Problem3

April 19, 2022

1 Problem 3 - PALEO, FLOPs, Platform Percent of Peak (PPP)

This question is based on modeling the execution time of deep learning networks by calculating the floating point operations required at each layer. We looked at two papers in the class, one by Lu et al. and the other by Qi et al.

1.1 1

Why achieving peak FLOPs from hardware devices like GPUs is a difficult propostion in real systems ? How does PPP help in capturing this inefficiency captured in Paleo model. (4)

Peak FLOPs is a difficult proposition in real systems: - usually requiring customized libraries developed by organizations with intimate knowledge of the underlying hardware - any computation done outside the scope of PALEO (e.g. job scheduling, data copying) will exacerbate the observed inefficiency in practice.

PPP help in capturing this inefficiency captured in Paleo model: Parameter which captures the average relative inefficiency of the platform compared to peak FLOPS. Highly specialized frameworks (e.g. cuDNN) will in general have a computational PPP that is close to 100%, while frameworks with higher overheads may have PPP constants closer to 50% or less.

In other works, PPP can help us evaluate how much time the computation is at their peak speed.

1.2 2

Lu et al. showed that FLOPs consumed by convolution layers in VG16 account for about 99% of the total FLOPS in the forward pass. We will do a similar analysis for VGG19. Calculate FLOPs for different layers in VGG19 and then calculate fraction of the total FLOPs attributed by convolution layers. (6)

[4]:
```
import torch
from torchvision.models import vgg16
from pthflops import count_ops

# Create a network and a corresponding input
device = 'cuda:0'
model = vgg16().to(device)
inp = torch.rand(1,3,224,224).to(device)
```

[5]:
```
model
```

[5]:
```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace=True)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace=True)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace=True)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
```

```
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

[44]:
```
ignore_list = []
for i, x in enumerate(model.features):
    name = "features_" + str(i)
    if not isinstance(x, torch.nn.modules.conv.Conv2d):
        ignore_list.append(name)

for i, x in enumerate(model.classifier):
    name = "classifier_" + str(i)
    if not isinstance(x, torch.nn.modules.conv.Conv2d):
        ignore_list.append(name)
```

[45]:
```
ignore_list
```

[45]:
```
['features_1',
 'features_3',
 'features_4',
 'features_6',
 'features_8',
 'features_9',
 'features_11',
 'features_13',
 'features_15',
 'features_16',
 'features_18',
 'features_20',
 'features_22',
 'features_23',
 'features_25',
 'features_27',
 'features_29',
 'features_30',
 'classifier_0',
 'classifier_1',
 'classifier_2',
 'classifier_3',
 'classifier_4',
 'classifier_5',
 'classifier_6']
```

[46]:
```
conv_res = count_ops(model, inp, ignore_layers=ignore_list)
total_res = count_ops(model, inp)
print(conv_res, total_res)
```

```
Operation     OPS
------------  -----------
features_0    89915392
features_2    1852899328
features_5    926449664
features_7    1851293696
features_10   925646848
features_12   1850490880
features_14   1850490880
features_17   925245440
features_19   1850089472
features_21   1850089472
features_24   462522368
features_26   462522368
features_28   462522368
avgpool       1229312
-----------   ----------
Input size: (1, 3, 224, 224)
15,361,407,488 FLOPs or approx. 15.36 GFLOPs
Operation      OPS
-------------  -----------
features_0     89915392
features_1     6422528
features_2     1852899328
features_3     6422528
features_4     2408448
features_5     926449664
features_6     3211264
features_7     1851293696
features_8     3211264
features_9     1204224
features_10    925646848
features_11    1605632
features_12    1850490880
features_13    1605632
features_14    1850490880
features_15    1605632
features_16    602112
features_17    925245440
features_18    802816
features_19    1850089472
features_20    802816
features_21    1850089472
features_22    802816
features_23    301056
features_24    462522368
features_25    200704
features_26    462522368
```

```
features_27    200704
features_28    462522368
features_29    200704
features_30    75264
avgpool        1229312
classifier_0   102764544
classifier_1   8192
classifier_3   16781312
classifier_4   8192
classifier_6   4097000
------------   ----------
Input size: (1, 3, 224, 224)
15,516,752,872 FLOPs or approx. 15.52 GFLOPs
(15361407488, [['features_0', 89915392], ['features_2', 1852899328],
['features_5', 926449664], ['features_7', 1851293696], ['features_10',
925646848], ['features_12', 1850490880], ['features_14', 1850490880],
['features_17', 925245440], ['features_19', 1850089472], ['features_21',
1850089472], ['features_24', 462522368], ['features_26', 462522368],
['features_28', 462522368], ['avgpool', 1229312]]) (15516752872, [['features_0',
89915392], ['features_1', 6422528], ['features_2', 1852899328], ['features_3',
6422528], ['features_4', 2408448], ['features_5', 926449664], ['features_6',
3211264], ['features_7', 1851293696], ['features_8', 3211264], ['features_9',
1204224], ['features_10', 925646848], ['features_11', 1605632], ['features_12',
1850490880], ['features_13', 1605632], ['features_14', 1850490880],
['features_15', 1605632], ['features_16', 602112], ['features_17', 925245440],
['features_18', 802816], ['features_19', 1850089472], ['features_20', 802816],
['features_21', 1850089472], ['features_22', 802816], ['features_23', 301056],
['features_24', 462522368], ['features_25', 200704], ['features_26', 462522368],
['features_27', 200704], ['features_28', 462522368], ['features_29', 200704],
['features_30', 75264], ['avgpool', 1229312], ['classifier_0', 102764544],
['classifier_1', 8192], ['classifier_3', 16781312], ['classifier_4', 8192],
['classifier_6', 4097000]])
```

[47]:
```python
ans = "%.2f" % (conv_res[0] / total_res[0] * 100)
print(f"The percentage of FLOPs attributed by convolution layers in VGG19 is␣
↪{ans}% for the forward pass.",)
```

