onnx_fft

April 5, 2022

1 ONNX and FFT

ONNX does not fully support complex yet. It does not have any FFT operators either. What if we need them anyway?

[1]:
```
from jyquickhelper import add_notebook_menu
add_notebook_menu()
```

[1]:
```
<IPython.core.display.HTML object>
```

[2]:
```
%load_ext mlprodict
```

[3]:
```
import numpy
numpy.__version__
```

[3]:
```
'1.21.5'
```

1.1 Python implementation of RFFT

We try to replicate numpy.rfft.

[4]:
```
import numpy


def almost_equal(a, b, error=1e-5):
    """
    The function compares two matrices, one may be complex. In that case,
    this matrix is changed into a new matrix with a new first dimension,
    [0,::] means real part, [1,::] means imaginary part.
    """
    if a.dtype in (numpy.complex64, numpy.complex128):
        dtype = numpy.float64 if a.dtype == numpy.complex128 else numpy.float32
        new_a = numpy.empty((2,) + a.shape).astype(dtype)
        new_a[0] = numpy.real(a)
        new_a[1] = numpy.imag(a)
        return almost_equal(new_a, b, error)
    if b.dtype in (numpy.complex64, numpy.complex128):
        return almost_equal(b, a, error)
    if a.shape != b.shape:
        raise AssertionError("Shape mismatch %r != %r." % (a.shape, b.shape))
    diff = numpy.abs(a.ravel() - b.ravel()).max()
    if diff > error:
```

```
        raise AssertionError("Mismatch max diff=%r > %r." % (diff, error))


def dft_real_cst(N, fft_length):
    n = numpy.arange(N)
    k = n.reshape((N, 1)).astype(numpy.float64)
    M = numpy.exp(-2j * numpy.pi * k * n / fft_length)
    both = numpy.empty((2,) + M.shape)
    both[0, :, :] = numpy.real(M)
    both[1, :, :] = numpy.imag(M)
    return both


def dft_real(x, fft_length=None, transpose=True):
    if len(x.shape) == 1:
        x = x.reshape((1, -1))
        N = 1
    else:
        N = x.shape[0]
    C = x.shape[-1] if transpose else x.shape[-2]
    if fft_length is None:
        fft_length = x.shape[-1]
    size = fft_length // 2 + 1

    cst = dft_real_cst(C, fft_length)
    if transpose:
        x = numpy.transpose(x, (1, 0))
        a = cst[:, :, :fft_length]
        b = x[:fft_length]
        res = numpy.matmul(a, b)
        res = res[:, :size, :]
        return numpy.transpose(res, (0, 2, 1))
    else:
        a = cst[:, :, :fft_length]
        b = x[:fft_length]
        return numpy.matmul(a, b)


rnd = numpy.random.randn(5, 7).astype(numpy.float32)
fft_np = numpy.fft.rfft(rnd)
fft_cus = dft_real(rnd)
fft_np
```

[4]:
```
array([[-0.33227623+0.j        , -1.53729601-0.93413037j,
         4.47973719+2.89019374j,  1.36392938-2.59133368j],
       [ 0.07591467+0.j        ,  0.51947711+0.624144j  ,
        -2.48242622-1.56579382j, -0.98728199+2.81434946j],
       [-0.55875075+0.j        , -0.83228203+2.25251549j,
         0.48281369+2.69338405j, -0.86559293+0.08437194j],
       [ 0.26185111+0.j        , -1.18143684+1.73623491j,
         0.96002386+0.39340971j,  3.53861562-1.32858241j],
       [ 1.06276855+0.j        ,  3.07258661-2.71505518j,
        -0.82579331-1.91852778j,  4.10811113-0.46836687j]])
```

Function `almost_equal` verifies both functions return the same results.

[5]:
```
almost_equal(fft_np, fft_cus)
```

Let's do the same with `fft_length < shape[1]`.

[6]:
```
fft_np3 = numpy.fft.rfft(rnd, n=3)
fft_cus3 = dft_real(rnd, fft_length=3)
fft_np3
```

[6]:
```
array([[-0.86976612+0.j        ,  2.20926839+0.35688821j],
       [ 0.33280143+0.j        , -1.41451804+0.2065253j ],
       [-2.30690554+0.j        ,  0.51297992+0.62331197j],
       [-0.72842433+0.j        ,  1.84198139+1.07546916j],
       [ 4.17533261+0.j        ,  0.86360028+0.36508775j]])
```

[7]:
```
almost_equal(fft_np3, fft_cus3)
```

1.2 RFFT in ONNX

Let's assume first the number of column of the input matrix is fixed. The result of function `dft_real_cst` can be considered as constant.

