLA compilation rule:

```python
with expectNotImplementedError():
  api.jit(square_add_prim)(2., 10.)
```

```
call square_add_prim(Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>,
Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>)
  call multiply_add_prim(Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>,
Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>,
Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>)
    call multiply_add_abstract_eval(ShapedArray(float32[], weak_type=True),
ShapedArray(float32[], weak_type=True), ShapedArray(float32[],
weak_type=True))
    |<- multiply_add_abstract_eval = ShapedArray(float32[])
  |<- multiply_add_prim =
Traced<ShapedArray(float32[])>with<DynamicJaxprTrace(level=0/1)>
|<- square_add_prim =
Traced<ShapedArray(float32[])>with<DynamicJaxprTrace(level=0/1)>

Found expected exception:
```

```
Traceback (most recent call last):
  File "/home/docs/.asdf/installs/python/3.9.15/lib/python3.9/runpy.py", line
197, in _run_module_as_main
    return _run_code(code, main_globals, None,
  File "/home/docs/.asdf/installs/python/3.9.15/lib/python3.9/runpy.py", line
87, in _run_code
    exec(code, run_globals)
  File
"/home/docs/checkouts/readthedocs.org/user_builds/jax/envs/latest/lib/python3.
9/site-packages/ipykernel_launcher.py", line 17, in <module>
    app.launch_new_instance()
jax._src.source_info_util.JaxStackTraceBeforeTransformation:
NotImplementedError: MLIR translation rule for primitive 'multiply_add' not
found for platform cpu

The preceding stack trace is the source of the JAX operation that, once
transformed by JAX, triggered the following exception.

--------------------

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "/tmp/ipykernel_893/1813425700.py", line 2, in <module>
    api.jit(square_add_prim)(2., 10.)
  File
"/home/docs/checkouts/readthedocs.org/user_builds/jax/envs/latest/lib/python3.
9/site-packages/jax/_src/traceback_util.py", line 163, in
reraise_with_filtered_traceback
    return fun(*args, **kwargs)
  File
"/home/docs/checkouts/readthedocs.org/user_builds/jax/envs/latest/lib/python3.
9/site-packages/jax/_src/api.py", line 567, in cache_miss
    execute = dispatch._xla_call_impl_lazy(fun_, *tracers, **params)
NotImplementedError: MLIR translation rule for primitive 'multiply_add' not
found for platform cpu
```

XLA Compilation rules

JAX compilation works by compiling each primitive into a graph of XLA operations.

This is the biggest hurdle to adding new functionality to JAX, because the set of XLA operations is limited, and JAX already has pre-defined primitives for most of them. However, XLA includes a `CustomCall` operation that can be used to encapsulate arbitrary functionality defined using C++.

```
from jax._src.lib import xla_client
@trace("multiply_add_xla_translation")
def multiply_add_xla_translation(ctx, avals_in, avals_out, xc, yc, zc):
  """The compilation to XLA of the primitive.

  Given an XlaBuilder and XlaOps for each argument, return the XlaOp for the
  result of the function.

  Does not need to be a JAX-traceable function.
  """
  return [xla_client.ops.Add(xla_client.ops.Mul(xc, yc), zc)]

# Now we register the XLA compilation rule with JAX
# TODO: for GPU? and TPU?
from jax.interpreters import xla
xla.register_translation(multiply_add_p, multiply_add_xla_translation,
platform='cpu')
```

Now we succeed to JIT. Notice below that JAX first evaluates the function abstractly, which triggers the `multiply_add_abstract_eval` function, and then compiles the set of primitives it has encountered, including `multiply_add`. At this point JAX invokes `multiply_add_xla_translation`.

```
assert api.jit(lambda x, y: square_add_prim(x, y))(2., 10.) == 14.
```

```
call square_add_prim(Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>,
Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>)
  call multiply_add_prim(Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>,
Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>,
Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>)
    call multiply_add_abstract_eval(ShapedArray(float32[], weak_type=True),
ShapedArray(float32[], weak_type=True), ShapedArray(float32[],
weak_type=True))
    |<- multiply_add_abstract_eval = ShapedArray(float32[])
  |<- multiply_add_prim =
Traced<ShapedArray(float32[])>with<DynamicJaxprTrace(level=0/1)>
|<- square_add_prim =
Traced<ShapedArray(float32[])>with<DynamicJaxprTrace(level=0/1)>
call multiply_add_xla_translation(TranslationContext(builder=
<jaxlib.xla_extension.XlaBuilder object at 0x7ff6d0188f30>, platform='cpu',
axis_env=AxisEnv(nreps=1, names=(), sizes=()), name_stack=NameStack(stack=
())), [ShapedArray(float32[], weak_type=True), ShapedArray(float32[],
weak_type=True), ShapedArray(float32[], weak_type=True)],
[ShapedArray(float32[])], <XlaOp at 0x7ff6d13703f0>, <XlaOp at
0x7ff6d0189130>, <XlaOp at 0x7ff6d01891b0>)
|<- multiply_add_xla_translation = [<jaxlib.xla_extension.XlaOp object at
0x7ff6d01890f0>]
```

