Building Multi-threaded Custom Data Pipelines for Tensorflow v1.12+

Nima Tajbakhsh and Jeff Chiang

I. Introduction

You are dealing with problems where data, fortunately or unfortunately, is not available in the typical 2D image format. For instance, your data may consist of 3D medical scans, which are available in DICOM format. As such, you cannot benefit from the standard Tensorflow data i/o functions (e.g. from_csv, from_pandas, etc) or data pre-processing functions (e.g. in tf.image) available for regular 2D images. Facing all these limitations, you have written your own customized, super fancy data pipeline in python, which interfaces with your model through the (soon obsolete) feed_dict mechanism. Your data pipeline works just fine, but you have always seen sub-optimal GPU usage when running `nvidia-smi`, thinking to yourself the data pipeline might not be optimized.

You have now heard that Tensorflow 2.0 is coming, and with it a potentially massive refactoring headache. Among many suggested changes is the following: "use tf.data datasets for data input". You are so tempted to give it a try, seeing for yourself whether your data pipeline can get any further optimized. Having tons of respect for yourself as an independent problem-solver, you opened the Tensorflow documentation, passionately scrolled up and down the pages for hours, and soon become frustrated from being unable to make sense of this supposedly comprehensive documentation. You think to yourself, "How come I cannot make much sense out of it?! Have I lost my edge or is this documentation just one of a kind? Sigh! Only if I could have some proper pointers on how to use this beautiful module of Tensorflow." Well, if this is you, stop fretting about it, because you have just found what you were looking for: a simple, easy-to-follow tutorial on how to make a fast, multi-threaded data pipeline based off of a python generator. And the best part is, what you learn here will be applicable to both TF1.12+ and TF 2.0.

II. TL; DR

1) Pre-processing operations in pure Python (or numpy, etc) can be integrated with tf.data using from_generator
2) Blindly wrapping your generator using from_generator is single-threaded, and suboptimal. Multi-threading exists in tf.data.
3) Additional speed-up can be gained by using prefetch() and separating time-consuming operations
4) Source codes for this tutorial are available here.

III. Structure of this tutorial

This tutorial explains tf.data through three easy-to-understand examples. Example 1 is a must-read, walking you through the most important concepts and pitfalls when using tf.data. You will learn how to write single-threaded and multi-threaded data pipelines as well as how to use pre-fetching to turn undesirable periodic delays into opportunities to keep your data queue full. In Example 2, you will learn how to set up your data pipeline using a python class and how to avoid some common pitfalls. Example 3 teaches you how you can break a data pipeline into two smaller pipelines, further accelerating the data preparation process for your models.

IV. Example 1

We have a python-based data generator, which takes 0.3 seconds to prepare data (for example, loading a large numpy array from disk and applying some data augmentation) and then we have this sophisticated TF operation (for example, a training step for an object detection model), which processes the input data in 0.1 seconds. We can simulate this scenario with the following data generator and TF operation:

```
import tensorflow as tf
from time import sleep
from time import time

# data generator
def py_gen(gen_name):
    gen_name = gen_name.decode('utf-8')
    for num in range(10):
        sleep(0.3)
        yield '{} yields {}'.format(gen_name, num)


# model operation
def model(data):
    sleep(0.1)
```

In this example, the generator takes as the input a name and then yields a string at each iteration. A pause of 0.3 seconds is included in the body of the generator to simulate a set of time-consuming operations, while the sophisticated TF operation is simulated as a pause of 0.1 seconds.

We start this example by writing a single-threaded data pipeline using tf.data. We then show how adding a dedicated data thread and using a prefetch mechanism can optimize this pipeline. Finally, we go over a multi-threaded solution to this example.

A. Single-thread

Using from_generator, we can convert our generator to a basic data pipeline.

```
Dataset = tf.data.Dataset
name = 'Gen_0'
ds = Dataset.from_generator(py_gen,
                            output_types=(tf.string),
                            args=(name,))
data_tf = ds.make_one_shot_iterator().get_next()
```

We have simply defined a name for our generator and then passed the data generator, `py_gen`, to the Dataset object of

TF, which uses a data queue in its back-end. We then create an iterator that runs through the queue. Each time the iterator moves forward, a string is yielded.