[8]:
```
from typing import Any
import mlprodict.npy.numpy_onnx_impl as npnx
from mlprodict.npy import onnxnumpy_np
from mlprodict.npy.onnx_numpy_annotation import NDArrayType
# from mlprodict.onnxrt import OnnxInference

@onnxnumpy_np(signature=NDArrayType(("T:all", ), dtypes_out=('T',)))
def onnx_rfft(x, fft_length=None):
    if fft_length is None:
        raise RuntimeError("fft_length must be specified.")

    size = fft_length // 2 + 1
    cst = dft_real_cst(fft_length, fft_length).astype(numpy.float32)
    xt = npnx.transpose(x, (1, 0))
    res = npnx.matmul(cst[:, :, :fft_length], xt[:fft_length])[:, :size, :]
    return npnx.transpose(res, (0, 2, 1))

fft_onx = onnx_rfft(rnd, fft_length=rnd.shape[1])
fft_onx
```

[8]:
```
array([[[-0.33227617, -1.5372959 ,  4.4797373 ,  1.3639294 ],
        [ 0.07591468,  0.51947707, -2.4824262 , -0.98728204],
        [-0.5587506 , -0.8322822 ,  0.48281363, -0.86559296],
        [ 0.26185107, -1.1814368 ,  0.96002394,  3.5386157 ],
        [ 1.0627685 ,  3.0725865 , -0.8257934 ,  4.108111  ]],

       [[ 0.        , -0.93413043,  2.890194  , -2.5913336 ],
        [ 0.        ,  0.624144  , -1.5657941 ,  2.8143494 ],
        [ 0.        ,  2.2525156 ,  2.6933842 ,  0.08437189],
        [ 0.        ,  1.7362347 ,  0.39340976, -1.3285824 ],
        [ 0.        , -2.7150555 , -1.9185277 , -0.4683669 ]]],
      dtype=float32)
```

[9]:
```
almost_equal(fft_cus, fft_onx)
```

The corresponding ONNX graph is the following:

[10]:
```
%onnxview onnx_rfft.to_onnx()
```

[10]:
```
<jyquickhelper.jspy.render_nb_js_dot.RenderJsDot at 0x25b38b3f9d0>
```

[11]:
```
fft_onx3 = onnx_rfft(rnd, fft_length=3)
almost_equal(fft_cus3, fft_onx3)
```

1.3 FFT 2D

Below the code for complex features.

[12]:
```
def _DFT_cst(N, fft_length, trunc=True):
    n = numpy.arange(N)
    k = n.reshape((N, 1)).astype(numpy.float64)
    M = numpy.exp(-2j * numpy.pi * k * n / fft_length)
    return M[:fft_length // 2 + 1] if trunc else M

def DFT(x, fft_length=None, axis=1):
    if axis == 1:
        x = x.T
    if fft_length is None:
        fft_length = x.shape[0]
    cst = _DFT_cst(x.shape[0], fft_length, trunc=axis==1)
    if axis == 1:
        return numpy.matmul(cst, x).T
    return numpy.matmul(cst, x)

def fft2d_(mat, fft_length):
    mat = mat[:fft_length[0], :fft_length[1]]
    res = mat.copy()
    res = DFT(res, fft_length[1], axis=1)
    res = DFT(res, fft_length[0], axis=0)
    return res[:fft_length[0], :fft_length[1]//2 + 1]


rnd = numpy.random.randn(5, 7).astype(numpy.float32)
fft2d_np_ = fft2d_(rnd, rnd.shape)
fft2d_np = numpy.fft.rfft2(rnd)
fft2d_np_
```

[12]:
```
array([[-4.14039719 +0.j        , -1.06715605 +1.16770652j,
         -0.27080808 +1.93562775j,  5.28785846 +2.27915445j],
        [-2.57576449 +3.09907081j, -8.90391777 -5.56953367j,
         -1.6455202  +2.03337471j,  4.21121677 -1.85803104j],
        [ 1.84529583 -0.54705419j,  3.61232172 -4.11661604j,
          1.00659205 +3.72264071j, -0.36878039 -8.21956881j],
        [ 1.84529583 +0.54705419j, -1.173484   +5.12345283j,
         -1.7897386 -10.15322422j, -0.17258219 +2.37388952j],
        [-2.57576449 -3.09907081j,  0.58355627 +1.62293628j,
          0.71779814 +4.64582025j, -6.32441255 -4.21906685j]])
```

[13]:
```
almost_equal(fft2d_np_, fft2d_np)
```

It implies the computation of two FFT 1D along both axes. However, as ONNX does not support complex, it needs to be rewritten with only real numbers. The algorithm can be summarized into this formula $FFT(FFT(x, axis = 1), axis = 0)$. If x is real, $FFT(x, .)$ is complex. We still assume x is real, it then becomes (FFT is a linear operator, so $FFT(ix) = iFFT(x)$):

- $y = FFT(x, axis = 1)$
- $z_r = FFT(Real(y), axis = 0)$, $z_i = FFT(Imag(y), axis = 0)$
- $z = z_r + iz_i$

z is the desired output. The following implementation is probably not the most efficient one. It avoids inplace computation as ONNX does like that.

[14]:
```
def fft2d(mat, fft_length):
    mat = mat[:fft_length[0], :fft_length[1]]
    res = mat.copy()

    # first FFT
    res = dft_real(res, fft_length=fft_length[1], transpose=True)

    # second FFT decomposed on FFT on real part and imaginary part
    res2_real = dft_real(res[0], fft_length=fft_length[0], transpose=False)
    res2_imag = dft_real(res[1], fft_length=fft_length[0], transpose=False)
    res2_imag2 = numpy.vstack([-res2_imag[1:2], res2_imag[:1]])
    res = res2_real + res2_imag2
    size = fft_length[1]//2 + 1
    return res[:, :fft_length[0], :size]


fft2d_np = numpy.fft.rfft2(rnd)
fft2d_cus = fft2d(rnd, rnd.shape)
almost_equal(fft2d_np, fft2d_cus)
```