Below is another use of `jit` where we compile only with respect to the first argument. Notice how the second argument to `square_add_prim` is concrete, which leads in the third argument to `multiply_add_abstract_eval` being `ConcreteArray`. We see that `multiply_add_abstract_eval` may be used with both `ShapedArray` and `ConcreteArray`.

```
assert api.jit(lambda x, y: square_add_prim(x, y),
               static_argnums=1)(2., 10.) == 14.
```

```
call square_add_prim(Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>, 10.0)
  call multiply_add_prim(Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>,
Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>, 10.0)
    call multiply_add_abstract_eval(ShapedArray(float32[], weak_type=True),
ShapedArray(float32[], weak_type=True), ShapedArray(float32[],
weak_type=True))
    |<- multiply_add_abstract_eval = ShapedArray(float32[])
  |<- multiply_add_prim =
Traced<ShapedArray(float32[])>with<DynamicJaxprTrace(level=0/1)>
|<- square_add_prim =
Traced<ShapedArray(float32[])>with<DynamicJaxprTrace(level=0/1)>
call multiply_add_xla_translation(TranslationContext(builder=
<jaxlib.xla_extension.XlaBuilder object at 0x7ff6d0113470>, platform='cpu',
axis_env=AxisEnv(nreps=1, names=(), sizes=()), name_stack=NameStack(stack=
())), [ShapedArray(float32[], weak_type=True), ShapedArray(float32[],
weak_type=True), ShapedArray(float32[], weak_type=True)],
[ShapedArray(float32[])], <XlaOp at 0x7ff6d01136b0>, <XlaOp at
0x7ff6d0113630>, <XlaOp at 0x7ff6d01136f0>)
|<- multiply_add_xla_translation = [<jaxlib.xla_extension.XlaOp object at
0x7ff6d01137f0>]
```

Forward differentiation

JAX implements forward differentiation in the form of a Jacobian-vector product (see the [JAX autodiff cookbook](#)).

If we attempt now to compute the `jvp` function we get an error because we have not yet told JAX how to differentiate the `multiply_add` primitive.

```python
# The second argument `(2., 10.)` are the argument values
# where we evaluate the Jacobian, and the third `(1., 1.)`
# are the values of the tangents for the arguments.
with expectNotImplementedError():
  api.jvp(square_add_prim, (2., 10.), (1., 1.))
```

```
call square_add_prim(Traced<ConcreteArray(2.0, dtype=float32,
weak_type=True)>, Traced<ConcreteArray(10.0, dtype=float32, weak_type=True)>)
  call multiply_add_prim(Traced<ConcreteArray(2.0, dtype=float32,
weak_type=True)>, Traced<ConcreteArray(2.0, dtype=float32, weak_type=True)>,
Traced<ConcreteArray(10.0, dtype=float32, weak_type=True)>)

Found expected exception:
```

```
Traceback (most recent call last):
  File "/tmp/ipykernel_893/800067577.py", line 5, in <module>
    api.jvp(square_add_prim, (2., 10.), (1., 1.))
  File
"/home/docs/checkouts/readthedocs.org/user_builds/jax/envs/latest/lib/python3.
9/site-packages/jax/_src/api.py", line 2404, in jvp
    return _jvp(lu.wrap_init(fun), primals, tangents, has_aux=has_aux)
  File
"/home/docs/checkouts/readthedocs.org/user_builds/jax/envs/latest/lib/python3.
9/site-packages/jax/_src/api.py", line 2433, in _jvp
    out_primals, out_tangents = ad.jvp(flat_fun).call_wrapped(ps_flat,
ts_flat)
NotImplementedError: Differentiation rule for 'multiply_add' not implemented
```

```
from jax.interpreters import ad


@trace("multiply_add_value_and_jvp")
def multiply_add_value_and_jvp(arg_values, arg_tangents):
  """Evaluates the primal output and the tangents (Jacobian-vector product).

  Given values of the arguments and perturbation of the arguments (tangents),
  compute the output of the primitive and the perturbation of the output.

  This method must be JAX-traceable. JAX may invoke it with abstract values
  for the arguments and tangents.

  Args:
    arg_values: a tuple of arguments
    arg_tangents: a tuple with the tangents of the arguments. The tuple has
      the same length as the arg_values. Some of the tangents may also be the
      special value ad.Zero to specify a zero tangent.
  Returns:
     a pair of the primal output and the tangent.
  """
  x, y, z = arg_values
  xt, yt, zt = arg_tangents
  _trace("Primal evaluation:")
  # Now we have a JAX-traceable computation of the output.
  # Normally, we can use the ma primtive itself to compute the primal output.
  primal_out = multiply_add_prim(x, y, z)

  _trace("Tangent evaluation:")
  # We must use a JAX-traceable way to compute the tangent. It turns out that
  # the output tangent can be computed as (xt * y + x * yt + zt),
  # which we can implement in a JAX-traceable way using the same
"multiply_add_prim" primitive.

  # We do need to deal specially with Zero. Here we just turn it into a
  # proper tensor of 0s (of the same shape as 'x').
  # An alternative would be to check for Zero and perform algebraic
  # simplification of the output tangent computation.
  def make_zero(tan):
    return lax.zeros_like_array(x) if type(tan) is ad.Zero else tan

  output_tangent = multiply_add_prim(make_zero(xt), y, multiply_add_prim(x,
make_zero(yt), make_zero(zt)))
  return (primal_out, output_tangent)

# Register the forward differentiation rule with JAX
ad.primitive_jvps[multiply_add_p] = multiply_add_value_and_jvp
```

```
# Tangent is: xt*y + x*yt + zt = 1.*2. + 2.*1. + 1. = 5.
assert api.jvp(square_add_prim, (2., 10.), (1., 1.)) == (14., 5.)
```

```
call square_add_prim(Traced<ConcreteArray(2.0, dtype=float32,
weak_type=True)>, Traced<ConcreteArray(10.0, dtype=float32, weak_type=True)>)
  call multiply_add_prim(Traced<ConcreteArray(2.0, dtype=float32,
weak_type=True)>, Traced<ConcreteArray(2.0, dtype=float32, weak_type=True)>,
Traced<ConcreteArray(10.0, dtype=float32, weak_type=True)>)
    call multiply_add_value_and_jvp((2.0, 2.0, 10.0), (1.0, 1.0, 1.0))
      Primal evaluation:
      call multiply_add_prim(2.0, 2.0, 10.0)
        call multiply_add_impl(2.0, 2.0, 10.0)
        |<- multiply_add_impl = 14.0
      |<- multiply_add_prim = 14.0
      Tangent evaluation:
      call multiply_add_prim(2.0, 1.0, 1.0)
        call multiply_add_impl(2.0, 1.0, 1.0)
        |<- multiply_add_impl = 3.0
      |<- multiply_add_prim = 3.0
      call multiply_add_prim(1.0, 2.0, 3.0)
        call multiply_add_impl(1.0, 2.0, 3.0)
        |<- multiply_add_impl = 5.0
      |<- multiply_add_prim = 5.0
    |<- multiply_add_value_and_jvp = (14.0, 5.0)
  |<- multiply_add_prim = Traced<ConcreteArray(14.0, dtype=float32)>
|<- square_add_prim = Traced<ConcreteArray(14.0, dtype=float32)>
```

