EE 5545 Assignment 1

Ethan Glaser (eg492)

Question 1:

Name, Year, Type, peak FLOPs, peak memory bandwidth, source

NVIDIA GeForce 256 SDR, 1999, GPU, 50 GFLOPs, 2.656 GB/s, source source

NVIDIA TITAN Xp, 2017, GPU, 11366.4 GFLOPs, 547.7 GB/s, source

NVIDIA TESLA K80, 2014, GPU, 4.113 TFLOPs, 240.6 GB/s, source

Intel i7 920 2008, CPU, 63 GFLOPs, 25.6 GB/s, source

AMD Ryzen 9 3900X, 2019, CPU, 2649.6 GFLOPs, 47.68 GB/s, source

AMD Radeon RX 6800, 2019, GPU, 18688 GFLOPS, 512 GB/s, source

Apple A14, 2020, SoC, 1000 GFLOPs, 42.7 GB/s, source

Apple M1 Max, 2020, SoC, 2.7 TFLOPs, 330 GB/s, source

TPU v3, 2018, ASIC, 420 TFLOPs, 1 TB/s, source

Most of the results are as expected – the GPU from 1999 and CPU on my laptop are both significantly lower performing (in terms of FLOPS/s and memory bandwidth) compared to the more traditional GPUs. Certain GPUs were higher performance than others, as mid-range and high-end GPUs were selected from both NVIDIA and AMD. The TPU had by far the highest FLOPS/s and memory bandwidth than the GPUs. These results are illustrated in the roofline plot below. No FLOPS/s values were calculated as the data was available on the various datasheets and pages associated with each device.

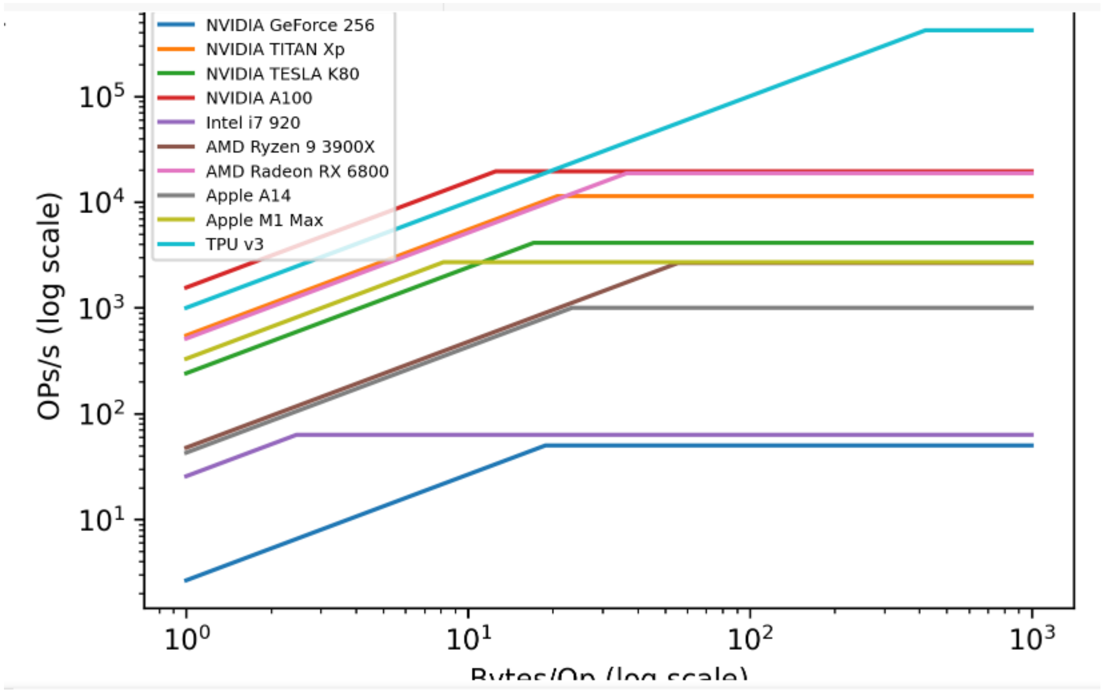
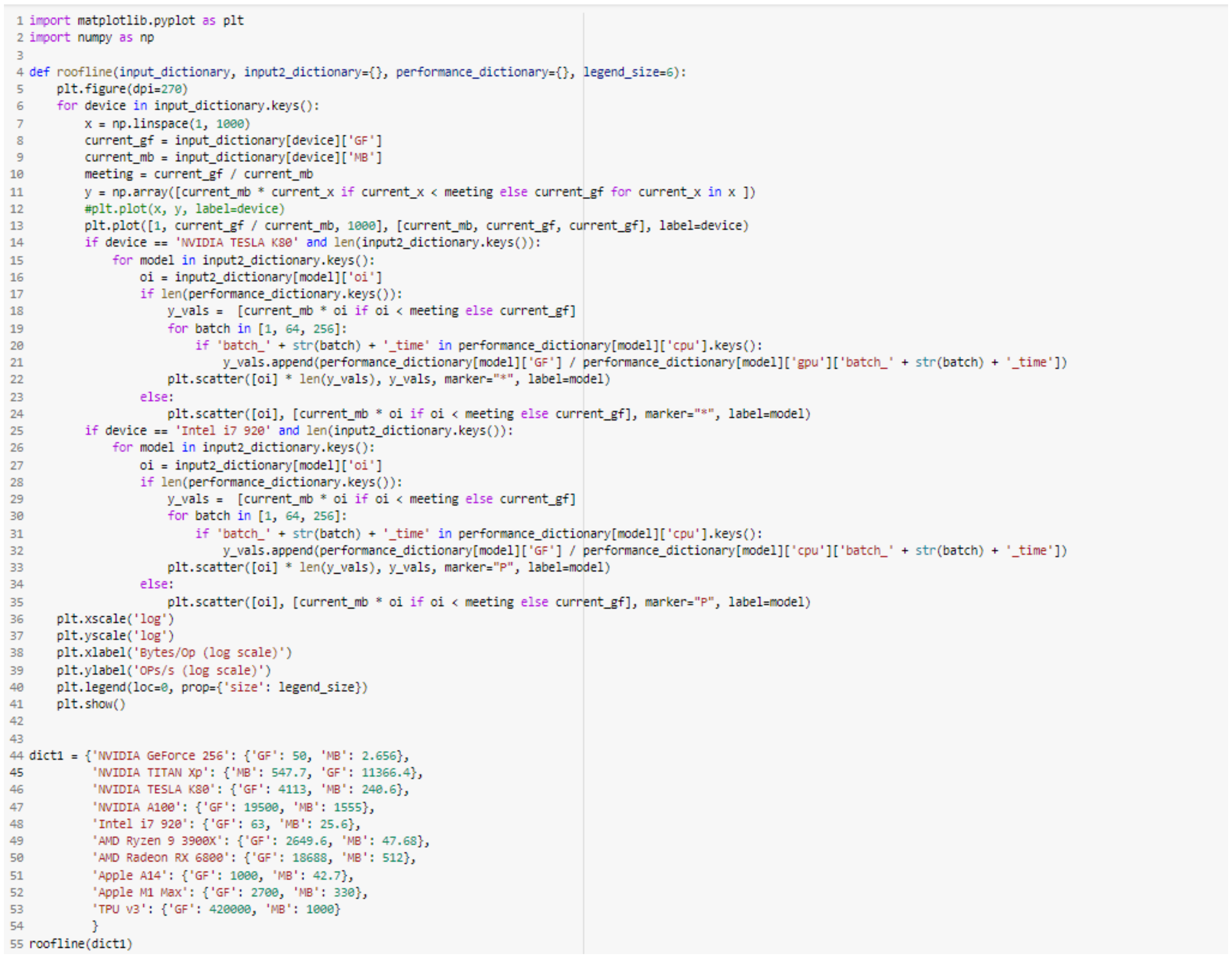