[15]:
```
fft2d_np
```

[15]:
```
array([[-4.14039719 +0.j        , -1.06715605 +1.16770652j,
        -0.27080808 +1.93562775j,  5.28785846 +2.27915445j],
       [-2.57576449 +3.09907081j, -8.90391777 -5.56953367j,
        -1.6455202  +2.03337471j,  4.21121677 -1.85803104j],
       [ 1.84529583 -0.54705419j,  3.61232172 -4.11661604j,
         1.00659205 +3.72264071j, -0.36878039 -8.21956881j],
       [ 1.84529583 +0.54705419j, -1.173484   +5.12345283j,
        -1.7897386 -10.15322422j, -0.17258219 +2.37388952j],
       [-2.57576449 -3.09907081j,  0.58355627 +1.62293628j,
         0.71779814 +4.64582025j, -6.32441255 -4.21906685j]])
```

[16]:
```
fft2d_cus
```

[16]:
```
array([[[ -4.14039719,  -1.06715605,  -0.27080808,   5.28785846],
        [ -2.57576449,  -8.90391777,  -1.6455202 ,   4.21121677],
        [  1.84529583,   3.61232172,   1.00659205,  -0.36878039],
        [  1.84529583,  -1.173484  ,  -1.7897386 ,  -0.17258219],
        [ -2.57576449,   0.58355627,   0.71779814,  -6.32441255]],
```

```
[[  0.        ,   1.16770652,   1.93562775,   2.27915445],
 [  3.09907081,  -5.56953367,   2.03337471,  -1.85803104],
 [ -0.54705419,  -4.11661604,   3.72264071,  -8.21956881],
 [  0.54705419,   5.12345283, -10.15322422,   2.37388952],
 [ -3.09907081,   1.62293628,   4.64582025,  -4.21906685]]])
```

And with a different `fft_length`.

[17]:
```
fft2d_np = numpy.fft.rfft2(rnd, (4, 6))
fft2d_cus = fft2d(rnd, (4, 6))
almost_equal(fft2d_np[:4, :], fft2d_cus)
```

1.4 FFT 2D in ONNX

We use again the numpy API for ONNX.

[18]:
```
def onnx_rfft_1d(x, fft_length=None, transpose=True):
    if fft_length is None:
        raise RuntimeError("fft_length must be specified.")

    size = fft_length // 2 + 1
    cst = dft_real_cst(fft_length, fft_length).astype(numpy.float32)
    if transpose:
        xt = npnx.transpose(x, (1, 0))
        res = npnx.matmul(cst[:, :, :fft_length], xt[:fft_length])[:, :size, :]
        return npnx.transpose(res, (0, 2, 1))
    else:
        return npnx.matmul(cst[:, :, :fft_length], x[:fft_length])


@onnxnumpy_np(signature=NDArrayType(("T:all", ), dtypes_out=('T',)))
def onnx_rfft_2d(x, fft_length=None):
    mat = x[:fft_length[0], :fft_length[1]]

    # first FFT
    res = onnx_rfft_1d(mat, fft_length=fft_length[1], transpose=True)

    # second FFT decomposed on FFT on real part and imaginary part
    res2_real = onnx_rfft_1d(res[0], fft_length=fft_length[0], transpose=False)
    res2_imag = onnx_rfft_1d(res[1], fft_length=fft_length[0], transpose=False)
    res2_imag2 = npnx.vstack(-res2_imag[1:2], res2_imag[:1])
    res = res2_real + res2_imag2
    size = fft_length[1]//2 + 1
    return res[:, :fft_length[0], :size]


fft2d_cus = fft2d(rnd, rnd.shape)
fft2d_onx = onnx_rfft_2d(rnd, fft_length=rnd.shape)
almost_equal(fft2d_cus, fft2d_onx)
```

The corresponding ONNX graph.

[19]:
```
%onnxview onnx_rfft_2d.to_onnx()
```

[19]:
```
<jyquickhelper.jspy.render_nb_js_dot.RenderJsDot at 0x25b44365b50>
```

[20]:
```
with open("fft2d.onnx", "wb") as f:
    f.write(onnx_rfft_2d.to_onnx().SerializeToString())
```

With a different `fft_length`.

[21]:
```
fft2d_cus = fft2d(rnd, (4, 5))
fft2d_onx = onnx_rfft_2d(rnd, fft_length=(4, 5))
almost_equal(fft2d_cus, fft2d_onx)
```

This implementation of FFT in ONNX assumes shapes and fft lengths are constant. Otherwise, the matrix returned by function `dft_real_cst` must be converted as well. That's left as an exercise.

1.5 FFT2D with shape (3,1,4)

Previous implementation expects the input matrix to have two dimensions. It fails with 3.