TO EXPLAIN:

- Why is JAX using ConcreteArray in square_add_prim? There is no abstract evaluation going on here.
- Not sure how to explain that multiply_add_prim is invoked with ConcreteValue, yet we do not call the multiply_add_abstract_eval.
- I think it would be useful to show the jaxpr here

JIT of forward differentiation

We can apply JIT to the forward differentiation function:

```
assert api.jit(lambda arg_values, arg_tangents:
                   api.jvp(square_add_prim, arg_values, arg_tangents))(
         (2., 10.), (1., 1.)) == (14., 5.)
```

```
call square_add_prim(Traced<ShapedArray(float32[], weak_type=True)>,
Traced<ShapedArray(float32[], weak_type=True)>)
  call multiply_add_prim(Traced<ShapedArray(float32[], weak_type=True)>,
Traced<ShapedArray(float32[], weak_type=True)>, Traced<ShapedArray(float32[],
weak_type=True)>)
    call multiply_add_value_and_jvp((Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>,
Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>,
Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>),
(Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>,
Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>,
Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>))
      Primal evaluation:
      call multiply_add_prim(Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>,
Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>,
Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>)
        call multiply_add_abstract_eval(ShapedArray(float32[],
weak_type=True), ShapedArray(float32[], weak_type=True),
ShapedArray(float32[], weak_type=True))
        |<- multiply_add_abstract_eval = ShapedArray(float32[])
      |<- multiply_add_prim =
Traced<ShapedArray(float32[])>with<DynamicJaxprTrace(level=0/1)>
      Tangent evaluation:
      call multiply_add_prim(Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>,
Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>,
Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>)
        call multiply_add_abstract_eval(ShapedArray(float32[],
weak_type=True), ShapedArray(float32[], weak_type=True),
ShapedArray(float32[], weak_type=True))
        |<- multiply_add_abstract_eval = ShapedArray(float32[])
      |<- multiply_add_prim =
Traced<ShapedArray(float32[])>with<DynamicJaxprTrace(level=0/1)>
      call multiply_add_prim(Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>,
Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>,
Traced<ShapedArray(float32[])>with<DynamicJaxprTrace(level=0/1)>)
        call multiply_add_abstract_eval(ShapedArray(float32[],
weak_type=True), ShapedArray(float32[], weak_type=True),
ShapedArray(float32[]))
        |<- multiply_add_abstract_eval = ShapedArray(float32[])
      |<- multiply_add_prim =
Traced<ShapedArray(float32[])>with<DynamicJaxprTrace(level=0/1)>
    |<- multiply_add_value_and_jvp =
(Traced<ShapedArray(float32[])>with<DynamicJaxprTrace(level=0/1)>,
Traced<ShapedArray(float32[])>with<DynamicJaxprTrace(level=0/1)>)
  |<- multiply_add_prim = Traced<ShapedArray(float32[])>
|<- square_add_prim = Traced<ShapedArray(float32[])>
call multiply_add_xla_translation(TranslationContext(builder=
<jaxlib.xla_extension.XlaBuilder object at 0x7ff6d0123230>, platform='cpu',
axis_env=AxisEnv(nreps=1, names=(), sizes=()), name_stack=NameStack(stack=
```

```
()), [ShapedArray(float32[], weak_type=True), ShapedArray(float32[],
weak_type=True), ShapedArray(float32[], weak_type=True)],
[ShapedArray(float32[])], <XlaOp at 0x7ff6d0123470>, <XlaOp at
0x7ff6d01233f0>, <XlaOp at 0x7ff6d01234b0>)
|<- multiply_add_xla_translation = [<jaxlib.xla_extension.XlaOp object at
0x7ff6d0123530>]
call multiply_add_xla_translation(TranslationContext(builder=
<jaxlib.xla_extension.XlaBuilder object at 0x7ff6d01234f0>, platform='cpu',
axis_env=AxisEnv(nreps=1, names=(), sizes=()), name_stack=NameStack(stack=
()), [ShapedArray(float32[], weak_type=True), ShapedArray(float32[],
weak_type=True), ShapedArray(float32[])], [ShapedArray(float32[])], <XlaOp at
0x7ff6d01238b0>, <XlaOp at 0x7ff6d0123830>, <XlaOp at 0x7ff6d01238f0>)
|<- multiply_add_xla_translation = [<jaxlib.xla_extension.XlaOp object at
0x7ff6d0123970>]
```

