Deep Learning System and Parallel Computing HW2

- Author: 黃柏瑄 (P78081528)

Experiment description

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

Improve ResNet20 inference for Cifar10 dataset on Chainer with GPU.

Environment setting

- Hardware:
  - CPU: Intel(R) Core(TM) i7-10700K CPU @ 3.80GHz (`$ lscpu | grep 'Model name'`)
  - Memory: 32 GB
  - GPU: GTX 1060 Mini ITX OC 3G
- Software:
  - OS: Ubuntu 18.04.5 LTS (`$ lsb_release -d`)
  - Nvdia GPU driver 455.23.05 (`$ nvidia-smi --query | grep 'Driver Version'`)
  - CUDA 10.2 (`$ nvidia-smi --query | grep 'CUDA Version'`)
  - Python 3.6.9
  - Chainer 7.7.0

```
$ ubuntu-drivers devices
$ sudo apt install nvidia-driver-455
$ sudo reboot
$ nvidia-smi # Check the installation of Nvidia driver

$ wget
https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1804/x86_64/cuda-
ubuntu1804.pin
$ sudo mv cuda-ubuntu1804.pin /etc/apt/preferences.d/cuda-repository-pin-600
$ wget
https://developer.download.nvidia.com/compute/cuda/10.2/Prod/local_installers/cud
a-repo-ubuntu1804-10-2-local-10.2.89-440.33.01_1.0-1_amd64.deb
$ sudo dpkg -i cuda-repo-ubuntu1804-10-2-local-10.2.89-440.33.01_1.0-1_amd64.deb
$ sudo apt-key add /var/cuda-repo-10-2-local-10.2.89-440.33.01/7fa2af80.pub
$ sudo apt-get update
$ sudo apt-get -y install cuda
$ printf '\nexport CUDA_HOME=/usr/local/cuda-10.2' >> ~/.bashrc
$ printf '\nexport LD_LIBRARY_PATH=${CUDA_HOME}/lib64:$LD_LIBRARY_PATH' >>
~/.bashrc
$ printf '\nexport PATH=$PATH:$CUDA_HOME/bin' >> ~/.bashrc
$ source ~/.bashrc
$ nvcc -V  # Check the installation of CUDA
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2019 NVIDIA Corporation
Built on Wed_Oct_23_19:24:38_PDT_2019
Cuda compilation tools, release 10.2, V10.2.89
$ sudo reboot # Solve "Failed to initialize NVML: Driver/library version
mismatch" of nvidia-smi

$ tar -xzvf cudnn-10.2-linux-x64-v8.0.5.39.tgz
$ sudo cp cuda/include/cudnn*.h /usr/local/cuda/include
$ sudo cp cuda/lib64/libcudnn* /usr/local/cuda/lib64
$ sudo chmod a+r /usr/local/cuda/include/cudnn*.h /usr/local/cuda/lib64/libcudnn*
```

```
$ virtualenv --python python3.6 env
$ source env/bin/activate
$ pip install chainer==7.7.0
$ pip install 'cupy-cuda102>=7.7.0,<8.0.0'
$ pip install tqdm
$ pip list
Package           Version
----------------- -------
chainer           7.7.0
cupy-cuda102      7.8.0
fastrlock         0.5
filelock          3.0.12
numpy             1.19.4
pip               20.2.4
protobuf          3.14.0
setuptools        50.3.2
six               1.15.0
tqdm              4.54.1
typing-extensions 3.7.4.3
wheel             0.35.1
```

Execution

- Measure the layer-wise execution time of all CPU version

```
$ python profile.py -p time -m 0 -o ./time_0.txt
```