[22]:
```
shape = (3, 1, 4)
fft_length = (1, 4)
rnd = numpy.random.randn(*list(shape)).astype(numpy.float32)
fft2d_numpy = numpy.fft.fft2(rnd, fft_length)
fft2d_numpy.shape
```

[22]:
```
(3, 1, 4)
```

[23]:
```
fft2d_numpy
```

[23]:
```
array([[[-1.04513007+0.j        ,  0.7261328 -0.1488841j ,
         -0.76143177+0.j        ,  0.7261328 +0.1488841j ]],

       [[ 0.13626025+0.j        , -0.37364573+0.49485394j,
         -0.5746009 +0.j        , -0.37364573-0.49485394j]],

       [[ 1.52022177+0.j        ,  0.35786384+1.09477997j,
          2.16783673+0.j        ,  0.35786384-1.09477997j]]])
```

[24]:
```
try:
    fft2d_cus = fft2d(rnd, fft_length)
except Exception as e:
    print(e)
# fft2d_onx = onnx_rfft_2d(rnd, fft_length=fft_length)
```

```
axes don't match array
```

1.5.1 numpy version

Let's do it again with numpy first. fft2 performs `fft2` on the last two axis as many times as the first axis. The goal is still to have an implementation which works for any dimension.

[25]:
```
conc = []
for i in range(rnd.shape[0]):
    f2 = fft2d(rnd[i], fft_length)
    conc.append(numpy.expand_dims(f2, 0))
res = numpy.vstack(conc).transpose(1, 0, 2, 3)
almost_equal(fft2d_numpy[:, :, :3], res)
```

It works. And now a more efficient implementation. It is better to read matmul description before. To summarize, a third axis is equivalent to many matrix multiplications over the last two axes, as many as the dimension of the first axis: `matmul(A[I,J,K], B[I,K,L]) --> C[I,J,L]`. Broadcasting also works… `matmul(A[1,J,K], B[I,K,L]) --> C[I,J,L]`.

[26]:
```
def dft_real_d3(x, fft_length=None, transpose=True):
    if len(x.shape) != 3:
        raise RuntimeError("Not implemented for shape=%r." % x.shape)
    N = x.shape[1]
    C = x.shape[-1] if transpose else x.shape[-2]
    if fft_length is None:
        fft_length = x.shape[-1]
    size = fft_length // 2 + 1

    cst = dft_real_cst(C, fft_length)
    if transpose:
        x = numpy.transpose(x, (0, 2, 1))
        a = cst[:, :, :fft_length]
        b = x[:, :fft_length, :]
        a = numpy.expand_dims(a, 0)
        b = numpy.expand_dims(b, 1)
        res = numpy.matmul(a, b)
        res = res[:, :, :size, :]
        return numpy.transpose(res, (1, 0, 3, 2))
    else:
        a = cst[:, :, :fft_length]
        b = x[:, :fft_length, :]
        a = numpy.expand_dims(a, 0)
        b = numpy.expand_dims(b, 1)
        res = numpy.matmul(a, b)
        return numpy.transpose(res, (1, 0, 2, 3))


def fft2d_d3(mat, fft_length):
    mat = mat[:, :fft_length[-2], :fft_length[-1]]
    res = mat.copy()

    # first FFT
    res = dft_real_d3(res, fft_length=fft_length[-1], transpose=True)

    # second FFT decomposed on FFT on real part and imaginary part
    res2_real = dft_real_d3(res[0], fft_length=fft_length[-2], transpose=False)
    res2_imag = dft_real_d3(res[1], fft_length=fft_length[-2], transpose=False)
    res2_imag2 = numpy.vstack([-res2_imag[1:2], res2_imag[:1]])
    res = res2_real + res2_imag2
    size = fft_length[-1]//2 + 1
    return res[:, :, :fft_length[-2], :size]


def fft2d_any(mat, fft_length):
    new_shape = (-1, ) + mat.shape[-2:]
    mat2 = mat.reshape(new_shape)
    f2 = fft2d_d3(mat2, fft_length)
    new_shape = (2, ) + mat.shape[:-2] + f2.shape[-2:]
```

```
    return f2.reshape(new_shape)


shape = (3, 1, 4)
fft_length = (1, 4)
rnd = numpy.random.randn(*list(shape)).astype(numpy.float32)
fft2d_numpy = numpy.fft.fft2(rnd, fft_length)
fft2d_cus = fft2d_any(rnd, fft_length)
almost_equal(fft2d_numpy[..., :3], fft2d_cus)
```

We check with more shapes to see if the implementation works for all of them.

