Deep Learning System and Parallel Computing HW2

- Author: 黃柏瑄 (P78081528)

1. Experiment description

- Framework: Chainer
- Dataset: Cifar10
- Model: ResNet20

Profile ResNet20 inference for Cifar10 dataset on Chainer without GPU.

2. Environment setting

- Hardware:
  - CPU: Intel(R) Core(TM) i7-10700K CPU @ 3.80GHz (`$ lscpu | grep 'Model name'`)
  - Memory: 32 GB
- Software:
  - OS: Ubuntu 18.04.5 LTS (`$ lsb_release -d`)
  - Python 3.6.9
  - Chainer 7.7.0

```
$ virtualenv --python python3.6 env
$ source env/bin/activate
$ pip install chainer==7.7.0
$ pip list
Package           Version
----------------- -------
chainer           7.7.0
filelock          3.0.12
numpy             1.19.4
pip               20.2.4
protobuf          3.14.0
setuptools        50.3.2
six               1.15.0
typing-extensions 3.7.4.3
wheel             0.35.1
```

3. Methodology

3.1. `hook.print_report()`

This function will print a pretty report but the time are fused by the type. We cannot figure out which part caused more time but only the type.

```
$ time python3 profile.py
         FunctionName  ElapsedTime  Occurrence
Convolution2DFunction     74.51sec      210000
   BatchNormalization     36.02sec      210000
                 ReLU     13.05sec      190000
                  Add      0.48sec       20000
     AveragePooling2D      2.43sec       10000
              Reshape      0.13sec       10000
       LinearFunction      0.33sec       10000
```

3.2. `hook.total_time()`

I found that `TimerHook` accumulated the result, so it could not be done by pseudo code (List 1) to profile the model.

```
# Layer1
print("#Layer1")
hook = TimerHook()
with hook:
    h = F.max_pooling_2d(F.local_response_normalization(F.relu(self.conv1(x))), 3, stride=2)
    print("total time:", hook.total_time(), "s") # Layer1's executioin time
# Layer2
print("#Layer2")
hook = TimerHook()
with hook:
    h = F.max_pooling_2d(F.local_response_normalization(F.relu(self.conv1(x))), 3, stride=2)
    print("total time:", hook.total_time(), "s") # Layer2's executioin time
```

> List 1: Layer2's executioin time would include Layer1's time

3.3. `hook.call_history`

To handle all layer once, I used `call_history`, which provided by `TimerHook`. The `call_history` variable contained layer-wise record with the layer type and the execution time. Therefore, List 2 showed the workable version.

```
hook = TimerHook()
with hook:
    y = model(x)

import pprint
pprint.pprint(hook.call_history) # print the layer-wise result
```

> List 2: use `hook.call_history`

3.3.1. Difference with hw1

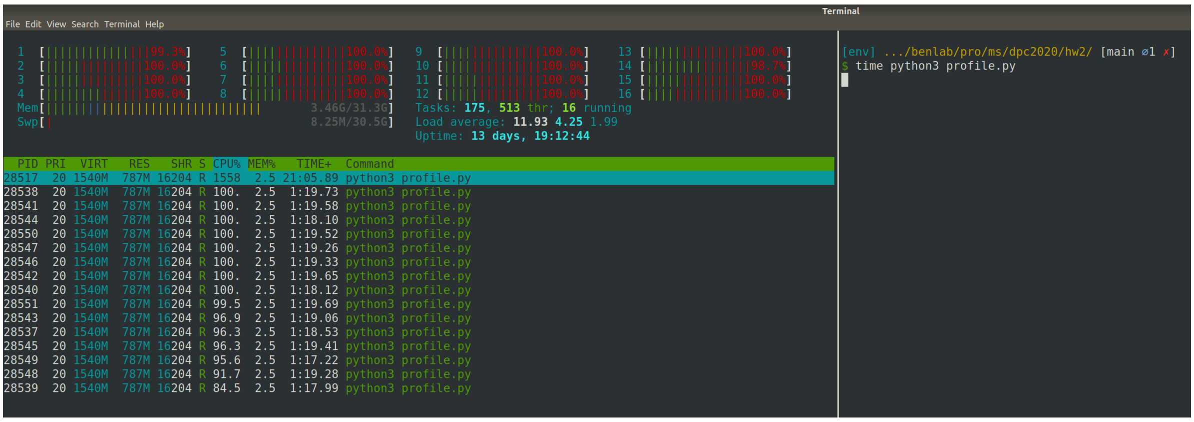
- hw1's inference.py:

```python
y = model(x)  # Inference result
```

- hw2's profile.py:

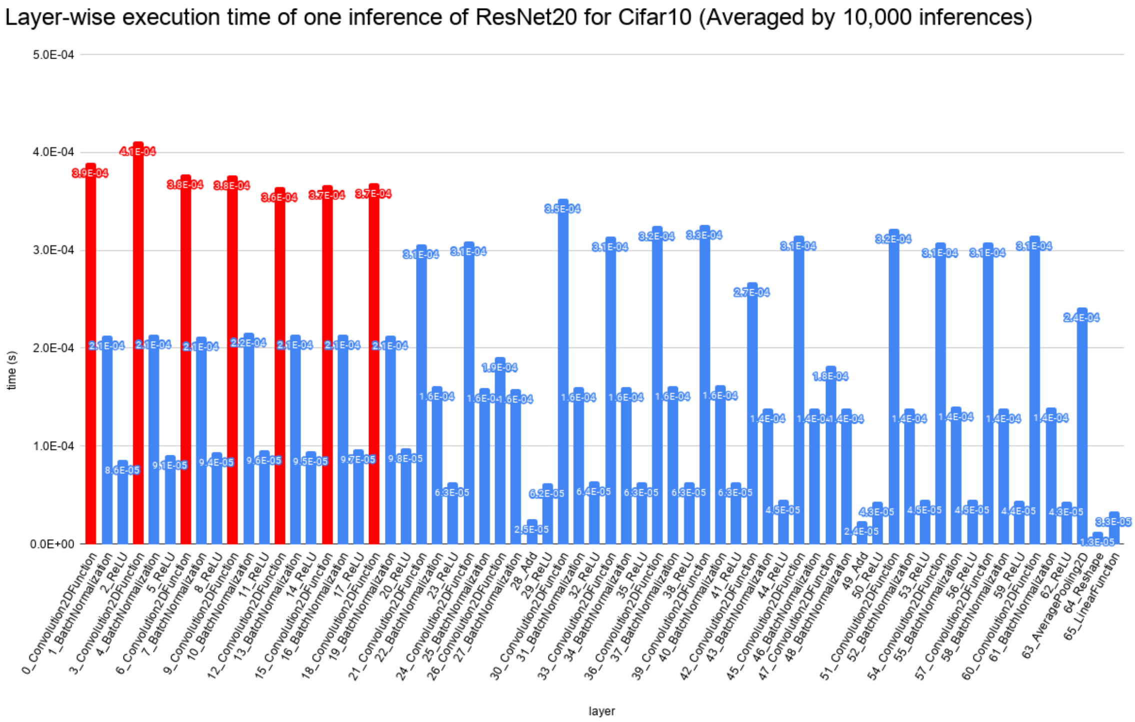
```python
trials = 10000
num_layer = 0  # For counting the number of layers of the model
hook = TimerHook()
y = []
with hook:
    for i in range(trials):
        y = model(x)  # Inference result
        if i == 0:
            num_layer = len(hook.call_history)

result = {}
for i in range(len(hook.call_history)):
    layer_i = i % num_layer
    if not layer_i in result.keys():
        result[layer_i] = {
            "type": hook.call_history[i][0],
            "time": hook.call_history[i][1],
        }
    else:
        result[layer_i]["time"] += hook.call_history[i][1]
    i += 1

average_time = dict(
    map(lambda kv: (kv[0], (kv[1]["time"] / trials)), result.items())
) # Calculate the average time
for k in result.keys():
    print(f'{k}, {result[k]["type"]}, {average_time[k]}')
```

After 10,000 inferences, `call_history` will contain 10,000 * 66 records (66 is the number of the layers of ResNet20). To get the average result, I accumulated the time of the same index, i.e., 0, 66, 132... will summed in 0. The final result showed in section 4.



> Execution progress

4. Result



layer	time (s)
0_Convolution2DFunction	3.9E-04
1_BatchNormalization	2.1E-04
2_ReLU	8.6E-05
3_Convolution2DFunction	4.1E-04
4_BatchNormalization	2.1E-04
5_ReLU	9.1E-05
6_Convolution2DFunction	3.8E-04
7_BatchNormalization	2.1E-04
8_ReLU	9.4E-05
9_Convolution2DFunction	3.8E-04
10_BatchNormalization	2.2E-04
11_ReLU	9.6E-05
12_Convolution2DFunction	3.6E-04
13_BatchNormalization	2.1E-04
14_ReLU	9.5E-05
15_Convolution2DFunction	3.7E-04
16_BatchNormalization	2.1E-04
17_ReLU	9.7E-05
18_Convolution2DFunction	3.7E-04
19_BatchNormalization	2.1E-04
20_ReLU	9.8E-05
21_Convolution2DFunction	3.1E-04
22_BatchNormalization	1.6E-04
23_ReLU	6.3E-05
24_Convolution2DFunction	3.1E-04
25_BatchNormalization	1.6E-04
26_Convolution2DFunction	1.9E-04
27_BatchNormalization	1.6E-04
28_Add	2.5E-05
29_ReLU	6.2E-05
30_Convolution2DFunction	3.5E-04
31_BatchNormalization	1.6E-04
32_ReLU	6.4E-05
33_Convolution2DFunction	3.1E-04
34_BatchNormalization	1.6E-04
35_ReLU	6.3E-05
36_Convolution2DFunction	3.2E-04
37_BatchNormalization	1.6E-04
38_ReLU	6.3E-05
39_Convolution2DFunction	3.3E-04
40_BatchNormalization	1.6E-04
41_ReLU	6.3E-05
42_Convolution2DFunction	2.7E-04
43_BatchNormalization	1.4E-04
44_ReLU	4.5E-05
45_Convolution2DFunction	3.1E-04
46_BatchNormalization	1.4E-04
47_Convolution2DFunction	1.8E-04
48_BatchNormalization	1.4E-04
49_Add	2.4E-05
50_ReLU	4.3E-05
51_Convolution2DFunction	3.2E-04
52_BatchNormalization	1.4E-04
53_ReLU	4.5E-05
54_Convolution2DFunction	3.1E-04
55_BatchNormalization	1.4E-04
56_ReLU	4.5E-05
57_Convolution2DFunction	3.1E-04
58_BatchNormalization	1.4E-04
59_ReLU	4.4E-05
60_Convolution2DFunction	3.1E-04
61_BatchNormalization	1.4E-04
62_ReLU	4.3E-05
63_AveragePooling2D	2.4E-04
64_Reshape	1.3E-05
65_LinearFunction	3.3E-05

> Raw data

5. Next step for improvement

We can see the red bars (time >= 3.6E-4) are all convolution layers. Therefore, the improvement may be done by parallel these layers or leverage GPU to accelerate the execution.

6. Appendix

6.1. Code

To avoid duplicate code, I isolated the model's code into independent file and import it in train and inference code.

- Code architecture

```
$ tree . -I 'env|result|__pycache__|images|*.md|*.pdf'
.
├── profile.py
└── resnet_cifar10.py
```

- Model (`resnet_cifar10.py`)
  - Reference:
    1. https://github.com/akamaster/pytorch_resnet_cifar10/blob/master/resnet.py
    2. https://github.com/mitmul/chainer-cifar10/blob/master/models/ResNet.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# Reference:
#   1. https://github.com/akamaster/pytorch_resnet_cifar10/blob/master/resnet.py
#   2. https://github.com/mitmul/chainer-cifar10/blob/master/models/ResNet.py
import chainer
import chainer.functions as F
import chainer.links as L


class BottleNeck(chainer.Chain):
    def __init__(self, n_in, n_out, stride=1):
        self.shortcut = stride != 1

        super(BottleNeck, self).__init__()
        with self.init_scope():
            self.conv1 = L.Convolution2D(
                n_in, n_out, ksize=3, stride=stride, pad=1, nobias=True
            )
            self.bn1 = L.BatchNormalization(n_out)
            self.conv2 = L.Convolution2D(
                n_out, n_out, ksize=3, stride=1, pad=1, nobias=True
            )
            self.bn2 = L.BatchNormalization(n_out)
            self.conv3 = L.Convolution2D(
                n_in, n_out, ksize=1, stride=stride, pad=0, nobias=True
            )
            self.bn3 = L.BatchNormalization(n_out)

    def __call__(self, x):
        h = F.relu(self.bn1(self.conv1(x)))
        h = self.bn2(self.conv2(h))
        if self.shortcut:
            h += self.bn3(self.conv3(x))
        h = F.relu(h)
        return h


class Block(chainer.ChainList):
    def __init__(self, n_in, n_out, n_bottlenecks, stride):
        super(Block, self).__init__()
        self.in_planes = n_in
        self.n_out = n_out
        strides = [stride] + [1] * (n_bottlenecks - 1)
        for stride in strides:
            self.add_link(BottleNeck(self.in_planes, n_out, stride))
```

```
            self.in_planes = n_out

    def __call__(self, x):
        for f in self:
            x = f(x)
        return x


class ResNet(chainer.Chain):
    def __init__(self, n_class=10, n_blocks=[3, 3, 3]):
        super(ResNet, self).__init__()

        with self.init_scope():
            self.conv1 = L.Convolution2D(None, 16, 3, 1, 1, nobias=True)
            self.bn2 = L.BatchNormalization(16)
            self.res3 = Block(16, 16, n_blocks[0], 1)
            self.res4 = Block(16, 32, n_blocks[1], 2)
            self.res5 = Block(32, 64, n_blocks[2], 2)
            self.fc7 = L.Linear(64, n_class)

    def __call__(self, x):
        h = F.relu(self.bn2(self.conv1(x)))
        h = self.res3(h)
        h = self.res4(h)
        h = self.res5(h)
        h = F.average_pooling_2d(h, h.shape[2:])
        h = self.fc7(h)
        return h


class ResNet20(ResNet):
    def __init__(self, n_class=10):
        super(ResNet20, self).__init__(n_class, [3, 3, 3])
```

- Train script (`profile.py`)

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
from resnet_cifar10 import ResNet20
import argparse
import chainer
from chainer import serializers
from chainer.function_hooks import TimerHook

import numpy as np


parser = argparse.ArgumentParser(description="Chainer example: Cifar-10")
parser.add_argument(
    "--out",
    "-o",
    default="../hw1/result/5/resnet20.model",
    help="Directory to output the result",
)
parser.add_argument(
    "--unit", "-u", type=int, default=10, help="Number of output layer units"
)
# parser.add_argument('--gpu', '-g', type=int, default=0,
# help='GPU ID (negative value indicates CPU)')#Set the initial matrixformat(numpy/cupy)
if __name__ == "__main__":
    args = parser.parse_args()  # The negative device number means CPU.
    # recorder.init()
    # print('GPU: {}'.format(args.gpu))
    # print("# unit: {}".format(args.unit))
    # print("")

    # Load the dataset
    _, test = chainer.datasets.get_cifar10()
    # Load trained model

    model = ResNet20(args.unit)
    # if args.gpu >= 0:
    # chainer.cuda.get_device(args.gpu).use() # Make a specified GPU current
    # model.to_gpu() # Copy the model to the GPU
    # xp = np if args.gpu < 0 else chainer.cuda.cupy
    xp = np

    serializers.load_npz(args.out, model)
```

```python
x = chainer.Variable(xp.asarray([test[0][0]]))  # test data
t = chainer.Variable(xp.asarray([test[0][1]]))  # labels

trials = 10000
num_layer = 0  # For counting the number of layers of the model
hook = TimerHook()
y = []
with hook:
    for i in range(trials):
        y = model(x)  # Inference result
        if i == 0:
            num_layer = len(hook.call_history)

i = 0
result = {}
for i in range(len(hook.call_history)):
    layer_i = i % num_layer
    if not layer_i in result.keys():
        result[layer_i] = {
            "type": hook.call_history[i][0],
            "time": hook.call_history[i][1],
        }
    else:
        result[layer_i]["time"] += hook.call_history[i][1]
    i += 1

average_time = dict(
    map(lambda kv: (kv[0], (kv[1]["time"] / trials)), result.items())
)
# import pprint
# pprint.pprint(hook.call_history)
# pprint.pprint(result)
# pprint.pprint(average_time)
for k in result.keys():
    print(f'{k}, {result[k]["type"]}, {average_time[k]}')

# print("The test data label:", xp.asarray([test[0][1]]))
# print("result:", y)
# y_top5 = y.array[0].argsort()[-5:][::-1]
# print("result Top 1:", [y_top5[0]])
# print("result Top 5:", y_top5)
```