Note: when passing the python generator to the Dataset object, you must specify the types of the tensors that your python generator returns. In this example, we return a string; thus, we have specified tf.string as the output type.

Note: we need to use "args" to pass the input arguments (if any) to our python generator. Beware that the code will crash if we omit the comma in the tuple provided for "args".

Note: Tensorflow returns string objects that are encoded as Bytes, so we need to decode them into a string. This is done using decode('utf-8').

Note: We know that TF 2.0 discourages the use of tf.Session(), but the timing differences are the same in both TF1.12+ and TF 2.0.

Let's run the session and see the output:

```
def run_session(data_tf):
    with tf.Session() as sess:
        while True:
            try:
                t1 = time()
                data_py = sess.run(data_tf)
                t2 = time()
                t = t2 - t1
                model(data_tf)
                msg = 'elapsed time: {:.3f}, {}'.format(t, data_py.decode('utf-8'))
                print(msg)
            except tf.errors.OutOfRangeError:
                print('data generator(s) are exhausted')
                break
```

```
"""
output:
    elapsed time: 0.320, Gen_0 yields 0
    elapsed time: 0.302, Gen_0 yields 1
    elapsed time: 0.302, Gen_0 yields 2
    elapsed time: 0.302, Gen_0 yields 3
    elapsed time: 0.302, Gen_0 yields 4
"""
```

We've converted our data pipeline to use tf.data, but wait a minute! In this example, the model operation takes 0.1 seconds to finish, so if the data generator were working in the background, it would take only 0.2 (=0.3-0.1) seconds to generate the next input. This observation suggests that the data generator does not take advantage of 0.1-second delay caused by the model. In other words, we could achieve faster performance if we loaded the next data while we perform the model operations on the current data. In fact, in this case the same thread prepares the data and also runs the TF operation. Thus, the preparation of the next input is postponed until after the TF operation has ended. If the data generator owned its own thread, the data preparation and TF operation could run simultaneously, reducing the wait time for the TF operation from 0.3 to 0.2 seconds. So let's assign the data generator a dedicated thread. Actually, the fix is quite simple:

```
Dataset = tf.data.Dataset
name = 'Gen_0'
ds = Dataset.from_generator(py_gen,
                            output_types=(tf.string),
                            args=(name,))
ds = ds.map(lambda x: x, num_parallel_calls=1)
data_tf = ds.make_one_shot_iterator().get_next()
```

The difference with the previous solution is in Line 6, where I apply an identity transformation to each element in the dataset. Of course, it is not the identity transformation that does the job, rather, `num_parallel_calls=1`, which assigns one dedicated thread to the data generator. To be frank, we are somewhat abusing the parallel feature of the map function, but it serves the purpose. Let's see the output:

Note: By default, `num_parallel_calls` is set to 0, instructing the data pipeline to use the existing thread. `num_parallel_calls=1` actually instructs tf.data to create a new thread for this operation.

```
"""
output:
    elapsed time: 0.322, Gen_0 yields 0
    elapsed time: 0.201, Gen_0 yields 1
    elapsed time: 0.201, Gen_0 yields 2
    elapsed time: 0.201, Gen_0 yields 3
    elapsed time: 0.201, Gen_0 yields 4
"""
```

The TF operation now has to wait only 0.2 seconds for the input data, which is what we wanted to achieve. Of course, the wait time is still 0.3 seconds for the very first input data, because the TF operation has not been executed yet; and thus, the 0.1-second delay cannot be taken advantage of.