[27]:
```
for shape in [(3, 1, 4), (5, 7), (3, 5, 7), (7, 5)]:
    for fft_length in [shape[-2:], (1, shape[-1]),
                       (min(2, shape[-2]), shape[-1]),
                       (shape[-2], 2),
                       (min(3, shape[-2]), min(4, shape[-2]))]:
        x  = numpy.random.randn(*list(shape)).astype(numpy.float32)
        fnp = numpy.fft.fft2(x, fft_length)
        if len(fnp.shape) == 2:
            fn= numpy.expand_dims(fnp, 0)
        try:
            cus = fft2d_any(x, fft_length)
        except IndexError as e:
            print("ERR x.shape=%r length=%r error=%r" % (x.shape, fft_length, e))
            continue
        try:
            almost_equal(fnp[..., :cus.shape[-1]], cus)
        except (AssertionError, IndexError) as e:
            print("DIS x.shape=%r length=%r error=%r  output shape=%r or %r" % (
                x.shape, fft_length, e, fnp.shape, cus.shape))
            continue
        print("OK  x.shape=%r length=%r output shape=%r or %r" % (
            x.shape, fft_length, fnp.shape, cus.shape))
```

```
OK  x.shape=(3, 1, 4) length=(1, 4) output shape=(3, 1, 4) or (2, 3, 1, 3)
OK  x.shape=(3, 1, 4) length=(1, 4) output shape=(3, 1, 4) or (2, 3, 1, 3)
OK  x.shape=(3, 1, 4) length=(1, 4) output shape=(3, 1, 4) or (2, 3, 1, 3)
OK  x.shape=(3, 1, 4) length=(1, 2) output shape=(3, 1, 2) or (2, 3, 1, 2)
OK  x.shape=(3, 1, 4) length=(1, 1) output shape=(3, 1, 1) or (2, 3, 1, 1)
OK  x.shape=(5, 7) length=(5, 7) output shape=(5, 7) or (2, 5, 4)
OK  x.shape=(5, 7) length=(1, 7) output shape=(1, 7) or (2, 1, 4)
OK  x.shape=(5, 7) length=(2, 7) output shape=(2, 7) or (2, 2, 4)
OK  x.shape=(5, 7) length=(5, 2) output shape=(5, 2) or (2, 5, 2)
OK  x.shape=(5, 7) length=(3, 4) output shape=(3, 4) or (2, 3, 3)
OK  x.shape=(3, 5, 7) length=(5, 7) output shape=(3, 5, 7) or (2, 3, 5, 4)
OK  x.shape=(3, 5, 7) length=(1, 7) output shape=(3, 1, 7) or (2, 3, 1, 4)
OK  x.shape=(3, 5, 7) length=(2, 7) output shape=(3, 2, 7) or (2, 3, 2, 4)
OK  x.shape=(3, 5, 7) length=(5, 2) output shape=(3, 5, 2) or (2, 3, 5, 2)
OK  x.shape=(3, 5, 7) length=(3, 4) output shape=(3, 3, 4) or (2, 3, 3, 3)
OK  x.shape=(7, 5) length=(7, 5) output shape=(7, 5) or (2, 7, 3)
OK  x.shape=(7, 5) length=(1, 5) output shape=(1, 5) or (2, 1, 3)
OK  x.shape=(7, 5) length=(2, 5) output shape=(2, 5) or (2, 2, 3)
OK  x.shape=(7, 5) length=(7, 2) output shape=(7, 2) or (2, 7, 2)
```

```
OK  x.shape=(7, 5) length=(3, 4) output shape=(3, 4) or (2, 3, 3)
```

1.5.2 ONNX version

Let's look into the differences first.

[28]:
```
%load_ext pyquickhelper
```

[29]:
```
%%html
<style>
table td, table th, table tr {text-align:left !important; white-space: pre;}
</style>
```

```
<IPython.core.display.HTML object>
```

[30]:
```
import inspect
text1 = inspect.getsource(dft_real)
text2 = inspect.getsource(dft_real_d3)
%codediff text1 text2 --verbose 1 --two 1
```

```
100%|¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ | 24/24 [00:00<00:00, 573.03it/s]
```

[30]:
```
<IPython.core.display.HTML object>
```

[31]:
```
text1 = inspect.getsource(fft2d)
text2 = inspect.getsource(fft2d_d3)
%codediff text1 text2 --verbose 1 --two 1
```

```
100%|¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ | 15/15 [00:00<00:00, 791.61it/s]
```

[31]:
```
<IPython.core.display.HTML object>
```

[32]:
```
def onnx_rfft_3d_1d(x, fft_length=None, transpose=True):
    if fft_length is None:
        raise RuntimeError("fft_length must be specified.")

    size = fft_length // 2 + 1
    cst = dft_real_cst(fft_length, fft_length).astype(numpy.float32)
    if transpose:
        xt = npnx.transpose(x, (0, 2, 1))
        a = cst[:, :, :fft_length]
        b = xt[:, :fft_length, :]
        a = npnx.expand_dims(a, 0)
        b = npnx.expand_dims(b, 1)
        res = npnx.matmul(a, b)
        res2 = res[:, :size, :]
        return npnx.transpose(res2, (1, 0, 3, 2))
    else:
        a = cst[:, :, :fft_length]
        b = x[:, :fft_length, :]
        a = npnx.expand_dims(a, 0)
        b = npnx.expand_dims(b, 1)
```

```
        res = npnx.matmul(a, b)
        return npnx.transpose(res, (1, 0, 2, 3))


def onnx_rfft_3d_2d(x, fft_length=None):
    mat = x[:, :fft_length[-2], :fft_length[-1]]

    # first FFT
    res = onnx_rfft_3d_1d(mat, fft_length=fft_length[-1], transpose=True)

    # second FFT decomposed on FFT on real part and imaginary part
    res2_real = onnx_rfft_3d_1d(res[0], fft_length=fft_length[0], transpose=False)
    res2_imag = onnx_rfft_3d_1d(res[1], fft_length=fft_length[0], transpose=False)
    res2_imag2 = npnx.vstack(-res2_imag[1:2], res2_imag[:1])
    res = res2_real + res2_imag2
    size = fft_length[1]//2 + 1
    return res[:, :, :fft_length[-2], :size]


@onnxnumpy_np(signature=NDArrayType(("T:all", ), dtypes_out=('T',)))
def onnx_rfft_2d_any(x, fft_length=None):
    new_shape = npnx.concat(
        numpy.array([-1], dtype=numpy.int64), x.shape[-2:], axis=0)
    mat2 = x.reshape(new_shape)
    f2 = onnx_rfft_3d_2d(mat2, fft_length)
    new_shape = npnx.concat(
        numpy.array([2], dtype=numpy.int64), x.shape[:-2], f2.shape[-2:])
    return f2.reshape(new_shape)


shape = (3, 1, 4)
fft_length = (1, 4)
rnd = numpy.random.randn(*list(shape)).astype(numpy.float32)
fft2d_cus = fft2d_any(rnd, fft_length)
fft2d_onx = onnx_rfft_2d_any(rnd, fft_length=fft_length)
almost_equal(fft2d_cus, fft2d_onx)
```

Let's do the same comparison.