Notice that first we evaluate `multiply_add_value_and_jvp` abstractly, which in turn evaluates abstractly both the primal and the tangent evaluation (a total of 3 invocations of the `ma` primitive). Then we compile the 3 occurrences of the primitive.

Reverse differentiation

If we attempt now to use reverse differentiation we see that JAX starts by using the `multiply_add_value_and_jvp` to compute the forward differentiation for abstract values, but then runs into a `NotImplementedError`.

When computing the reverse differentiation JAX first does abstract evaluation of the forward differentiation code `multiply_add_value_and_jvp` to obtain a trace of primitives that compute the output tangent. Observe that JAX performs this abstract evaluation with concrete values for the differentiation point, and abstract values for the tangents. Observe also that JAX uses the special abstract tangent value `Zero` for the tangent corresponding to the 3rd argument of `ma`. This reflects the fact that we do not differentiate w.r.t. the 2nd argument to `square_add_prim`, which flows to the 3rd argument to `multiply_add_prim`.

Observe also that during the abstract evaluation of the tangent we pass the value 0.0 as the tangent for the 3rd argument. This is due to the use of the `make_zero` function in the definition of `multiply_add_value_and_jvp`.

```
# This is reverse differentiation w.r.t. the first argument of square_add_prim
with expectNotImplementedError():
  api.grad(square_add_prim)(2., 10.)
```

```
call square_add_prim(Traced<ConcreteArray(2.0, dtype=float32,
weak_type=True)>, 10.0)
  call multiply_add_prim(Traced<ConcreteArray(2.0, dtype=float32,
weak_type=True)>, Traced<ConcreteArray(2.0, dtype=float32, weak_type=True)>,
10.0)
    call multiply_add_value_and_jvp((2.0, 2.0, 10.0),
(Traced<ShapedArray(float32[], weak_type=True)>, Traced<ShapedArray(float32[],
weak_type=True)>, Zero(ShapedArray(float32[], weak_type=True))))
      Primal evaluation:
      call multiply_add_prim(2.0, 2.0, 10.0)
        call multiply_add_impl(2.0, 2.0, 10.0)
        |<- multiply_add_impl = 14.0
      |<- multiply_add_prim = 14.0
      Tangent evaluation:
      call multiply_add_prim(2.0, Traced<ShapedArray(float32[],
weak_type=True)>, 0.0)
        call multiply_add_abstract_eval(ConcreteArray(2.0, dtype=float32,
weak_type=True), ShapedArray(float32[], weak_type=True), ConcreteArray(0.0,
dtype=float32, weak_type=True))
        |<- multiply_add_abstract_eval = ShapedArray(float32[])
      |<- multiply_add_prim = Traced<ShapedArray(float32[])>
      call multiply_add_prim(Traced<ShapedArray(float32[], weak_type=True)>,
2.0, Traced<ShapedArray(float32[])>)
        call multiply_add_abstract_eval(ShapedArray(float32[],
weak_type=True), ConcreteArray(2.0, dtype=float32, weak_type=True),
ShapedArray(float32[]))
        |<- multiply_add_abstract_eval = ShapedArray(float32[])
      |<- multiply_add_prim = Traced<ShapedArray(float32[])>
    |<- multiply_add_value_and_jvp = (14.0, Traced<ShapedArray(float32[])>)
  |<- multiply_add_prim = Traced<ConcreteArray(14.0, dtype=float32)>
|<- square_add_prim = Traced<ConcreteArray(14.0, dtype=float32)>

Found expected exception:
```

```
Traceback (most recent call last):
  File
"/home/docs/checkouts/readthedocs.org/user_builds/jax/envs/latest/lib/python3.
9/site-packages/jax/_src/interpreters/ad.py", line 281, in
get_primitive_transpose
    return primitive_transposes[p]
KeyError: multiply_add

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "/home/docs/.asdf/installs/python/3.9.15/lib/python3.9/runpy.py", line
197, in _run_module_as_main
    return _run_code(code, main_globals, None,
  File "/home/docs/.asdf/installs/python/3.9.15/lib/python3.9/runpy.py", line
87, in _run_code
    exec(code, run_globals)
  File
"/home/docs/checkouts/readthedocs.org/user_builds/jax/envs/latest/lib/python3.
9/site-packages/ipykernel_launcher.py", line 17, in <module>
    app.launch_new_instance()
jax._src.source_info_util.JaxStackTraceBeforeTransformation:
NotImplementedError: Transpose rule (for reverse-mode differentiation) for
'multiply_add' not implemented

The preceding stack trace is the source of the JAX operation that, once
transformed by JAX, triggered the following exception.

--------------------

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "/tmp/ipykernel_893/339076514.py", line 3, in <module>
    api.grad(square_add_prim)(2., 10.)
  File
"/home/docs/checkouts/readthedocs.org/user_builds/jax/envs/latest/lib/python3.
9/site-packages/jax/_src/traceback_util.py", line 163, in
reraise_with_filtered_traceback
    return fun(*args, **kwargs)
  File
"/home/docs/checkouts/readthedocs.org/user_builds/jax/envs/latest/lib/python3.
9/site-packages/jax/_src/api.py", line 1039, in grad_f
    _, g = value_and_grad_f(*args, **kwargs)
NotImplementedError: Transpose rule (for reverse-mode differentiation) for
'multiply_add' not implemented
```