Figure 1: Roofline Plot for Q1



```python
import matplotlib.pyplot as plt
import numpy as np

def roofline(input_dictionary, input2_dictionary={}, performance_dictionary={}, legend_size=6):
    plt.figure(dpi=270)
    for device in input_dictionary.keys():
        x = np.linspace(1, 1000)
        current_gf = input_dictionary[device]['GF']
        current_mb = input_dictionary[device]['MB']
        meeting = current_gf / current_mb
        y = np.array([current_mb * current_x if current_x < meeting else current_gf for current_x in x ])
        #plt.plot(x, y, label=device)
        plt.plot([1, current_gf / current_mb, 1000], [current_mb, current_gf, current_gf], label=device)
        if device == 'NVIDIA TESLA K80' and len(input2_dictionary.keys()):
            for model in input2_dictionary.keys():
                oi = input2_dictionary[model]['oi']
                if len(performance_dictionary.keys()):
                    y_vals =  [current_mb * oi if oi < meeting else current_gf]
                    for batch in [1, 64, 256]:
                        if 'batch_' + str(batch) + '_time' in performance_dictionary[model]['cpu'].keys():
                            y_vals.append(performance_dictionary[model]['GF'] / performance_dictionary[model]['gpu']['batch_' + str(batch) + '_time'])
                    plt.scatter([oi] * len(y_vals), y_vals, marker="*", label=model)
                else:
                    plt.scatter([oi], [current_mb * oi if oi < meeting else current_gf], marker="*", label=model)
        if device == 'Intel i7 920' and len(input2_dictionary.keys()):
            for model in input2_dictionary.keys():
                oi = input2_dictionary[model]['oi']
                if len(performance_dictionary.keys()):
                    y_vals =  [current_mb * oi if oi < meeting else current_gf]
                    for batch in [1, 64, 256]:
                        if 'batch_' + str(batch) + '_time' in performance_dictionary[model]['cpu'].keys():
                            y_vals.append(performance_dictionary[model]['GF'] / performance_dictionary[model]['cpu']['batch_' + str(batch) + '_time'])
                    plt.scatter([oi] * len(y_vals), y_vals, marker="P", label=model)
                else:
                    plt.scatter([oi], [current_mb * oi if oi < meeting else current_gf], marker="P", label=model)
    plt.xscale('log')
    plt.yscale('log')
    plt.xlabel('Bytes/Op (log scale)')
    plt.ylabel('OPs/s (log scale)')
    plt.legend(loc=0, prop={'size': legend_size})
    plt.show()


dict1 = {'NVIDIA GeForce 256': {'GF': 50, 'MB': 2.656},
         'NVIDIA TITAN Xp': {'MB': 547.7, 'GF': 11366.4},
         'NVIDIA TESLA K80': {'GF': 4113, 'MB': 240.6},
         'NVIDIA A100': {'GF': 19500, 'MB': 1555},
         'Intel i7 920': {'GF': 63, 'MB': 25.6},
         'AMD Ryzen 9 3900X': {'GF': 2649.6, 'MB': 47.68},
         'AMD Radeon RX 6800': {'GF': 18688, 'MB': 512},
         'Apple A14': {'GF': 1000, 'MB': 42.7},
         'Apple M1 Max': {'GF': 2700, 'MB': 330},
         'TPU v3': {'GF': 420000, 'MB': 1000}
         }
roofline(dict1)
```

Figure 2: Code to plot roofline, used in Q1 and expanded on for use in Q2 and Q4

Question 2:

Model	FLOPs	Memory footprint
ResNet50	8.22 GFLOPs	384.62 MB
AlexNet	1.43 GFLOPs	242.03 MB
MobileNet v3 large	0.45 GFLOPs	126.9 MB
VGG19	39.29 GFLOPs	787.31 MB
SqueezeNet1_0	1.65 GFLOPs	97.14 MB
EfficientNetb7	10.53 GFLOPs	1.33 GB
GoogleNet	3.01 GFLOPs	119.95 MB
ShuffleNet	0.09 GFLOPs	30.85 MB
WideResNet101_2	45.59 GFLOPs	1.03 GB
Inception v3	5.69 GFLOPs	209.48 MB

Every model was compute bound on the CPU, whereas about half (shufflenet, alexnet, efficientnet, mobilenet v3) with relatively lower operational intensities were memory bound on the GPU provided by colab (Nvidia TESLA K80) – as seen in Figure 4.

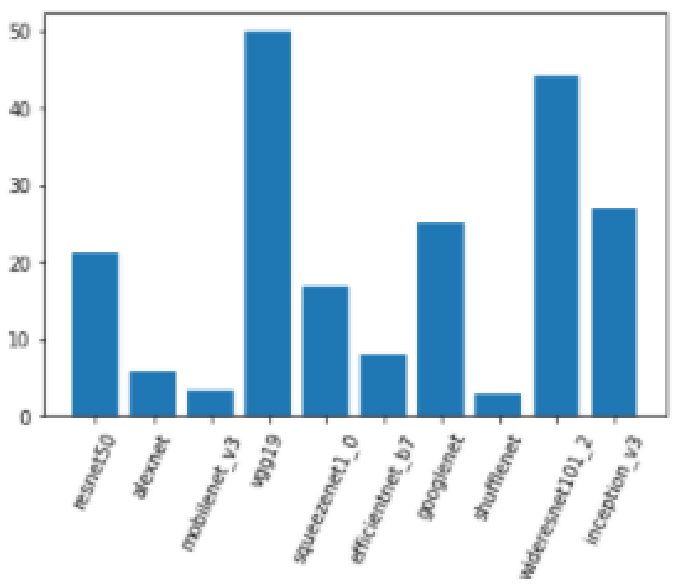


Figure 3: Operational Intensity (Bytes/Op) for each of 10 DNN models

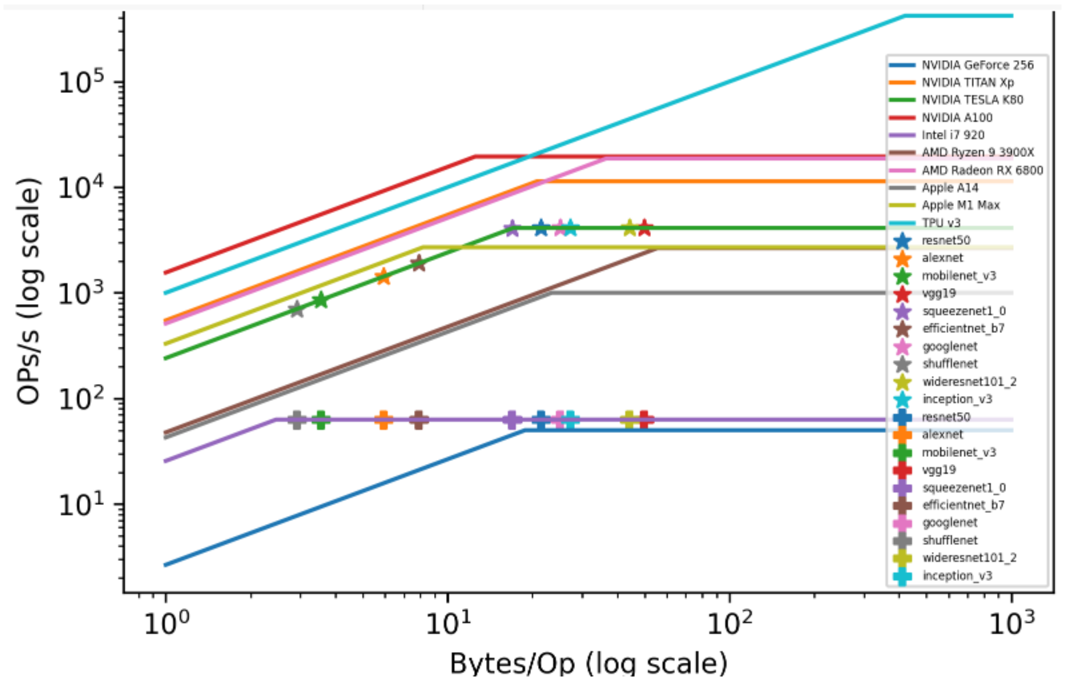


Figure 4: Updated Roofline Plot for Q2 with expected model performance for GPU and CPU indicated