Before we move on to a multi-threaded solution, let's inject a one-time delay to our TF session. As seen below, we now have a delay of 3 seconds at the onset of iteration 4, which gives the dedicated data thread ample amount of time to create the next 10 input data for the model operation. Recall that the data thread needs 0.3 seconds to prepare each input data.

```
def run_session_w_delay(data_tf):
    with tf.Session() as sess:
        counter = 1
        while True:
            try:
                if counter == 4:
                    sleep(3)
                t1 = time()
                data_py = sess.run(data_tf)
                t2 = time()
                t = t2 - t1
                model(data_tf)
                msg = 'elapsed time: {:.3f}, {}'.format(t, data_py.decode('utf-8'))
                print(msg)
                counter += 1
            except tf.errors.OutOfRangeError:
                print('data generator(s) are exhausted')
                break
```

Let's run the session and see the output:

```
"""
output without pre-fetching:
    elapsed time: 0.322, Gen_0 yields 0
    elapsed time: 0.201, Gen_0 yields 1
    elapsed time: 0.201, Gen_0 yields 2
    elapsed time: 0.001, Gen_0 yields 3
    elapsed time: 0.201, Gen_0 yields 4
    elapsed time: 0.201, Gen_0 yields 5
    elapsed time: 0.201, Gen_0 yields 6
    elapsed time: 0.201, Gen_0 yields 7
    elapsed time: 0.201, Gen_0 yields 8
    elapsed time: 0.201, Gen_0 yields 9
"""
```

Iterations 1 to 3 are just the same as before. As expected, iteration 4 has no delay thanks to the 3-second delay that we injected. However, surprisingly, we go back to having a delay of 0.2 seconds for iteration 5 and onward! This is counter-intuitive because the data thread has had enough time to prepare the next 10 input data, then why has it created only the next input?! Well, this may be a design consideration to keep memory usage limited especially when the input data is memory-intensive (let's say a 3D scan). So, what's the solution? Easy! Just have the data thread pre-fetch the input data, which can be done by adding one line of code, as follows:

```
ds = Dataset.from_generator(py_gen,
                            output_types=(tf.string),
                            args=(name,))
ds = ds.map(lambda x: x, num_parallel_calls=1)
ds = ds.prefetch(buffer_size=10)
data_tf = ds.make_one_shot_iterator().get_next()
```

As seen in Line 5, the data thread is now instructed to prepare and hold 10 input data in the queue. If we run the code once again, we see no delay after iteration 4.

```
"""
output with pre-fetching:
```

```
    elapsed time: 0.324, Gen_0 yields 0
    elapsed time: 0.201, Gen_0 yields 1
    elapsed time: 0.201, Gen_0 yields 2
    elapsed time: 0.001, Gen_0 yields 3
    elapsed time: 0.001, Gen_0 yields 4
    elapsed time: 0.001, Gen_0 yields 5
    elapsed time: 0.001, Gen_0 yields 6
    elapsed time: 0.001, Gen_0 yields 7
    elapsed time: 0.001, Gen_0 yields 8
    elapsed time: 0.001, Gen_0 yields 9
"""
```

Of course, this no-delay behaviour won't last long, because the model runs faster than the data generator in this example. So, after a certain number of iterations, we once again observe a delay of 0.2 seconds prior to each model operation.

You may be wondering what the point of using data pre-fetching is, if it is only going to be effective for a few iterations? In my opinion, data pre-fetching begins to shine when we have a periodic delay in our TF session, as opposed to a 1-time delay in this example. A periodic delay could be some post-processing or data IO that can take place after every several iterations.

B. multi-thread

A common approach to creating a multi-threaded pipeline is to instantiate several "replicas" of the data generator and assign them to different threads. It turns out the the `interleave` function of TF Dataset mimics this behavior. We can use it to create several instances of our data generator, allowing us to pull data from all instances in parallel, and then interleave the pulled data into the final data queue. To take advantage of `interleave`, you only need to change the following lines of code:

```
Dataset = tf.data.Dataset
ds = Dataset.from_tensor_slices(['Gen_0', 'Gen_1', 'Gen_2'])
ds = ds.interleave(lambda x: Dataset.from_generator(py_gen, output_types=(tf.string
    ), args=(x,)),
                   cycle_length=3,
                   block_length=1,
                   num_parallel_calls=3)
data_tf = ds.make_one_shot_iterator().get_next()
```