The above error is because there is a missing piece for JAX to be able to use the forward differentiation code to compute reverse differentiation.

Transposition

As explained above, when computing reverse differentiation JAX obtains a trace of primitives that compute the tangent using forward differentiation. Then, **JAX interprets this trace abstractly backwards** and for each primitive it applies a **transposition** rule.

To understand what is going on, consider for now a simpler example of the function “f(x, y) = x * y + y”. Assume we need to differentiate at the point `(2., 4.)`. JAX will produce the following JVP tangent calculation of `ft` from the tangents of the input `xt` and `yt`:

```
a = xt * 4.
b = 2. * yt
c = a + b
ft = c + yt
```

By construction, the tangent calculation is always linear in the input tangents. The only non-linear operator that may arise in the tangent calculation is multiplication, but then one of the operands is constant.

JAX will produce the reverse differentiation computation by processing the JVP computation backwards. For each operation in the tangent computation, it accumulates the cotangents of the variables used by the operation, using the cotangent of the result of the operation:

```
# Initialize cotangents of inputs and intermediate vars
xct = yct = act = bct = cct = 0.
# Initialize cotangent of the output
fct = 1.
# Process "ft = c + yt"
cct += fct
yct += fct
# Process "c = a + b"
act += cct
bct += cct
# Process "b = 2. * yt"
yct += 2. * bct
# Process "a = xt * 4."
xct += act * 4.
```

One can verify that this computation produces `xct = 4.` and `yct = 3.`, which are the partial derivatives of the function `f`.

JAX knows for each primitive that may appear in a JVP calculation how to transpose it. Conceptually, if the primitive `p(x, y, z)` is linear in the arguments `y` and `z` for a constant value of `x`, e.g., `p(x, y, z) = y*cy + z*cz`, then the transposition of the primitive is:

```
p_transpose(out_ct, x, _, _) = (None, out_ct*cy, out_ct*cz)
```

Notice that `p_transpose` takes the cotangent of the output of the primitive and a value corresponding to each argument of the primitive. For the linear arguments, the transposition gets an undefined `_` value, and for the other arguments it gets the actual constants. The transposition returns a cotangent value for each argument of the primitive, with the value `None` returned for the constant arguments.

In particular,

```
add_transpose(out_ct, _, _) = (out_ct, out_ct)
mult_transpose(out_ct, x, _) = (None, x * out_ct)
mult_transpose(out_ct, _, y) = (out_ct * y, None)
```

```
@trace("multiply_add_transpose")
def multiply_add_transpose(ct, x, y, z):
  """Evaluates the transpose of a linear primitive.

  This method is only used when computing the backward gradient following
  value_and_jvp, and is only needed for primitives that are used in the JVP
  calculation for some other primitive. We need transposition for
multiply_add_prim,
  because we have used multiply_add_prim in the computation of the
output_tangent in
  multiply_add_value_and_jvp.

  In our case, multiply_add is not a linear primitive. However, it is used
linearly
  w.r.t. tangents in multiply_add_value_and_jvp:
       output_tangent(xt, yt, zt) = multiply_add_prim(xt, y,
multiply_add_prim(x, yt, zt))

  Always one of the first two multiplicative arguments is a constant.

  Args:
      ct: the cotangent of the output of the primitive.
      x, y, z: values of the arguments. The arguments that are used linearly
        get an ad.UndefinedPrimal value. The other arguments get a constant
        value.
  Returns:
      a tuple with the cotangent of the inputs, with the value None
      corresponding to the constant arguments.
  """
  if not ad.is_undefined_primal(x):
    # This use of multiply_add is with a constant "x"
    assert ad.is_undefined_primal(y)
    ct_y = ad.Zero(y.aval) if type(ct) is ad.Zero else multiply_add_prim(x,
ct, lax.zeros_like_array(x))
    res = None, ct_y, ct
  else:
    # This use of multiply_add is with a constant "y"
    assert ad.is_undefined_primal(x)
    ct_x = ad.Zero(x.aval) if type(ct) is ad.Zero else multiply_add_prim(ct,
y, lax.zeros_like_array(y))
    res = ct_x, None, ct
  return res


ad.primitive_transposes[multiply_add_p] = multiply_add_transpose
```