[33]:
```
for shape in [(3, 1, 4), (5, 7), (3, 5, 7), (7, 5)]:
    for fft_length in [shape[-2:], (1, shape[-1]),
                       (min(2, shape[-2]), shape[-1]),
                       (shape[-2], 2),
                       (min(3, shape[-2]), min(4, shape[-2]))]:
        x  = numpy.random.randn(*list(shape)).astype(numpy.float32)
        if len(fnp.shape) == 2:
            fn= numpy.expand_dims(fnp, 0)
        try:
            cus = fft2d_any(x, fft_length)
        except IndexError as e:
            print("ERR x.shape=%r length=%r error=%r" % (x.shape, fft_length, e))
            continue
        try:
            onx = onnx_rfft_2d_any(x, fft_length=fft_length)
```

```
        except IndexError as e:
            print("ERR x.shape=%r length=%r error=%r" % (x.shape, fft_length, e))
            continue
        try:
            almost_equal(onx, cus)
        except (AssertionError, IndexError) as e:
            print("DIS x.shape=%r length=%r error=%r  output shape=%r or %r" % (
                x.shape, fft_length, e, fnp.shape, cus.shape))
            continue
        print("OK  x.shape=%r length=%r output shape=%r or %r" % (
            x.shape, fft_length, fnp.shape, cus.shape))
```

```
OK  x.shape=(3, 1, 4) length=(1, 4) output shape=(3, 4) or (2, 3, 1, 3)
OK  x.shape=(3, 1, 4) length=(1, 4) output shape=(3, 4) or (2, 3, 1, 3)
OK  x.shape=(3, 1, 4) length=(1, 4) output shape=(3, 4) or (2, 3, 1, 3)
OK  x.shape=(3, 1, 4) length=(1, 2) output shape=(3, 4) or (2, 3, 1, 2)
OK  x.shape=(3, 1, 4) length=(1, 1) output shape=(3, 4) or (2, 3, 1, 1)
OK  x.shape=(5, 7) length=(5, 7) output shape=(3, 4) or (2, 5, 4)
OK  x.shape=(5, 7) length=(1, 7) output shape=(3, 4) or (2, 1, 4)
OK  x.shape=(5, 7) length=(2, 7) output shape=(3, 4) or (2, 2, 4)
OK  x.shape=(5, 7) length=(5, 2) output shape=(3, 4) or (2, 5, 2)
OK  x.shape=(5, 7) length=(3, 4) output shape=(3, 4) or (2, 3, 3)
OK  x.shape=(3, 5, 7) length=(5, 7) output shape=(3, 4) or (2, 3, 5, 4)
OK  x.shape=(3, 5, 7) length=(1, 7) output shape=(3, 4) or (2, 3, 1, 4)
OK  x.shape=(3, 5, 7) length=(2, 7) output shape=(3, 4) or (2, 3, 2, 4)
OK  x.shape=(3, 5, 7) length=(5, 2) output shape=(3, 4) or (2, 3, 5, 2)
OK  x.shape=(3, 5, 7) length=(3, 4) output shape=(3, 4) or (2, 3, 3, 3)
OK  x.shape=(7, 5) length=(7, 5) output shape=(3, 4) or (2, 7, 3)
OK  x.shape=(7, 5) length=(1, 5) output shape=(3, 4) or (2, 1, 3)
OK  x.shape=(7, 5) length=(2, 5) output shape=(3, 4) or (2, 2, 3)
OK  x.shape=(7, 5) length=(7, 2) output shape=(3, 4) or (2, 7, 2)
OK  x.shape=(7, 5) length=(3, 4) output shape=(3, 4) or (2, 3, 3)
```

There is one issue with `fft_length=(1, 1)` but that case is out of scope.

1.5.3 ONNX graph

[34]:
```
key = list(onnx_rfft_2d_any.signed_compiled)[0]
%onnxview onnx_rfft_2d_any.signed_compiled[key].compiled.onnx_
```

[34]:
```
<jyquickhelper.jspy.render_nb_js_dot.RenderJsDot at 0x25b44556730>
```

[35]:
```
with open("fft2d_any.onnx", "wb") as f:
    key = list(onnx_rfft_2d_any.signed_compiled)[0]
    f.write(onnx_rfft_2d_any.signed_compiled[key].compiled.onnx_.SerializeToString())
```

Let's check the intermediate results.