```
The percentage of FLOPs attributed by convolution layers in VGG19 is 99.00% for
the forward pass.
```

1.3 3

Study the tables showing timing benchmarks from Alexnet (Table 2), VGG16 (Table 3), Googlenet (Table 5), and Resnet50 (Table 6). Why the measured time and sum of layerwise timings for forward pass did not match on GPUs ? What approach was adopted in Sec. 5 of the paper to mitigate the measurement overhead in GPUs. (2+2)

Why the measured time and sum of layerwise timings for forward pass did not match on GPUs?

The reason for the mismatch is that CUDA supports asynchronous programming. Before time measurement, an API (cudaDeviceSynchronize) has to be called to make sure that all cores have finished their tasks. is explicit synchronization is the overhead of measuring time on the GPUs.

In other words, when we calculate the layer-wise time, CUDA need addtitionnal call which make the measured time larger than the real case.

What approach was adopted in Sec. 5 of the paper to mitigate the measurement overhead in GPUs?

Transforming the computation into matrix multiplication, which can be accelerately measured by BLAS and cuBLAS libraries. Thus we have a way to accurately approximate and measure the calculation time.

1.4 4

In Lu et al. FLOPs for different layers of a DNN are calculated. Use FLOPs numbers for VGG16 (Table 3), Googlenet (Table 5), and Resnet50 (Table 6), and calculate the inference time (time to have a forward pass with one image) using published Tflops number for K80 (Refer to NVIDIA TESLA GPU Accelerators) both for single-precision and double-precision calculations. Use this to calculate the peak (theoretical) throughput achieved with K80 for these 3 models. (6)

1.4.1 VGG16

[14]:
```python
import sys
import os
def get_num(num_str):
    res = os.popen(f"numfmt --from si {num_str}").read()
    res = int(res)
    # res = os.system()
    return res
# print(res)
vgg16_total_flops = get_num("15503M")
googlenet_total_flops = get_num("1606M")
resnet_total_flops = get_num("3922M")



k80_single_flops = get_num("8.73T")
k80_double_flops = get_num("2.91T")

vgg16_single_time = vgg16_total_flops / k80_single_flops
vgg16_double_time = vgg16_total_flops / k80_double_flops

googlenet_single_time = googlenet_total_flops / k80_single_flops
googlenet_double_time = googlenet_total_flops / k80_double_flops

resnet_single_time = resnet_total_flops / k80_single_flops
resnet_double_time = resnet_total_flops / k80_double_flops
```

```python
vgg16_single_throughput = 1 / vgg16_single_time
vgg16_double_throughput = 1 / vgg16_double_time

googlenet_single_throughput = 1 / googlenet_single_time
googlenet_double_throughput = 1 / googlenet_double_time

resnet_single_throughput = 1 / resnet_single_time
resnet_double_throughput = 1 / resnet_double_time

print(f"The single-GPU inference time for VGG16 is {vgg16_single_time}s, and the throughput is {vgg16_single_throughput}.")
print(f"The double-GPU inference time for VGG16 is {vgg16_double_time}s, and the throughput is {vgg16_double_throughput}.")
print(f"The single-GPU inference time for GoogLeNet is {googlenet_single_time}s, and the throughput is {googlenet_single_throughput}.")
print(f"The double-GPU inference time for GoogLeNet is {googlenet_double_time}s, and the throughput is {googlenet_double_throughput}.")
print(f"The single-GPU inference time for ResNet is {resnet_single_time}s, and the throughput is {resnet_single_throughput}.")
print(f"The double-GPU inference time for ResNet is {resnet_double_time}s, and the throughput is {resnet_double_throughput}.")
```

```
The single-GPU inference time for VGG16 is 0.0017758304696449026s, and the
throughput is 563.1168161001096.
The double-GPU inference time for VGG16 is 0.005327491408934708s, and the
throughput is 187.7056053667032.
The single-GPU inference time for GoogLeNet is 0.00018396334478808707s, and the
throughput is 5435.865504358655.
The double-GPU inference time for GoogLeNet is 0.0005518900343642612s, and the
throughput is 1811.9551681195517.
The single-GPU inference time for ResNet is 0.0004492554410080183s, and the
throughput is 2225.9051504334525.
The double-GPU inference time for ResNet is 0.001347766323024055s, and the
throughput is 741.9683834778174.
```