Now we can complete the run of the `grad`:

```
assert api.grad(square_add_prim)(2., 10.) == 4.
```

```
call square_add_prim(Traced<ConcreteArray(2.0, dtype=float32,
weak_type=True)>, 10.0)
  call multiply_add_prim(Traced<ConcreteArray(2.0, dtype=float32,
weak_type=True)>, Traced<ConcreteArray(2.0, dtype=float32, weak_type=True)>,
10.0)
    call multiply_add_value_and_jvp((2.0, 2.0, 10.0),
(Traced<ShapedArray(float32[], weak_type=True)>, Traced<ShapedArray(float32[],
weak_type=True)>, Zero(ShapedArray(float32[], weak_type=True))))
      Primal evaluation:
      call multiply_add_prim(2.0, 2.0, 10.0)
        call multiply_add_impl(2.0, 2.0, 10.0)
        |<- multiply_add_impl = 14.0
      |<- multiply_add_prim = 14.0
      Tangent evaluation:
      call multiply_add_prim(2.0, Traced<ShapedArray(float32[],
weak_type=True)>, 0.0)
        call multiply_add_abstract_eval(ConcreteArray(2.0, dtype=float32,
weak_type=True), ShapedArray(float32[], weak_type=True), ConcreteArray(0.0,
dtype=float32, weak_type=True))
        |<- multiply_add_abstract_eval = ShapedArray(float32[])
      |<- multiply_add_prim = Traced<ShapedArray(float32[])>
      call multiply_add_prim(Traced<ShapedArray(float32[], weak_type=True)>,
2.0, Traced<ShapedArray(float32[])>)
        call multiply_add_abstract_eval(ShapedArray(float32[],
weak_type=True), ConcreteArray(2.0, dtype=float32, weak_type=True),
ShapedArray(float32[]))
        |<- multiply_add_abstract_eval = ShapedArray(float32[])
      |<- multiply_add_prim = Traced<ShapedArray(float32[])>
    |<- multiply_add_value_and_jvp = (14.0, Traced<ShapedArray(float32[])>)
  |<- multiply_add_prim = Traced<ConcreteArray(14.0, dtype=float32)>
|<- square_add_prim = Traced<ConcreteArray(14.0, dtype=float32)>
call multiply_add_transpose(1.0, UndefinedPrimal(ShapedArray(float32[],
weak_type=True)), 2.0, UndefinedPrimal(ShapedArray(float32[])))
  call multiply_add_prim(1.0, 2.0, 0.0)
    call multiply_add_impl(1.0, 2.0, 0.0)
    |<- multiply_add_impl = 2.0
  |<- multiply_add_prim = 2.0
|<- multiply_add_transpose = (2.0, None, 1.0)
call multiply_add_transpose(1.0, 2.0, UndefinedPrimal(ShapedArray(float32[],
weak_type=True)), 0.0)
  call multiply_add_prim(2.0, 1.0, 0.0)
    call multiply_add_impl(2.0, 1.0, 0.0)
    |<- multiply_add_impl = 2.0
  |<- multiply_add_prim = 2.0
|<- multiply_add_transpose = (None, 2.0, 1.0)
```

Notice the two calls to `multiply_add_transpose`. They correspond to the two uses of `multiply_add_prim` in the computation of the `output_tangent` in `multiply_add_value_and_jvp`. The first call to transpose corresponds to the last use of `multiply_add_prim`: `multiply_add_prim(xt, y, ...)` where `y` is the constant 2.0.