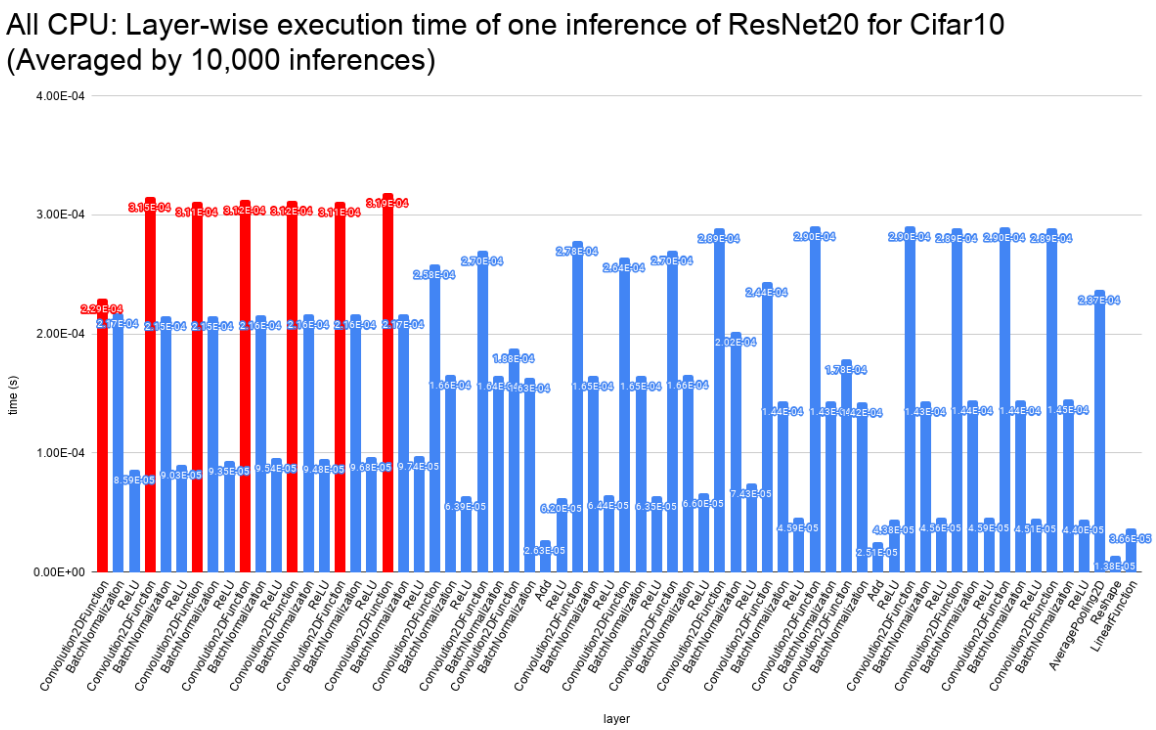
- Measure the layer-wise execution time of all GPU version

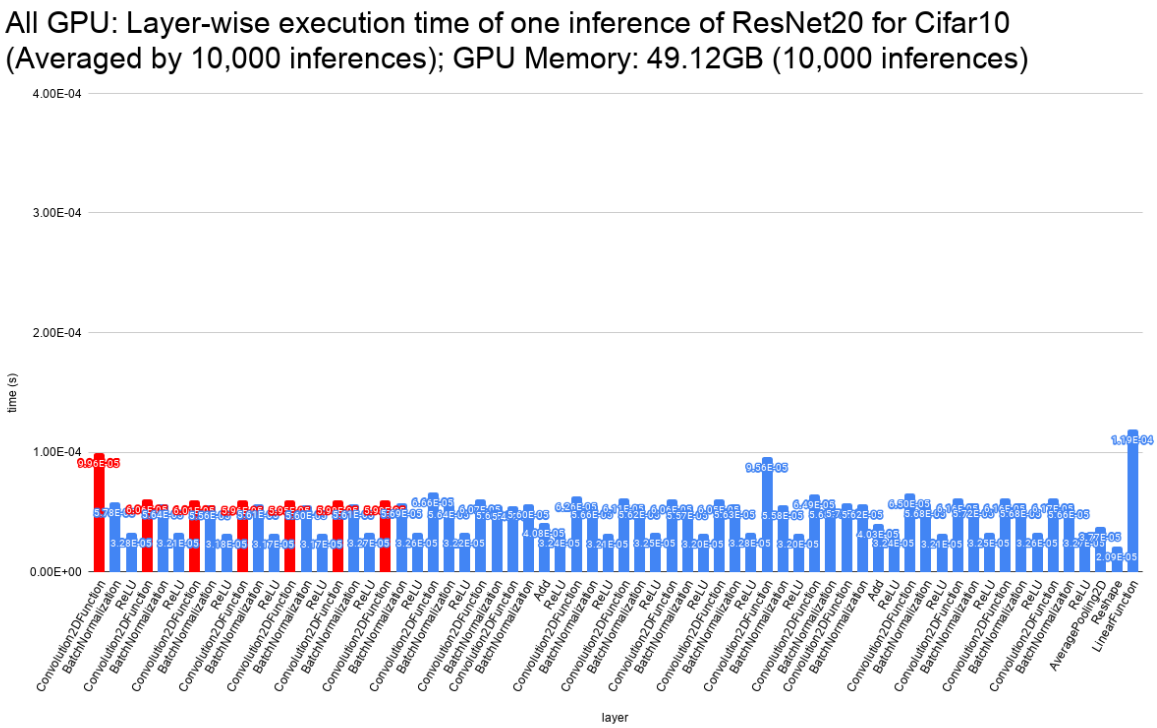
```
$ python profile.py -p time -m 1 -o ./time_1.txt
```

- Measure the layer-wise execution time of some Conv2D on GPU version

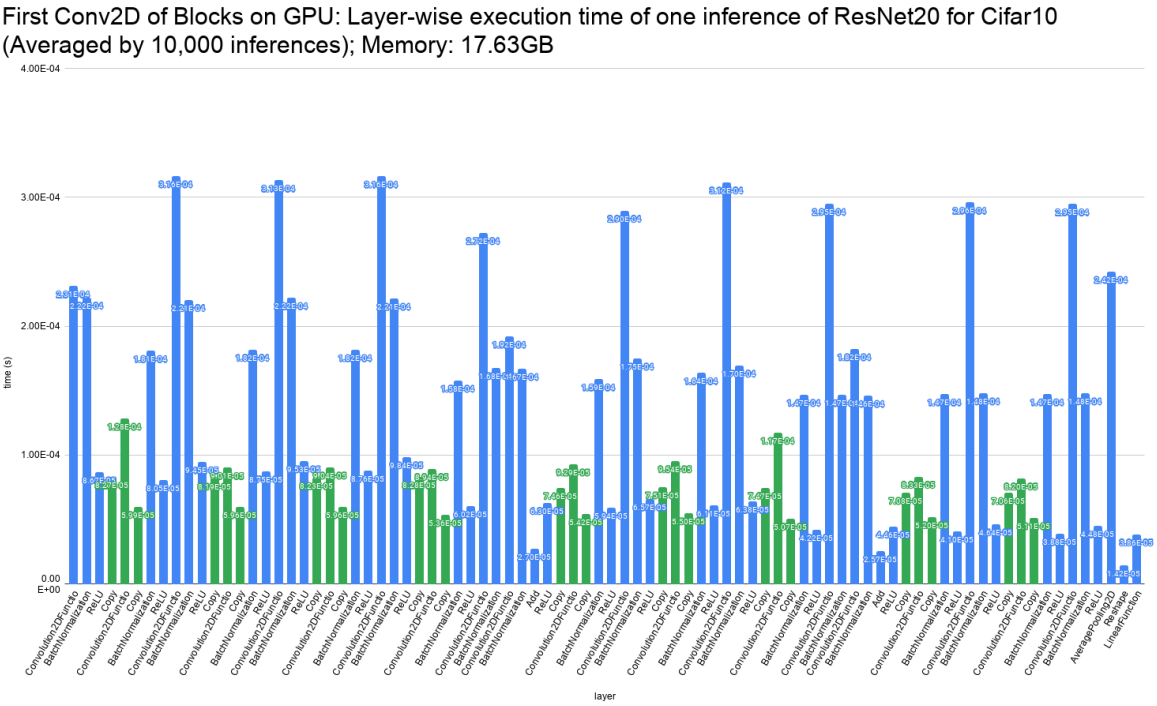
```
$ python profile.py -p time -m 2 -o ./time_2.txt
```

Result





We found that the memory usage was a lot, so another approach was came out by copying first Conv2D to GPU for each block. However, the total execution time had very small improvement when comparing with all CPU version.