The multi-threaded pipeline removes the wait time for the TF operation. Here is the output:

```
"""
output:
    time lapsed: 0.327, Gen_0 yields 0
    time lapsed: 0.001, Gen_1 yields 0
    time lapsed: 0.001, Gen_2 yields 0
    time lapsed: 0.001, Gen_0 yields 1
    time lapsed: 0.001, Gen_1 yields 1
    time lapsed: 0.001, Gen_2 yields 1
    time lapsed: 0.001, Gen_0 yields 2
    time lapsed: 0.001, Gen_1 yields 2
    time lapsed: 0.001, Gen_2 yields 2
    time lapsed: 0.001, Gen_0 yields 3
    time lapsed: 0.001, Gen_1 yields 3
    time lapsed: 0.001, Gen_2 yields 3
    time lapsed: 0.001, Gen_0 yields 4
    time lapsed: 0.001, Gen_1 yields 4
    time lapsed: 0.001, Gen_2 yields 4
"""
```

You might be scratching your head thinking what the hell just happened. I agree—these few lines of code are a little too hard to digest particularly for someone who is new to the Dataset module of TF. Well, I presented the solution right away to make it easy for those of you who are here looking for a quick answer. We now go through the code above line-by-line.

It all starts with creating a base dataset that contains, in each element, the arguments needed to initialize our data generator. Here, I intend to have 3 instances of data generator and each generator needs a mandatory string argument. Hence, I have used `from_tensor_slices` to generate a dataset with 3 elements where each element is a string. Obviously, if the generator needed 2 input arguments, I would have created a dataset where each element is a tuple with 2 values. Once the base dataset is constructed, I call the `interleave` method, which applies a same function to each element in the dataset. As you can guess this function is nothing but the same old `from_generator`, which takes as input the current string element, `x`, and then creates a new dataset object via our `py_gen`. By using `from_tensor_slices` and `interleave` methods, I have asked TF to generate 3 datasets each fed by 1 instance of our data generator.

Now, let's merge these 3 datasets into 1 final dataset. Fortunately, we do not need to write extra codes for this purpose, as the `interleave` method accepts two additional arguments called `cycle_length` and `block_length`, which allow us to combine the resulting datasets in a flexible manner:

1) if `cycle_length` is set to 1, the merged dataset begins with elements of dataset 1, and then elements of dataset 2, and then elements of dataset 3.
2) if `cycle_length` is set to 2, then the merged dataset will have `block_length` elements of dataset 1, followed by `block_length` of dataset 2, and then `block_length` elements of dataset 1, followed by `block_length` of dataset 2, and it continues until datasets 1 and 2 are exhausted. Next, elements of dataset 3 are appended to the merged dataset.
3) if `cycle_length` is set to 3, then the merged dataset will have `block_length` elements of dataset 1, followed by `block_length` of dataset 2, followed by `block_length` elements of dataset 3, and then back to dataset 1 and this loop continues until all 3 dataset are exhausted.

So, you are thinking how to set these 2 parameters? Easy! Set the `cycle_length` equal to the number of datasets you have created and set `block_length` to 1. By doing so, dataset 1 will have ample amount of time to prepare the next element while our TF operation consumes elements from datasets 2 and 3. One last thing, TF by default assigns the same thread to all 3 datasets, meaning no performance improvement whatsoever. You need to manually set `num_parallel_calls` to 3, which assigns a separate thread to each instance of our data generator, resulting in a true multi-threaded data pipeline.

Note: We discussed the necessity of having a base dataset as a means of providing the input arguments for multiple instances of our data generator. In the example above, the base datasets had 3 strings in it. But what if the data data generator needs no input arguments? Well, we still need to have a base dummy dataset with as many elements as the number of generator instances we wish to create. The content of each element does not matter though—it can be an int, float or string. Here is an example where the data generator needs no input argument, but we still need to use a base dummy dataset:

```
# data generator
def py_gen():
    for num in range(5):
        sleep(0.3)
        yield '{} yields {}'.format('no name', num)
```

```
# set up tf generator
Dataset = tf.data.Dataset
ds = Dataset.from_tensor_slices(['xxx', 'xxx', 'xxx'])
ds = ds.interleave(lambda x: Dataset.from_generator(py_gen, output_types=(tf.string
    )),
```

```
                    cycle_length=3,
                    block_length=1,
                    num_parallel_calls=3)
data_tf = ds.make_one_shot_iterator().get_next()
```