JIT of reverse differentiation

Notice that the abstract evaluation of the `multiply_add_value_and_jvp` is using only abstract values, while in the absence of JIT we used `ConcreteArray`.

```
assert api.jit(api.grad(square_add_prim))(2., 10.) == 4.
```

```
call square_add_prim(Traced<ShapedArray(float32[], weak_type=True)>,
Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>)
  call multiply_add_prim(Traced<ShapedArray(float32[], weak_type=True)>,
Traced<ShapedArray(float32[], weak_type=True)>, Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>)
    call multiply_add_value_and_jvp((Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>,
Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>,
Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>),
(Traced<ShapedArray(float32[], weak_type=True)>, Traced<ShapedArray(float32[],
weak_type=True)>, Zero(ShapedArray(float32[], weak_type=True))))
      Primal evaluation:
      call multiply_add_prim(Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>,
Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>,
Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>)
        call multiply_add_abstract_eval(ShapedArray(float32[],
weak_type=True), ShapedArray(float32[], weak_type=True),
ShapedArray(float32[], weak_type=True))
        |<- multiply_add_abstract_eval = ShapedArray(float32[])
      |<- multiply_add_prim =
Traced<ShapedArray(float32[])>with<DynamicJaxprTrace(level=0/1)>
      Tangent evaluation:
      call multiply_add_prim(Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>,
Traced<ShapedArray(float32[], weak_type=True)>, Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>)
        call multiply_add_abstract_eval(ShapedArray(float32[],
weak_type=True), ShapedArray(float32[], weak_type=True),
ShapedArray(float32[], weak_type=True))
        |<- multiply_add_abstract_eval = ShapedArray(float32[])
      |<- multiply_add_prim = Traced<ShapedArray(float32[])>
      call multiply_add_prim(Traced<ShapedArray(float32[], weak_type=True)>,
Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>,
Traced<ShapedArray(float32[])>)
        call multiply_add_abstract_eval(ShapedArray(float32[],
weak_type=True), ShapedArray(float32[], weak_type=True),
ShapedArray(float32[]))
        |<- multiply_add_abstract_eval = ShapedArray(float32[])
      |<- multiply_add_prim = Traced<ShapedArray(float32[])>
    |<- multiply_add_value_and_jvp =
(Traced<ShapedArray(float32[])>with<DynamicJaxprTrace(level=0/1)>,
Traced<ShapedArray(float32[])>)
  |<- multiply_add_prim = Traced<ShapedArray(float32[])>
|<- square_add_prim = Traced<ShapedArray(float32[])>
call
multiply_add_transpose(Traced<ShapedArray(float32[])>with<DynamicJaxprTrace(le
vel=0/1)>, UndefinedPrimal(ShapedArray(float32[], weak_type=True)),
Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>,
UndefinedPrimal(ShapedArray(float32[])))
  call
multiply_add_prim(Traced<ShapedArray(float32[])>with<DynamicJaxprTrace(level=0
/1)>, Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>,
Traced<ShapedArray(float32[],
```

```
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>)
    call multiply_add_abstract_eval(ShapedArray(float32[]),
ShapedArray(float32[], weak_type=True), ShapedArray(float32[],
weak_type=True))
    |<- multiply_add_abstract_eval = ShapedArray(float32[])
  |<- multiply_add_prim =
Traced<ShapedArray(float32[])>with<DynamicJaxprTrace(level=0/1)>
|<- multiply_add_transpose =
(Traced<ShapedArray(float32[])>with<DynamicJaxprTrace(level=0/1)>, None,
Traced<ShapedArray(float32[])>with<DynamicJaxprTrace(level=0/1)>)
call
multiply_add_transpose(Traced<ShapedArray(float32[])>with<DynamicJaxprTrace(le
vel=0/1)>, Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>,
UndefinedPrimal(ShapedArray(float32[], weak_type=True)),
Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>)
  call multiply_add_prim(Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>,
Traced<ShapedArray(float32[])>with<DynamicJaxprTrace(level=0/1)>,
Traced<ShapedArray(float32[],
weak_type=True)>with<DynamicJaxprTrace(level=0/1)>)
    call multiply_add_abstract_eval(ShapedArray(float32[], weak_type=True),
ShapedArray(float32[]), ShapedArray(float32[], weak_type=True))
    |<- multiply_add_abstract_eval = ShapedArray(float32[])
  |<- multiply_add_prim =
Traced<ShapedArray(float32[])>with<DynamicJaxprTrace(level=0/1)>
|<- multiply_add_transpose = (None,
Traced<ShapedArray(float32[])>with<DynamicJaxprTrace(level=0/1)>,
Traced<ShapedArray(float32[])>with<DynamicJaxprTrace(level=0/1)>)
call multiply_add_xla_translation(TranslationContext(builder=
<jaxlib.xla_extension.XlaBuilder object at 0x7ff6d013adf0>, platform='cpu',
axis_env=AxisEnv(nreps=1, names=(), sizes=()), name_stack=NameStack(stack=
())), [ShapedArray(float32[]), ShapedArray(float32[], weak_type=True),
ShapedArray(float32[], weak_type=True)], [ShapedArray(float32[])], <XlaOp at
0x7ff6d013b070>, <XlaOp at 0x7ff6d013afb0>, <XlaOp at 0x7ff6d013b0b0>)
|<- multiply_add_xla_translation = [<jaxlib.xla_extension.XlaOp object at
0x7ff6d013b130>]
call multiply_add_xla_translation(TranslationContext(builder=
<jaxlib.xla_extension.XlaBuilder object at 0x7ff6d013b270>, platform='cpu',
axis_env=AxisEnv(nreps=1, names=(), sizes=()), name_stack=NameStack(stack=
())), [ShapedArray(float32[], weak_type=True), ShapedArray(float32[]),
ShapedArray(float32[], weak_type=True)], [ShapedArray(float32[])], <XlaOp at
0x7ff6d013b1f0>, <XlaOp at 0x7ff6d013b030>, <XlaOp at 0x7ff6d013b3f0>)
|<- multiply_add_xla_translation = [<jaxlib.xla_extension.XlaOp object at
0x7ff6d013b470>]
```

Batching

The batching transformation takes a point-wise computation and turns it into a computation on vectors. If we try it right now, we get a `NotImplementedError`:

```python
# The arguments are two vectors instead of two scalars
with expectNotImplementedError():
  api.vmap(square_add_prim, in_axes=0, out_axes=0)(np.array([2., 3.]),
                                               np.array([10., 20.]))
```

```
call square_add_prim(Traced<ShapedArray(float32[])>,
Traced<ShapedArray(float32[])>)
  call multiply_add_prim(Traced<ShapedArray(float32[])>,
Traced<ShapedArray(float32[])>, Traced<ShapedArray(float32[])>)

Found expected exception:
```

```
Traceback (most recent call last):
  File "/tmp/ipykernel_893/2641678767.py", line 3, in <module>
    api.vmap(square_add_prim, in_axes=0, out_axes=0)(np.array([2., 3.]),
  File
"/home/docs/checkouts/readthedocs.org/user_builds/jax/envs/latest/lib/python3.
9/site-packages/jax/_src/traceback_util.py", line 163, in
reraise_with_filtered_traceback
    return fun(*args, **kwargs)
  File
"/home/docs/checkouts/readthedocs.org/user_builds/jax/envs/latest/lib/python3.
9/site-packages/jax/_src/api.py", line 1630, in vmap_f
    out_flat = batching.batch(
NotImplementedError: Batching rule for 'multiply_add' not implemented
```