Appendix

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|img|*.txt'
.
├── common.py
├── 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

```python
#!/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
import cupy
import common


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():
            if common.FIRST_CONV_GPU_FLAG:
                self.conv1 = L.Convolution2D(
                    n_in, n_out, ksize=3, stride=stride, pad=1, nobias=True
                ).to_gpu(0)
            else:
                self.conv1 = L.Convolution2D(
                    n_in, n_out, ksize=3, stride=stride, pad=1, nobias=True
                )
            # self.conv1_gpu = L.Convolution2D(
            #     n_in, n_out, ksize=3, stride=stride, pad=1, nobias=True
            # ).to_gpu(0)
            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 = x
        if common.FIRST_CONV_GPU_FLAG:
            h = F.copy(h, 0)
            h = self.conv1(h)
```

```
            h = F.copy(h, -1)
        else:
            h = self.conv1(h)
        h = F.relu(self.bn1(h))
        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)
            # common.FIRST_CONV_GPU_FLAG = False
            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])
```

- For cross file global variable (`common.py`)

```
global FIRST_CONV_GPU_FLAG
FIRST_CONV_GPU_FLAG = False
```

- 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
from tqdm import tqdm

import numpy as np
import cupy
import common


parser = argparse.ArgumentParser(description="Chainer example: Cifar-10")
parser.add_argument(
    "--weights",
    "-w",
    default="../hw1/result/5/resnet20.model",
    help="Path of the trained weights",
)
parser.add_argument(
    "--unit", "-u", type=int, default=10, help="Number of output layer units"
)
parser.add_argument(
    "--mode",
    "-m",
    type=int,
    default=0,
    help="0: All CPU; 1: All GPU; 2: First Conv for each Block on GPU",
)
parser.add_argument(
    "--profile", "-p", type=str, default="time", choices=["time", "memory"]
)
parser.add_argument(
    "--output", "-o", type=str, default="./log.txt", help="Log path for output"
)
# 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.

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

    xp = np
    common.FIRST_CONV_GPU_FLAG = False
    if args.mode == 2:
        common.FIRST_CONV_GPU_FLAG = True
```

```
    model = ResNet20(args.unit)
    if args.mode == 0:
        pass
    elif args.mode == 1:
        model.to_gpu()  # Copy the model to the GPU
        xp = cupy

    # Load trained model
    serializers.load_npz(args.weights, model)

    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
    pbar = tqdm(total=trials)
    if args.profile == "time":
        hook = TimerHook()
        y = []
        with hook:
            for i in range(trials):
                y = model(x)  # Inference result
                if i == 0:
                    num_layer = len(hook.call_history)
                pbar.update()
            pbar.close()

        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())
        )
        # hook.print_report()
        # import pprint
        # pprint.pprint(hook.call_history)
        # pprint.pprint(result)
        # pprint.pprint(average_time)
        with open(args.output, "w") as fout:
            for k in result.keys():
                fout.write(f'{k}, {result[k]["type"]}, {average_time[k]}\n')

    elif args.profile == "memory":
        from cupy.cuda import memory_hooks

        mem_hook = memory_hooks.LineProfileHook(max_depth=0)
        y = []
        with mem_hook:
            for i in range(trials):
```

```
                y = model(x)  # Inference result
                pbar.update()
            pbar.close()
        with open(args.output, "w") as fout:
            mem_hook.print_report(file=fout)
```