V. Example 2

It is often desirable to have the data generator as an instance method of a python class. By doing so, different instances of our data generators can have access to class-level attributes. In fact, instead of passing a bunch of common arguments to your generators, you can define the arguments as class attributes, and let the generator instances access them during the run-time. For instance, I have put together a data generator class that allows me to create the generator name on-the-fly by incrementally updating a `counter` variable, which I have defined as a class attribute as follows:

```
class DataGen():
    counter = 0

    def __init__(self):
        self.gen_num = DataGen.counter
        DataGen.counter += 1

    def py_gen(self, gen_name):
        gen_name = gen_name.decode('utf8') + '_' + str(self.gen_num)
        for num in range(5):
            sleep(0.3)
            yield '{} yields {}'.format(gen_name, num)
```

Since I have changed the interface to our data generator, I also need to change what I pass to `from_generator` as follows:

```
Dataset = tf.data.Dataset
dummy_ds = Dataset.from_tensor_slices(['Gen', 'Gen', 'Gen'])
dummy_ds = dummy_ds.interleave(lambda x: Dataset.from_generator(DataGen().py_gen,
      output_types=(tf.string), args=(x,)),
                               cycle_length=3,
                               block_length=1,
                               num_parallel_calls=3)
```

Now the desired output should look like “Gen_0 yields ...” followed by “Gen_1 yields ...” and so on, but let’s run the code and see the output:

```
"""
output:
    elapsed time: 0.329, Gen_0 yields 0
    elapsed time: 0.001, Gen_0 yields 0
    elapsed time: 0.001, Gen_0 yields 0
    elapsed time: 0.001, Gen_0 yields 1
    elapsed time: 0.001, Gen_0 yields 1
    elapsed time: 0.001, Gen_0 yields 1
    elapsed time: 0.001, Gen_0 yields 2
    elapsed time: 0.001, Gen_0 yields 2
    elapsed time: 0.001, Gen_0 yields 2
    elapsed time: 0.001, Gen_0 yields 3
    elapsed time: 0.001, Gen_0 yields 3
    elapsed time: 0.001, Gen_0 yields 3
    elapsed time: 0.001, Gen_0 yields 4
    elapsed time: 0.001, Gen_0 yields 4
    elapsed time: 0.001, Gen_0 yields 4
"""
```

Okay, we have a big problem! The GEN ID always remains the same (Gen_0). I must confess this output was quite puzzling to me as well: why the heck is the GEN ID not updated? It turns out that python evaluates the input argument to `from_generator` only once, as such, the `counter` is updated only once. The fix is to force python into evaluating the input argument for each data element passed, which can be done by placing the input argument into a lambda function. Here is the fix and the resulting output:

```
Dataset = tf.data.Dataset
dummy_ds = Dataset.from_tensor_slices(['Gen', 'Gen', 'Gen'])
dummy_ds = dummy_ds.interleave(
    lambda x: Dataset.from_generator(lambda x: DataGen().py_gen(x), output_types=(
       tf.string), args=(x,)),
    cycle_length=3,
    block_length=1,
    num_parallel_calls=3)
data_tf = dummy_ds.make_one_shot_iterator().get_next()
```

```
"""
output:
    elapsed time: 0.327, Gen_0 yields 0
    elapsed time: 0.001, Gen_1 yields 0
    elapsed time: 0.001, Gen_2 yields 0
    elapsed time: 0.001, Gen_0 yields 1
    elapsed time: 0.001, Gen_1 yields 1
    elapsed time: 0.001, Gen_2 yields 1
    elapsed time: 0.001, Gen_0 yields 2
    elapsed time: 0.001, Gen_1 yields 2
    elapsed time: 0.001, Gen_2 yields 2
    elapsed time: 0.001, Gen_0 yields 3
    elapsed time: 0.001, Gen_1 yields 3
    elapsed time: 0.001, Gen_2 yields 3
    elapsed time: 0.001, Gen_0 yields 4
    elapsed time: 0.001, Gen_1 yields 4
    elapsed time: 0.001, Gen_2 yields 4
"""
```

In this example, the bug above resulted in only some error in the print messages, which is not a big deal. However, this bug could have a devastating effect if the generator required vital data from the class attributes.

VI. Example 3

Now, let’s consider a 2-stage data generator where stage 1 reads a large 3D image (e.g., a chest CT scan), and stage 2 extracts a number of local cubes from random locations in this 3D image. Stage 1 is slow, taking 1.5 seconds to load one 3D image, but stage 2 is much faster, taking only 0.3 seconds to return a local cube. We simulate this problem by defining the following data generator:

```
# data generator
def py_gen(num):
    for _ in range(3):
        tensor = np.arange(num * 10, (num + 1) * 10)
        sleep(1.5)
        for x in range(5):
            sleep(0.3)
            yield tensor[x * 2: (x + 1) * 2]
```

where I have used 1D tensors instead of 3D tensors for code brevity. Since we have changed the data generator, some minor modifications in the TF code are necessary:

```
# set up tf generator
Dataset = tf.data.Dataset
ds = Dataset.from_tensor_slices([0, 1, 2])
ds = ds.interleave(lambda x: Dataset.from_generator(py_gen, output_types=(tf.
      float32), args=(x,)),
                               cycle_length=3,
                               block_length=1,
                               num_parallel_calls=3)
data_tf = ds.make_one_shot_iterator().get_next()
```

Running the session gives the following output:

```
"""
output:
    elapsed time: 1.828, [0. 1.]
    elapsed time: 0.001, [10. 11.]
    elapsed time: 0.001, [20. 21.]
    elapsed time: 0.001, [2. 3.]
    elapsed time: 0.001, [12. 13.]
    elapsed time: 0.001, [22. 23.]
    elapsed time: 0.001, [4. 5.]
    elapsed time: 0.001, [14. 15.]
    elapsed time: 0.001, [24. 25.]
    elapsed time: 0.001, [6. 7.]
    elapsed time: 0.001, [16. 17.]
    elapsed time: 0.001, [26. 27.]
    elapsed time: 0.001, [8. 9.]
    elapsed time: 0.001, [18. 19.]
    elapsed time: 0.001, [28. 29.]
    elapsed time: 1.498, [0. 1.]
    elapsed time: 0.001, [10. 11.]
    elapsed time: 0.001, [20. 21.]
    elapsed time: 0.001, [2. 3.]
    elapsed time: 0.001, [12. 13.]
    elapsed time: 0.001, [22. 23.]
    elapsed time: 0.001, [4. 5.]
    elapsed time: 0.001, [14. 15.]
    elapsed time: 0.001, [24. 25.]
    elapsed time: 0.001, [6. 7.]
    elapsed time: 0.001, [16. 17.]
    elapsed time: 0.001, [26. 27.]
    elapsed time: 0.001, [8. 9.]
    elapsed time: 0.001, [18. 19.]
    elapsed time: 0.001, [28. 29.]
    elapsed time: 1.499, [0. 1.]
    elapsed time: 0.001, [10. 11.]
    elapsed time: 0.001, [20. 21.]
    elapsed time: 0.001, [2. 3.]
    elapsed time: 0.001, [12. 13.]
    elapsed time: 0.001, [22. 23.]
    elapsed time: 0.001, [4. 5.]
```

```
    elapsed time: 0.001, [14. 15.]
    elapsed time: 0.001, [24. 25.]
    elapsed time: 0.001, [6. 7.]
    elapsed time: 0.001, [16. 17.]
    elapsed time: 0.001, [26. 27.]
    elapsed time: 0.001, [8. 9.]
    elapsed time: 0.001, [18. 19.]
    elapsed time: 0.001, [28. 29.]
"""
```

Not too bad except that we have a relatively large lag in our data generator after every 15 iterations. This is because we assign 1 thread to each data generator: that same thread is responsible for creating a tensor (1.5 seconds) and then performing tensor slicing (0.3 seconds for each slice). As such, we will experience a delay when all slices are extracted from the current tensor and the next tensor for slicing needs to be created. You may ask what about the other threads? Can they not compensate for this lag? The answer is that since `cycle_length` is set to 3, all 3 threads reach stage 1 at the same time, and while stuck there, no tensor slicing can be performed for the subsequent TF operation.

One fix would be to break the data generator into two sub-generators, each covering one stage of data generation. We can then assign a dedicated thread to each sub-generator and take advantage of asynchronous processing. By doing so, some threads will create the new tensors, while other threads will slice the current tensor. Here are the new data generators:

```
def py_gen_p1(num):
    for _ in range(3):
        sleep(1.5)
        yield np.arange(num * 10, (num+1) * 10)

def py_gen_p2(tensor):
    for x in range(5):
        sleep(0.3)
        yield tensor[x * 2: (x + 1) * 2]
```

And here is how we feed the two generators into the Dataset module of TF:

```
Dataset = tf.data.Dataset
ds = Dataset.from_tensor_slices([0, 1, 2])
ds = ds.interleave(lambda x: Dataset.from_generator(py_gen_p1, output_types=(tf.
    float32), args=(x,)),
                           cycle_length=3,
                           block_length=1,
                           num_parallel_calls=3)
ds = ds.interleave(lambda x: Dataset.from_generator(py_gen_p2, output_types=(tf.
    float32), args=(x,)),
                           cycle_length=3,
                           block_length=1,
                           num_parallel_calls=3)
data_tf = ds.make_one_shot_iterator().get_next()
```

Running the new multi-threaded solution outputs the following:

```
"""
output:
    elapsed time: 1.846, [0. 1.]
    elapsed time: 0.001, [10. 11.]
    elapsed time: 0.001, [20. 21.]
    elapsed time: 0.001, [2. 3.]
    elapsed time: 0.001, [12. 13.]
    elapsed time: 0.001, [22. 23.]
    elapsed time: 0.001, [4. 5.]
    elapsed time: 0.001, [14. 15.]
    elapsed time: 0.001, [24. 25.]
    elapsed time: 0.001, [6. 7.]
    elapsed time: 0.001, [16. 17.]
    elapsed time: 0.001, [26. 27.]
    elapsed time: 0.001, [8. 9.]
    elapsed time: 0.001, [18. 19.]
    elapsed time: 0.001, [28. 29.]
    elapsed time: 0.306, [0. 1.]
    elapsed time: 0.001, [10. 11.]
    elapsed time: 0.001, [20. 21.]
    elapsed time: 0.001, [2. 3.]
    elapsed time: 0.001, [12. 13.]
    elapsed time: 0.001, [22. 23.]
    elapsed time: 0.001, [4. 5.]
    elapsed time: 0.001, [14. 15.]
    elapsed time: 0.001, [24. 25.]
    elapsed time: 0.001, [6. 7.]
    elapsed time: 0.001, [16. 17.]
    elapsed time: 0.001, [26. 27.]
    elapsed time: 0.001, [8. 9.]
    elapsed time: 0.001, [18. 19.]
    elapsed time: 0.001, [28. 29.]
    elapsed time: 0.305, [0. 1.]
    elapsed time: 0.001, [10. 11.]
    elapsed time: 0.001, [20. 21.]
    elapsed time: 0.001, [2. 3.]
    elapsed time: 0.001, [12. 13.]
    elapsed time: 0.001, [22. 23.]
    elapsed time: 0.001, [4. 5.]
    elapsed time: 0.001, [14. 15.]
    elapsed time: 0.001, [24. 25.]
    elapsed time: 0.001, [6. 7.]
    elapsed time: 0.001, [16. 17.]
    elapsed time: 0.001, [26. 27.]
    elapsed time: 0.001, [8. 9.]
    elapsed time: 0.001, [18. 19.]
    elapsed time: 0.001, [28. 29.]
"""
```

As seen, using a dedicated thread for each stage of the data generator reduces the delay from 1.5 seconds to only 0.3 seconds.

Note: The input to the second generator cannot have a complex structure (e.g., a dataset of dictionaries).