[36]:
```
key = list(onnx_rfft_2d_any.signed_compiled)[0]
key
```

[36]:
```
FctVersion((numpy.float32,), ((1, 4),))
```

[37]:
```
from mlprodict.onnxrt import OnnxInference

x = numpy.random.randn(3, 1, 4).astype(numpy.float32)
onx = onnx_rfft_2d_any.signed_compiled[key].compiled.onnx_
oinf = OnnxInference(onx)
oinf.run({'x': x}, verbose=1, fLOG=print)
```

```
+ki='init': (1,) (dtype=int64 min=0 max=0)
+ki='init_1': (1,) (dtype=int64 min=-2 max=-2)
+ki='init_3': (1,) (dtype=int64 min=-1 max=-1)
+ki='init_4': (2,) (dtype=int64 min=0 max=0)
+ki='init_5': (2,) (dtype=int64 min=1 max=4)
+ki='init_6': (2,) (dtype=int64 min=1 max=2)
+ki='init_8': (1,) (dtype=int64 min=4 max=4)
+ki='init_9': (1,) (dtype=int64 min=1 max=1)
+ki='init_b11': (2, 4, 4) (dtype=float32 min=-1.0 max=1.0)
+ki='init_b14': (1,) (dtype=int64 min=3 max=3)
+ki='init_b16': () (dtype=int64 min=1 max=1)
+ki='init_b21': (2, 1, 1) (dtype=float32 min=0.0 max=1.0)
+ki='init_b23': () (dtype=int64 min=0 max=0)
+ki='init_b28': (1,) (dtype=int64 min=2 max=2)
+ki='init_b37': (2,) (dtype=int64 min=1 max=3)
+ki='init_b38': (2,) (dtype=int64 min=2 max=3)
-- OnnxInference: run 38 nodes
Onnx-Shape(x) -> out_sha_0    (name='_shape')
+kr='out_sha_0': (3,) (dtype=int64 min=1 max=4)
Onnx-Shape(out_sha_0) -> out_sha_0_1    (name='_shape_1')
+kr='out_sha_0_1': (1,) (dtype=int64 min=3 max=3)
Onnx-Gather(out_sha_0_1, init) -> out_gat_0    (name='_gather')
+kr='out_gat_0': (1,) (dtype=int64 min=3 max=3)
Onnx-Slice(out_sha_0, init_1, out_gat_0, init) -> out_sli_0    (name='_slice')
+kr='out_sli_0': (2,) (dtype=int64 min=1 max=4)
Onnx-Concat(init_3, out_sli_0) -> out_con_0    (name='_concat')
+kr='out_con_0': (3,) (dtype=int64 min=-1 max=4)
Onnx-Reshape(x, out_con_0) -> out_res_0    (name='_reshape')
+kr='out_res_0': (3, 1, 4) (dtype=float32 min=-2.0340726375579834
max=2.391742706298828)
Onnx-Slice(out_res_0, init_4, init_5, init_6) -> out_sli_0_1
(name='_slice_1')
+kr='out_sli_0_1': (3, 1, 4) (dtype=float32 min=-2.0340726375579834
max=2.391742706298828)
Onnx-Transpose(out_sli_0_1) -> out_tra_0    (name='_transpose')
+kr='out_tra_0': (3, 4, 1) (dtype=float32 min=-2.0340726375579834
max=2.391742706298828)
Onnx-Slice(out_tra_0, init, init_8, init_9) -> out_sli_0_2    (name='_slice_2')
+kr='out_sli_0_2': (3, 4, 1) (dtype=float32 min=-2.0340726375579834
max=2.391742706298828)
Onnx-Unsqueeze(out_sli_0_2, init_9) -> out_uns_0    (name='_unsqueeze')
+kr='out_uns_0': (3, 1, 4, 1) (dtype=float32 min=-2.0340726375579834
max=2.391742706298828)
Onnx-Unsqueeze(init_b11, init) -> out_uns_0_1    (name='_unsqueeze_1')
+kr='out_uns_0_1': (1, 2, 4, 4) (dtype=float32 min=-1.0 max=1.0)
Onnx-MatMul(out_uns_0_1, out_uns_0) -> out_mat_0    (name='_matmul')
+kr='out_mat_0': (3, 2, 4, 1) (dtype=float32 min=-2.188795566558838
```

```
max=3.3646905422210693)
Onnx-Slice(out_mat_0, init, init_b14, init_9) -> out_sli_0_3
(name='_slice_3')
+kr='out_sli_0_3': (3, 2, 4, 1) (dtype=float32 min=-2.188795566558838
max=3.3646905422210693)
Onnx-Transpose(out_sli_0_3) -> out_tra_0_1    (name='_transpose_1')
+kr='out_tra_0_1': (2, 3, 1, 4) (dtype=float32 min=-2.188795566558838
max=3.3646905422210693)
Onnx-Gather(out_tra_0_1, init_b16) -> out_gat_0_1    (name='_gather_1')
+kr='out_gat_0_1': (3, 1, 4) (dtype=float32 min=-2.054079532623291
max=2.054079532623291)
Onnx-Slice(out_gat_0_1, init, init_9, init_9) -> out_sli_0_4
(name='_slice_4')
+kr='out_sli_0_4': (3, 1, 4) (dtype=float32 min=-2.054079532623291
max=2.054079532623291)
Onnx-Unsqueeze(out_sli_0_4, init_9) -> out_uns_0_2    (name='_unsqueeze_2')
+kr='out_uns_0_2': (3, 1, 1, 4) (dtype=float32 min=-2.054079532623291