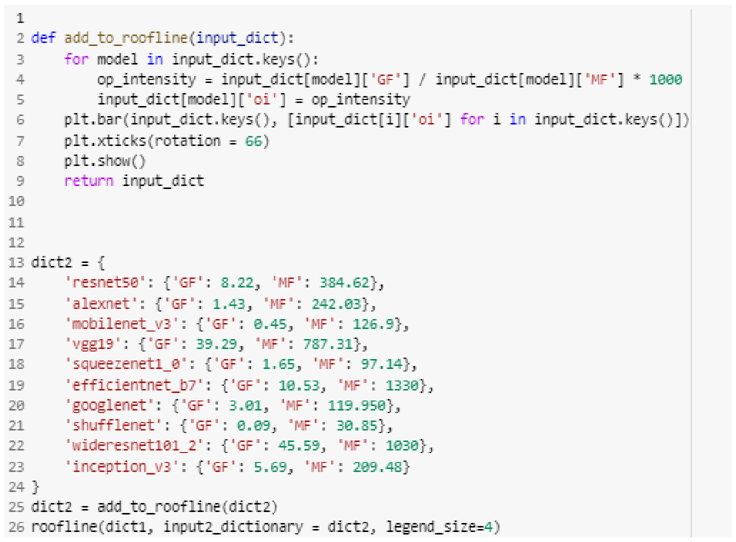
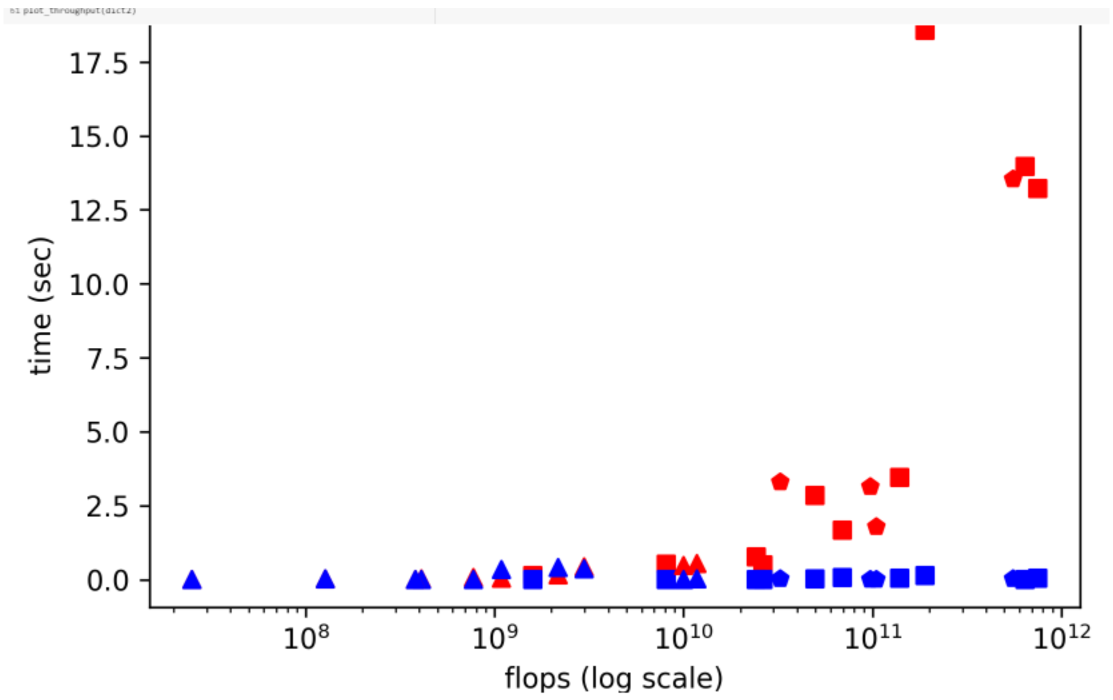

```

def add_to_roofline(input_dict):
    for model in input_dict.keys():
        op_intensity = input_dict[model]['GF'] / input_dict[model]['MF'] * 1000
        input_dict[model]['oi'] = op_intensity
    plt.bar(input_dict.keys(), [input_dict[i]['oi'] for i in input_dict.keys()])
    plt.xticks(rotation = 66)
    plt.show()
    return input_dict



dict2 = {
    'resnet50': {'GF': 8.22, 'MF': 384.62},
    'alexnet': {'GF': 1.43, 'MF': 242.03},
    'mobilenet_v3': {'GF': 0.45, 'MF': 126.9},
    'vgg19': {'GF': 39.29, 'MF': 787.31},
    'squeezenet1_0': {'GF': 1.65, 'MF': 97.14},
    'efficientnet_b7': {'GF': 10.53, 'MF': 1330},
    'googlenet': {'GF': 3.01, 'MF': 119.950},
    'shufflenet': {'GF': 0.09, 'MF': 30.85},
    'wideresnet101_2': {'GF': 45.59, 'MF': 1030},
    'inception_v3': {'GF': 5.69, 'MF': 209.48}
}
dict2 = add_to_roofline(dict2)
roofline(dict1, input2_dictionary = dict2, legend_size=4)
```

Figure 5: Code used to generate Q2 results (note function call to main Q1 function with additional argument)

Question 3:

The Spearman Coefficients indicate that there is a strong correlation between FLOPs and CPU latency, suggesting that FLOPs are a good indicator of the CPU's runtime with a coefficient of 0.83. Number of parameters is also somewhat correlated to CPU latency but with a less strong relationship (0.39). These indicators are much less telling for GPU latency, with Spearman coefficients of -0.14 and -0.03 respectively. Therefore there is not a strong relationship between the FLOPs or parameters and the GPU latency, which may be a result of the parallelization of operations running on the GPU, since batches may be handled uniformly instead of in a more sequential manner like with the CPU. The plots of flops and parameters vs. latency aren't particularly revealing aside from showing the general trend that the latency is generally much higher on the CPU and the opposite for throughput.



```
61 plot_throughput(dict2)
```

Figure 6: Latency vs FLOPs (where CPU in red, GPU in blue, batch size 1 as triangles, batch size 64 as squares, and batch size 256 as pentagons)

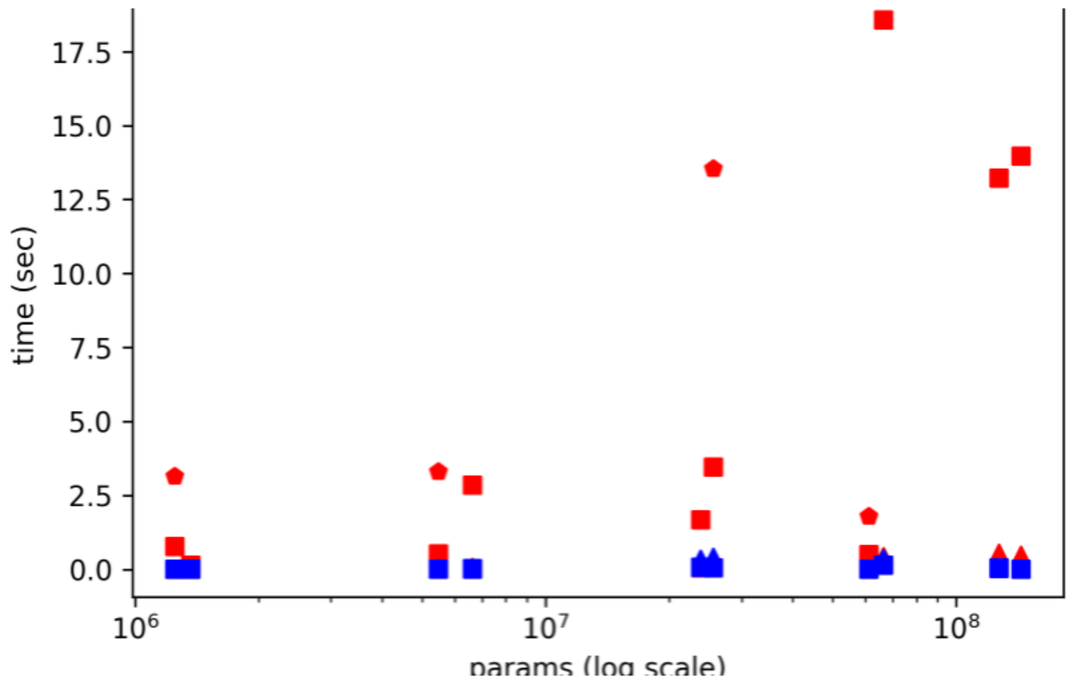


Figure 7: Latency vs parameters (where CPU in red, GPU in blue, batch size 1 as triangles, batch size 64 as squares, and batch size 256 as pentagons)

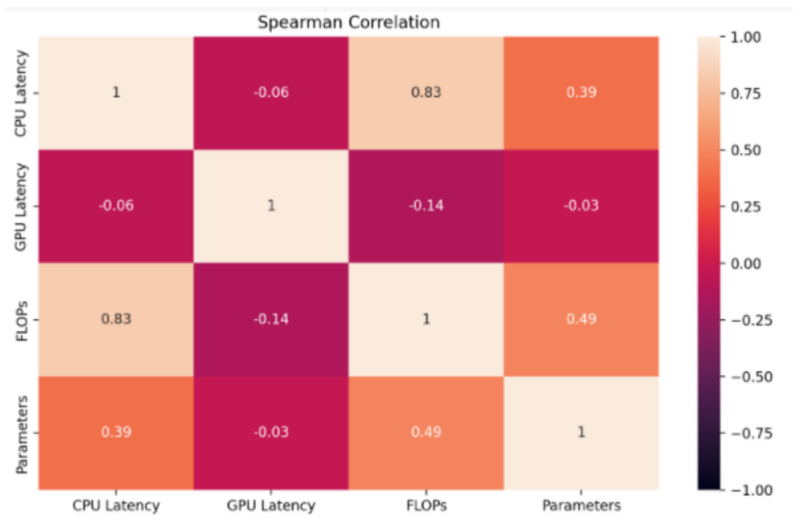


Figure 8: Spearman Correlation between CPU latency, GPU latency, FLOPs, and number of parameters

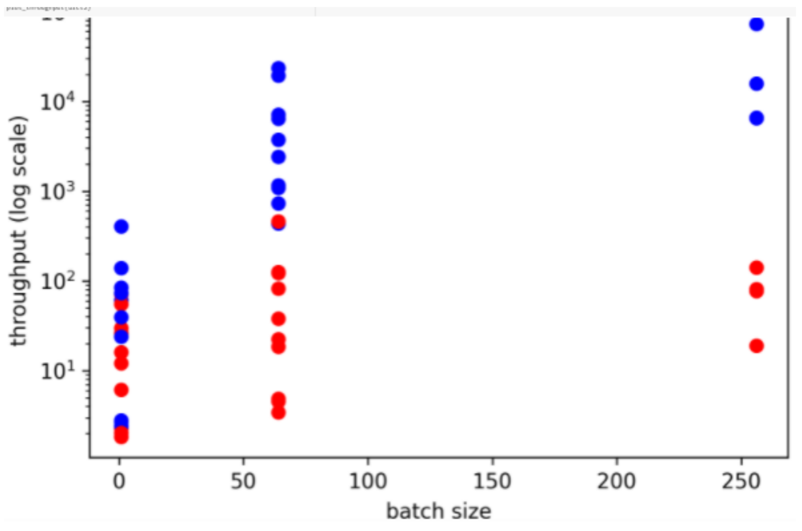
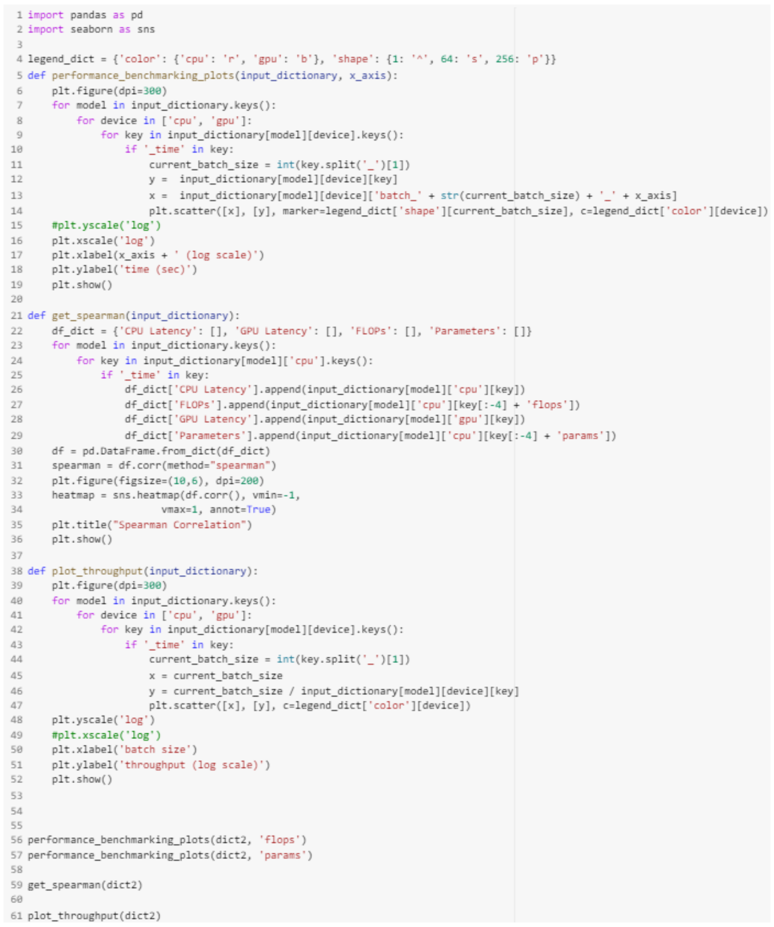


Figure 9: Throughput vs. Batch Size (CPU in red, GPU in blue)



```python
import pandas as pd
import seaborn as sns

legend_dict = {'color': {'cpu': 'r', 'gpu': 'b'}, 'shape': {1: '^', 64: 's', 256: 'p'}}
def performance_benchmarking_plots(input_dictionary, x_axis):
    plt.figure(dpi=300)
    for model in input_dictionary.keys():
        for device in ['cpu', 'gpu']:
            for key in input_dictionary[model][device].keys():
                if '_time' in key:
                    current_batch_size = int(key.split('_')[1])
                    y =  input_dictionary[model][device][key]
                    x =  input_dictionary[model][device]['batch_' + str(current_batch_size) + '_' + x_axis]
                    plt.scatter([x], [y], marker=legend_dict['shape'][current_batch_size], c=legend_dict['color'][device])
    #plt.yscale('log')
    plt.xscale('log')
    plt.xlabel(x_axis + ' (log scale)')
    plt.ylabel('time (sec)')
    plt.show()

def get_spearman(input_dictionary):
    df_dict = {'CPU Latency': [], 'GPU Latency': [], 'FLOPs': [], 'Parameters': []}
    for model in input_dictionary.keys():
        for key in input_dictionary[model]['cpu'].keys():
            if '_time' in key:
                df_dict['CPU Latency'].append(input_dictionary[model]['cpu'][key])
                df_dict['FLOPs'].append(input_dictionary[model]['cpu'][key[:-4] + 'flops'])
                df_dict['GPU Latency'].append(input_dictionary[model]['gpu'][key])
                df_dict['Parameters'].append(input_dictionary[model]['cpu'][key[:-4] + 'params'])
    df = pd.DataFrame.from_dict(df_dict)
    spearman = df.corr(method="spearman")
    plt.figure(figsize=(10,6), dpi=200)
    heatmap = sns.heatmap(df.corr(), vmin=-1,
                          vmax=1, annot=True)
    plt.title("Spearman Correlation")
    plt.show()

def plot_throughput(input_dictionary):
    plt.figure(dpi=300)
    for model in input_dictionary.keys():
        for device in ['cpu', 'gpu']:
            for key in input_dictionary[model][device].keys():
                if '_time' in key:
                    current_batch_size = int(key.split('_')[1])
                    x = current_batch_size
                    y = current_batch_size / input_dictionary[model][device][key]
                    plt.scatter([x], [y], c=legend_dict['color'][device])
    plt.yscale('log')
    #plt.xscale('log')
    plt.xlabel('batch size')
    plt.ylabel('throughput (log scale)')
    plt.show()



performance_benchmarking_plots(dict2, 'flops')
performance_benchmarking_plots(dict2, 'params')

get_spearman(dict2)

plot_throughput(dict2)
```

Figure 10: Code used to complete Q3 with 3 main functions for each of the plots (first 2 bullet points combined into the first function)

Question 4:

There are many interesting trends that can be observed in Figure 11. One noteworthy observation that doesn't make a ton of sense is that some implementations of vgg19, alexnet, and a few other models outperformed the roofline plot limits, which may be due to incorrect values determined for the vgg19 earlier on, or differences in GPUs that were used in runtime compared to when the Nvidia Tesla K80 was first realized as the Colab GPU. For the most part, achieved performance was distributed in varying degrees of smallness compared to the expected peak performance.

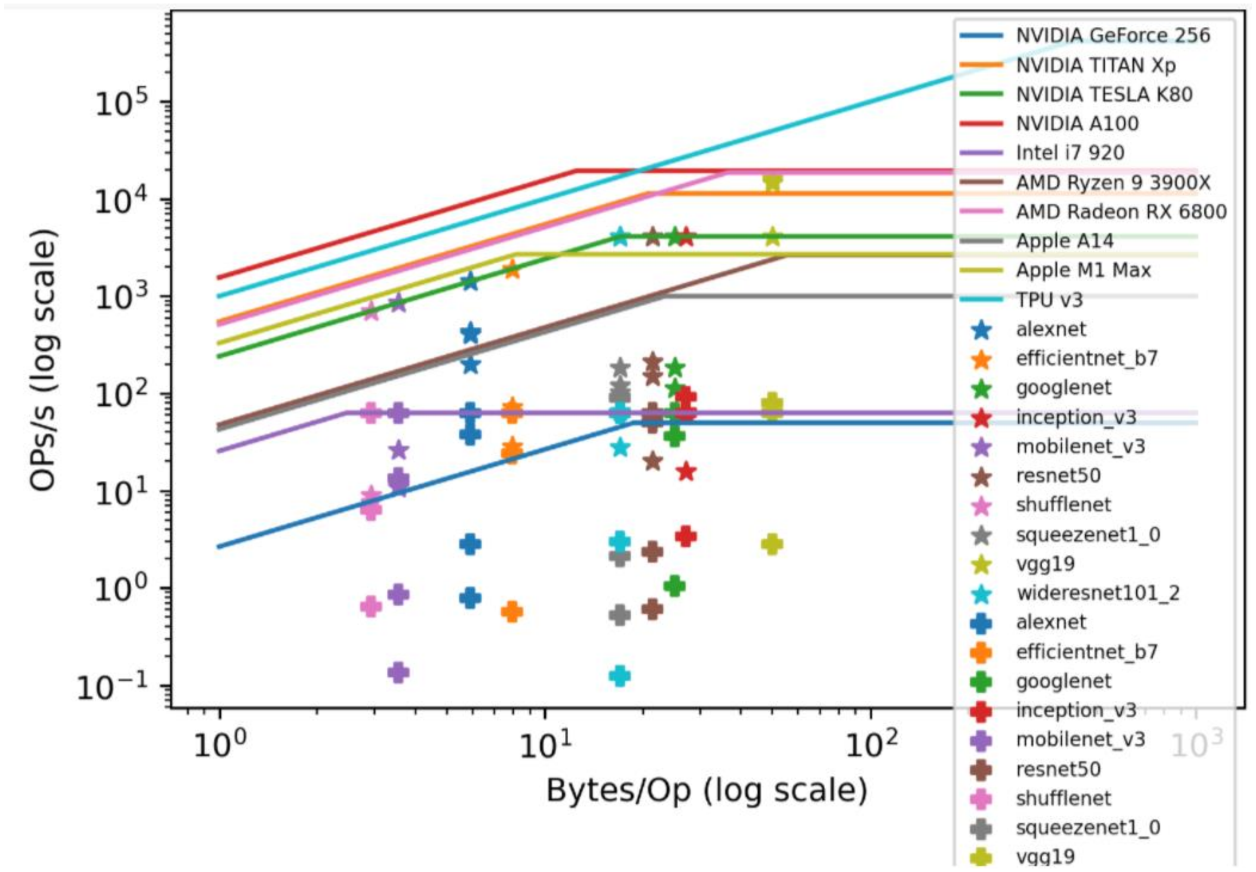
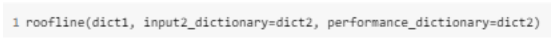


Figure 11: Roofline Plot for Q4 with actual performance added onto the plot

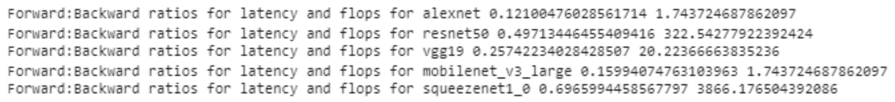


```
1 roofline(dict1, input2_dictionary=dict2, performance_dictionary=dict2)
```

Figure 12: Code used to produce Q4 – again note that most changes were additions to the original function and can be observed in Figure 2

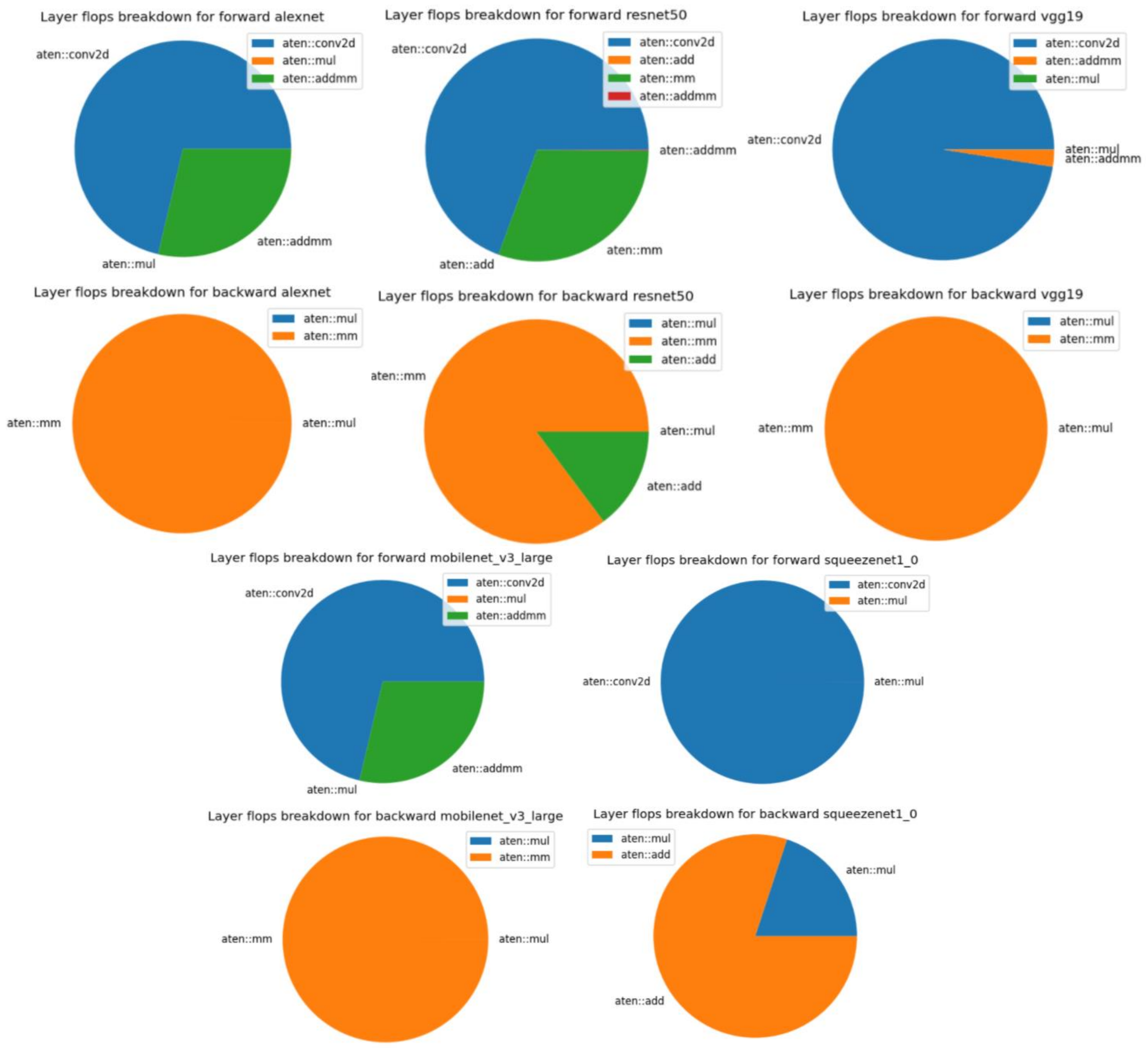
Question 5:

Based on the results of the 5 indicated DNNs that were tested, the ratio of latency for forward to backward pass can range anywhere from an approximately 2:3 relationship with squeezenet to an approximately 1:8 ratio as seen in alexnet. The flops vary drastically from 1.74:1 all the way to 3866:1. None of the flop ratios experimentally verify the 1:2 ratio that was expected based on the description from lecture.

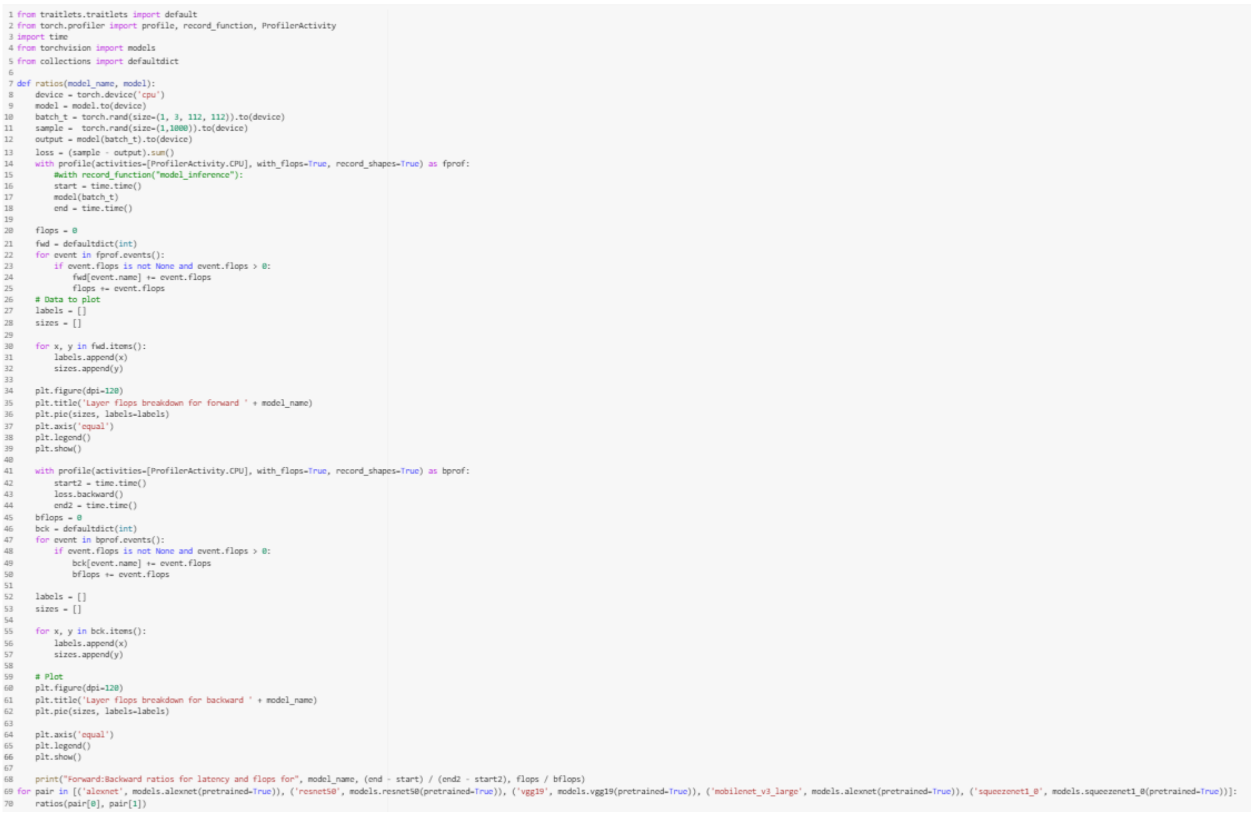


```
Forward:Backward ratios for latency and flops for alexnet 0.12100476028561714 1.743724687862097
Forward:Backward ratios for latency and flops for resnet50 0.49713446455409416 322.54277922392424
Forward:Backward ratios for latency and flops for vgg19 0.25742234028428507 20.22366663835236
Forward:Backward ratios for latency and flops for mobilenet_v3_large 0.15994074763103963 1.743724687862097
Forward:Backward ratios for latency and flops for squeezenet1_0 0.6965994458567797 3866.176504392086
```

Figure 13: Examples of latency and flop forward pass to backward pass ratios for 5 DNNs



Figures 14-18: Pie Charts showing the proportion of flops from various layers for the forward and backward passes of 5 DNNs



```python
from traitlets.traitlets import default
from torch.profiler import profile, record_function, ProfilerActivity
import time
from torchvision import models
from collections import defaultdict

def ratios(model_name, model):
    device = torch.device('cpu')
    model = model.to(device)
    batch_t = torch.rand(size=(1, 3, 112, 112)).to(device)
    sample =  torch.rand(size=(1,1000)).to(device)
    output = model(batch_t).to(device)
    loss = (sample - output).sum()
    with profile(activities=[ProfilerActivity.CPU], with_flops=True, record_shapes=True) as fprof:
        #with record_function("model_inference"):
        start = time.time()
        model(batch_t)
        end = time.time()

    flops = 0
    fwd = defaultdict(int)
    for event in fprof.events():
        if event.flops is not None and event.flops > 0:
            fwd[event.name] += event.flops
            flops += event.flops
    # Data to plot
    labels = []
    sizes = []

    for x, y in fwd.items():
        labels.append(x)
        sizes.append(y)

    plt.figure(dpi=120)
    plt.title('Layer flops breakdown for forward ' + model_name)
    plt.pie(sizes, labels=labels)
    plt.axis('equal')
    plt.legend()
    plt.show()

    with profile(activities=[ProfilerActivity.CPU], with_flops=True, record_shapes=True) as bprof:
        start2 = time.time()
        loss.backward()
        end2 = time.time()
    bflops = 0
    bck = defaultdict(int)
    for event in bprof.events():
        if event.flops is not None and event.flops > 0:
            bck[event.name] += event.flops
            bflops += event.flops

    labels = []
    sizes = []

    for x, y in bck.items():
        labels.append(x)
        sizes.append(y)

    # Plot
    plt.figure(dpi=120)
    plt.title('Layer flops breakdown for backward ' + model_name)
    plt.pie(sizes, labels=labels)

    plt.axis('equal')
    plt.legend()
    plt.show()

    print("Forward:Backward ratios for latency and flops for", model_name, (end - start) / (end2 - start2), flops / bflops)
for pair in [('alexnet', models.alexnet(pretrained=True)), ('resnet50', models.resnet50(pretrained=True)), ('vgg19', models.vgg19(pretrained=True)), ('mobilenet_v3_large', models.alexnet(pretrained=True)), ('squeezenet1_0', models.squeezenet1_0(pretrained=True))]:
    ratios(pair[0], pair[1])
```

Figure 19: Code used to generate ratios and plots for Q5