We need to tell JAX how to evaluate the batched version of the primitive. In this particular case, the `multiply_add_prim` already operates pointwise for any dimension of input vectors. So the batched version can use the same `multiply_add_prim` implementation.

```python
from jax.interpreters import batching


@trace("multiply_add_batch")
def multiply_add_batch(vector_arg_values, batch_axes):
  """Computes the batched version of the primitive.

  This must be a JAX-traceable function.

  Since the multiply_add primitive already operates pointwise on arbitrary
  dimension tensors, to batch it we can use the primitive itself. This works
as
  long as both the inputs have the same dimensions and are batched along the
  same axes. The result is batched along the axis that the inputs are batched.

  Args:
    vector_arg_values: a tuple of two arguments, each being a tensor of
matching
      shape.
    batch_axes: the axes that are being batched. See vmap documentation.
  Returns:
    a tuple of the result, and the result axis that was batched.
  """
  assert batch_axes[0] == batch_axes[1]
  assert batch_axes[0] == batch_axes[2]
  _trace("Using multiply_add to compute the batch:")
  res = multiply_add_prim(*vector_arg_values)
  return res, batch_axes[0]


batching.primitive_batchers[multiply_add_p] = multiply_add_batch
```

```
assert np.allclose(api.vmap(square_add_prim, in_axes=0, out_axes=0)(
  np.array([2., 3.]),
  np.array([10., 20.])),
  [14., 29.])
```

```
call square_add_prim(Traced<ShapedArray(float32[])>,
Traced<ShapedArray(float32[])>)
  call multiply_add_prim(Traced<ShapedArray(float32[])>,
Traced<ShapedArray(float32[])>, Traced<ShapedArray(float32[])>)
    call multiply_add_batch(([2. 3.], [2. 3.], [10. 20.]), (0, 0, 0))
      Using multiply_add to compute the batch:
      call multiply_add_prim([2. 3.], [2. 3.], [10. 20.])
        call multiply_add_impl([2. 3.], [2. 3.], [10. 20.])
        |<- multiply_add_impl = [14. 29.]
      |<- multiply_add_prim = [14. 29.]
    |<- multiply_add_batch = ([14. 29.], 0)
  |<- multiply_add_prim = Traced<ShapedArray(float32[])>
|<- square_add_prim = Traced<ShapedArray(float32[])>
```

JIT of batching

```
assert np.allclose(api.jit(api.vmap(square_add_prim, in_axes=0, out_axes=0))
                    (np.array([2., 3.]),
                     np.array([10., 20.])),
                    [14., 29.])
```

```
call square_add_prim(Traced<ShapedArray(float32[])>,
Traced<ShapedArray(float32[])>)
  call multiply_add_prim(Traced<ShapedArray(float32[])>,
Traced<ShapedArray(float32[])>, Traced<ShapedArray(float32[])>)
    call
multiply_add_batch((Traced<ShapedArray(float32[2])>with<DynamicJaxprTrace(leve
l=0/1)>, Traced<ShapedArray(float32[2])>with<DynamicJaxprTrace(level=0/1)>,
Traced<ShapedArray(float32[2])>with<DynamicJaxprTrace(level=0/1)>), (0, 0, 0))
      Using multiply_add to compute the batch:
      call
multiply_add_prim(Traced<ShapedArray(float32[2])>with<DynamicJaxprTrace(level=
0/1)>, Traced<ShapedArray(float32[2])>with<DynamicJaxprTrace(level=0/1)>,
Traced<ShapedArray(float32[2])>with<DynamicJaxprTrace(level=0/1)>)
        call multiply_add_abstract_eval(ShapedArray(float32[2]),
ShapedArray(float32[2]), ShapedArray(float32[2]))
        |<- multiply_add_abstract_eval = ShapedArray(float32[2])
      |<- multiply_add_prim =
Traced<ShapedArray(float32[2])>with<DynamicJaxprTrace(level=0/1)>
    |<- multiply_add_batch =
(Traced<ShapedArray(float32[2])>with<DynamicJaxprTrace(level=0/1)>, 0)
  |<- multiply_add_prim = Traced<ShapedArray(float32[])>
|<- square_add_prim = Traced<ShapedArray(float32[])>
call multiply_add_xla_translation(TranslationContext(builder=
<jaxlib.xla_extension.XlaBuilder object at 0x7ff6d013b7b0>, platform='cpu',
axis_env=AxisEnv(nreps=1, names=(), sizes=()), name_stack=NameStack(stack=
())), [ShapedArray(float32[2]), ShapedArray(float32[2]),
ShapedArray(float32[2])], [ShapedArray(float32[2])], <XlaOp at
0x7ff6d013b9f0>, <XlaOp at 0x7ff6d013b970>, <XlaOp at 0x7ff6d013ba30>)
|<- multiply_add_xla_translation = [<jaxlib.xla_extension.XlaOp object at
0x7ff6d013bb70>]
```