max=2.054079532623291)
Onnx-Unsqueeze(init_b21, init) -> out_uns_0_3    (name='_unsqueeze_3')
+kr='out_uns_0_3': (1, 2, 1, 1) (dtype=float32 min=0.0 max=1.0)
Onnx-MatMul(out_uns_0_3, out_uns_0_2) -> out_mat_0_1    (name='_matmul_1')
+kr='out_mat_0_1': (3, 2, 1, 4) (dtype=float32 min=-2.054079532623291
max=2.054079532623291)
Onnx-Gather(out_tra_0_1, init_b23) -> out_gat_0_2    (name='_gather_2')
+kr='out_gat_0_2': (3, 1, 4) (dtype=float32 min=-2.188795566558838
max=3.3646905422210693)
Onnx-Transpose(out_mat_0_1) -> out_tra_0_2    (name='_transpose_2')
+kr='out_tra_0_2': (2, 3, 1, 4) (dtype=float32 min=-2.054079532623291
max=2.054079532623291)
Onnx-Slice(out_gat_0_2, init, init_9, init_9) -> out_sli_0_5
(name='_slice_5')
+kr='out_sli_0_5': (3, 1, 4) (dtype=float32 min=-2.188795566558838
max=3.3646905422210693)
Onnx-Slice(out_tra_0_2, init_9, init_b28, init) -> out_sli_0_6
(name='_slice_6')
+kr='out_sli_0_6': (1, 3, 1, 4) (dtype=float32 min=0.0 max=0.0)
Onnx-Unsqueeze(out_sli_0_5, init_9) -> out_uns_0_4    (name='_unsqueeze_4')
+kr='out_uns_0_4': (3, 1, 1, 4) (dtype=float32 min=-2.188795566558838
max=3.3646905422210693)
Onnx-Slice(out_tra_0_2, init, init_9, init) -> out_sli_0_7    (name='_slice_7')
+kr='out_sli_0_7': (1, 3, 1, 4) (dtype=float32 min=-2.054079532623291
max=2.054079532623291)
Onnx-Neg(out_sli_0_6) -> out_neg_0    (name='_neg')
+kr='out_neg_0': (1, 3, 1, 4) (dtype=float32 min=-0.0 max=-0.0)
Onnx-MatMul(out_uns_0_3, out_uns_0_4) -> out_mat_0_2    (name='_matmul_2')
+kr='out_mat_0_2': (3, 2, 1, 4) (dtype=float32 min=-2.188795566558838
max=3.3646905422210693)
Onnx-Concat(out_neg_0, out_sli_0_7) -> out_con_0_1    (name='_concat_1')
+kr='out_con_0_1': (2, 3, 1, 4) (dtype=float32 min=-2.054079532623291
max=2.054079532623291)
Onnx-Transpose(out_mat_0_2) -> out_tra_0_3    (name='_transpose_3')
+kr='out_tra_0_3': (2, 3, 1, 4) (dtype=float32 min=-2.188795566558838
max=3.3646905422210693)
Onnx-Add(out_tra_0_3, out_con_0_1) -> out_add_0    (name='_add')
```

```
+kr='out_add_0': (2, 3, 1, 4) (dtype=float32 min=-2.188795566558838
max=3.3646905422210693)
Onnx-Slice(out_add_0, init_4, init_b37, init_b38) -> out_sli_0_8
(name='_slice_8')
+kr='out_sli_0_8': (2, 3, 1, 3) (dtype=float32 min=-2.188795566558838
max=3.3646905422210693)
Onnx-Shape(out_sli_0_8) -> out_sha_0_2    (name='_shape_2')
+kr='out_sha_0_2': (4,) (dtype=int64 min=1 max=3)
Onnx-Shape(out_sha_0_2) -> out_sha_0_3    (name='_shape_3')
+kr='out_sha_0_3': (1,) (dtype=int64 min=4 max=4)
Onnx-Gather(out_sha_0_3, init) -> out_gat_0_3    (name='_gather_3')
+kr='out_gat_0_3': (1,) (dtype=int64 min=4 max=4)
Onnx-Slice(out_sha_0_2, init_1, out_gat_0_3, init) -> out_sli_0_9
(name='_slice_9')
+kr='out_sli_0_9': (2,) (dtype=int64 min=1 max=3)
Onnx-Slice(out_sha_0, init, init_1, init) -> out_sli_0_b10
(name='_slice_b10')
+kr='out_sli_0_b10': (1,) (dtype=int64 min=3 max=3)
Onnx-Concat(init_b28, out_sli_0_b10, out_sli_0_9) -> out_con_0_2
(name='_concat_2')
+kr='out_con_0_2': (4,) (dtype=int64 min=1 max=3)
Onnx-Reshape(out_sli_0_8, out_con_0_2) -> y    (name='_reshape_1')
+kr='y': (2, 3, 1, 3) (dtype=float32 min=-2.188795566558838
max=3.3646905422210693)
```

[37]:
```
{'y': array([[[[-8.3439898e-01,  6.9026375e-01,  3.2907667e+00]],

              [[ 3.3646905e+00, -2.9031307e-01, -2.0941215e+00]],

              [[ 2.1246734e+00,  5.1293659e-01, -2.1887956e+00]]],


             [[[ 0.0000000e+00, -2.0055625e+00,  8.1667386e-16]],

              [[ 0.0000000e+00,  2.0540795e+00, -8.0671079e-16]],

              [[ 0.0000000e+00, -3.2617974e-01, -5.5504507e-16]]]],
            dtype=float32)}
```

[38]: