Wayne Zhu
11/05/2022

Steps to Replicate:

. Create soft links to sqf file.
n -s /scratch/work/public/imagenet/imagenet10k_eccv2010.sqf imagenetImage10k.sqf

2. Join a GPU server for test run. Example:
srun --account=csci_ga_3033_085-2022fa --partition=c12m85-a100-1 --gres=gpu:1 --pty /
bin/bash

3. Collaborating with teammates and eventually found out that the file systems of slurm and
Greene differ. But the good news is that our training data can be found as /share/apps/
datasets/imagenet/imagenet-train.sqf on slurm.

4. Added these line to singularity container launching bash file to ensure that the sqf image
files are uncompressed:

--overlay /share/apps/datasets/imagenet/imagenet-train.sqf:ro \
--overlay /share/apps/datasets/imagenet/imagenet-val.sqf:ro \

5. After launching the singularity container, copy 10 folders of images as training (a total of
3000 files) to tmp, where IO is less of a bottleneck. Originally, there are more than 1
million images taking up 138 GB after uncompressing. That's way too big.

cp -r n04346328 n06359193 \
n01775062 n02088238 n02108915 n02321529 n02749479 \
n03028079 n03459775 n03785016 \tmp/imagenet/train

cp -r n01798484 n02089973 n02110341 n02356798 n02786058 n03062245 n0348117
n03792782 n04044716 n04366367 n07579787 /tmp/imagenet/val

6. Calculate the theoratical memory and flop complexity with the help of pytorch_summary.

7. Calculate actual memory usage with GPUtil profiler.

8. Estimate the theoretical FLOP with pytorch-estimate-flops library.

9. Calculate the actual FLOP with ncu. The metrics are given in lec7 slide.

Note that when using ncu, the effective use of grep command to sort and summarize all
relevant output are extremely important. e.g. grep sum ncuTrial2.out | grep
dram__bytes_read| grep Kbyte|grep -Eo '[-+]?([0-9]*.[0-9]+|[0-9]+)'|awk '{ SUM += $1} END
print SUM }'

10. Calculate the runtime with timer without ncu.

11. Search and write down the peak memory and FLOPS upper bound based on the GPU
hardware.

12. Calculate the actual memory and FLOP usage with ncu's metrics:
dram__bytes_write.sum,dram__bytes_read.sum.

13. Produce the roofline tables and charts.

Experiment design:

For Memory Complexity Comparison, we tried out different numbers of GPU:

1. Resnet 18, 1 Tesla V100-SXM2 GPU
2. Resnet 50, 1 Tesla V100-SXM2 GPU
3. Resnet 18, 2 Tesla V100-SXM2 GPU
4. Resnet 50, 2 Tesla V100-SXM2 GPU

For FLOP Complexity and Roofline Model, we tried out different flavor of GPU:
1. Resnet 18, 1 Tesla V100-SXM2 GPU
2. Resnet 50, 1 Tesla V100-SXM2 GPU
3. Resnet 18, 1 Tesla A100-SXM GPU
4. Resnet 50, 1 Tesla A100-SXM GPU

Note that for the roofline model, trial 1 and 2 share a plot. Trial 3 and 4 share another plot.

Multiple tables will be generated and explained. We use 1 epoch in all training to simplify the design. For the memory analysis, we tried 10 classes of images. But for FLOP and roofline analysis , I used only 10 images because otherwise ncu would take too long.

Hypothesis:

1. I expect actual memory usage to be much higher than the theoretical parameter size because we don't account for overheads in theoretical estimation.

2. I expect the actual FLOP to be of the same order as the theoretical FLOP as in HW4. But still, I expect our theoretical FLOP to be lower than actual because we don't account for overhead.

3. As for roofline modeling, I expect all 4 trials to be communication bound—which is usually the case unless the code are written excellently for parallel computing. Also, our test dataset is small, so computation bound would be extremely unlikely.

4. I expect Resnet50 runtime to be much larger than Resnet18, but I am not sure if 2 GPU will necessarily outperform 1 GPU.

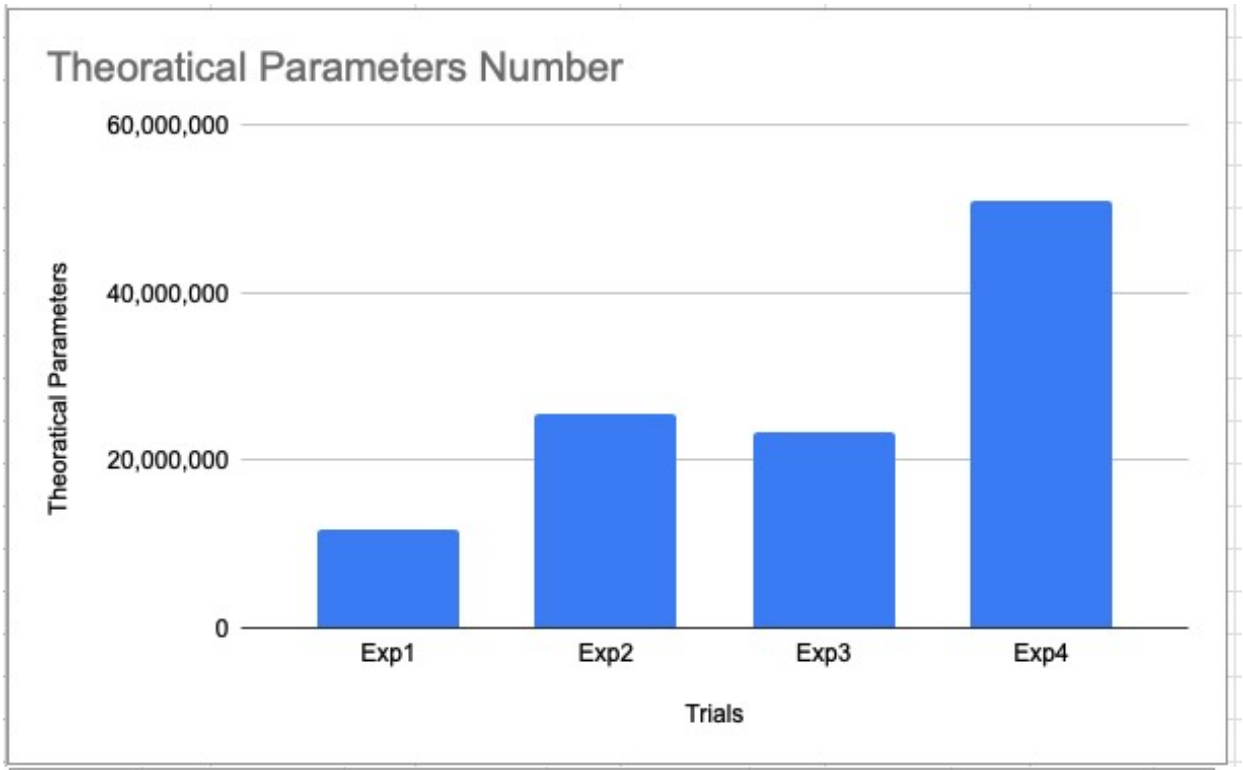
5. I expect a100 GPU to outperform v100.

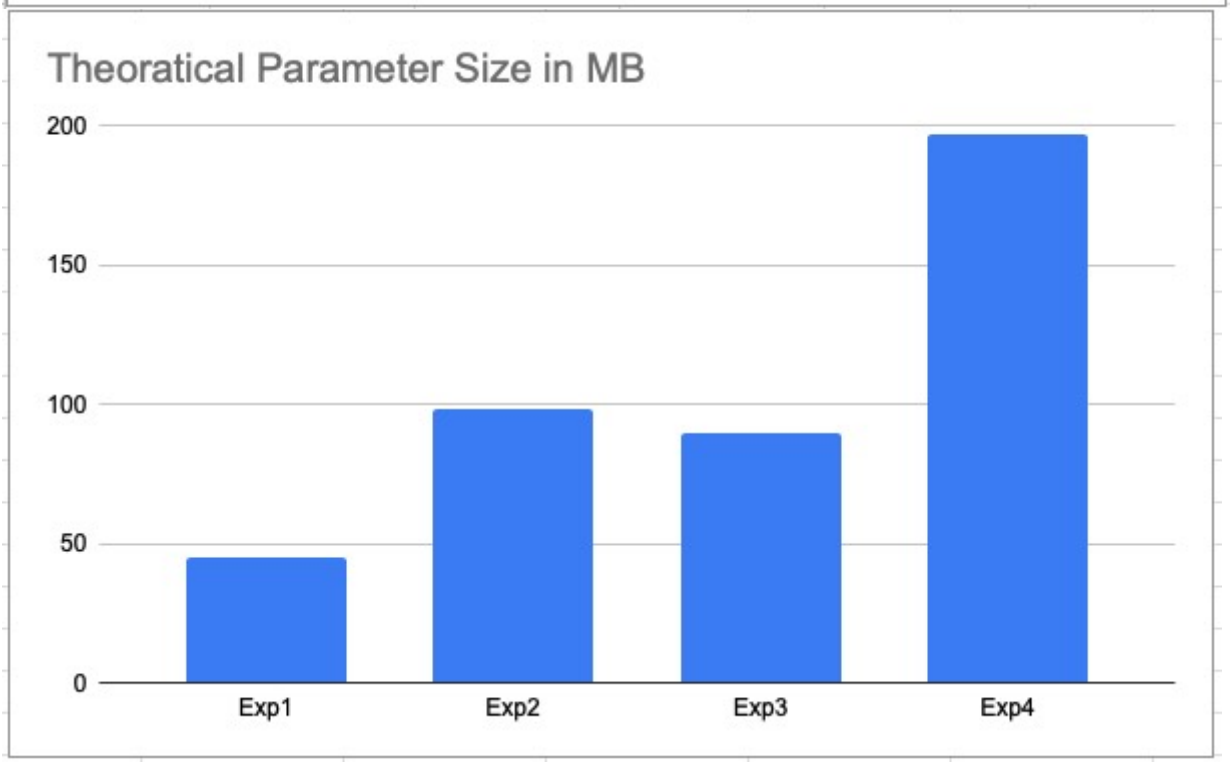
Comparing Memory Usage:

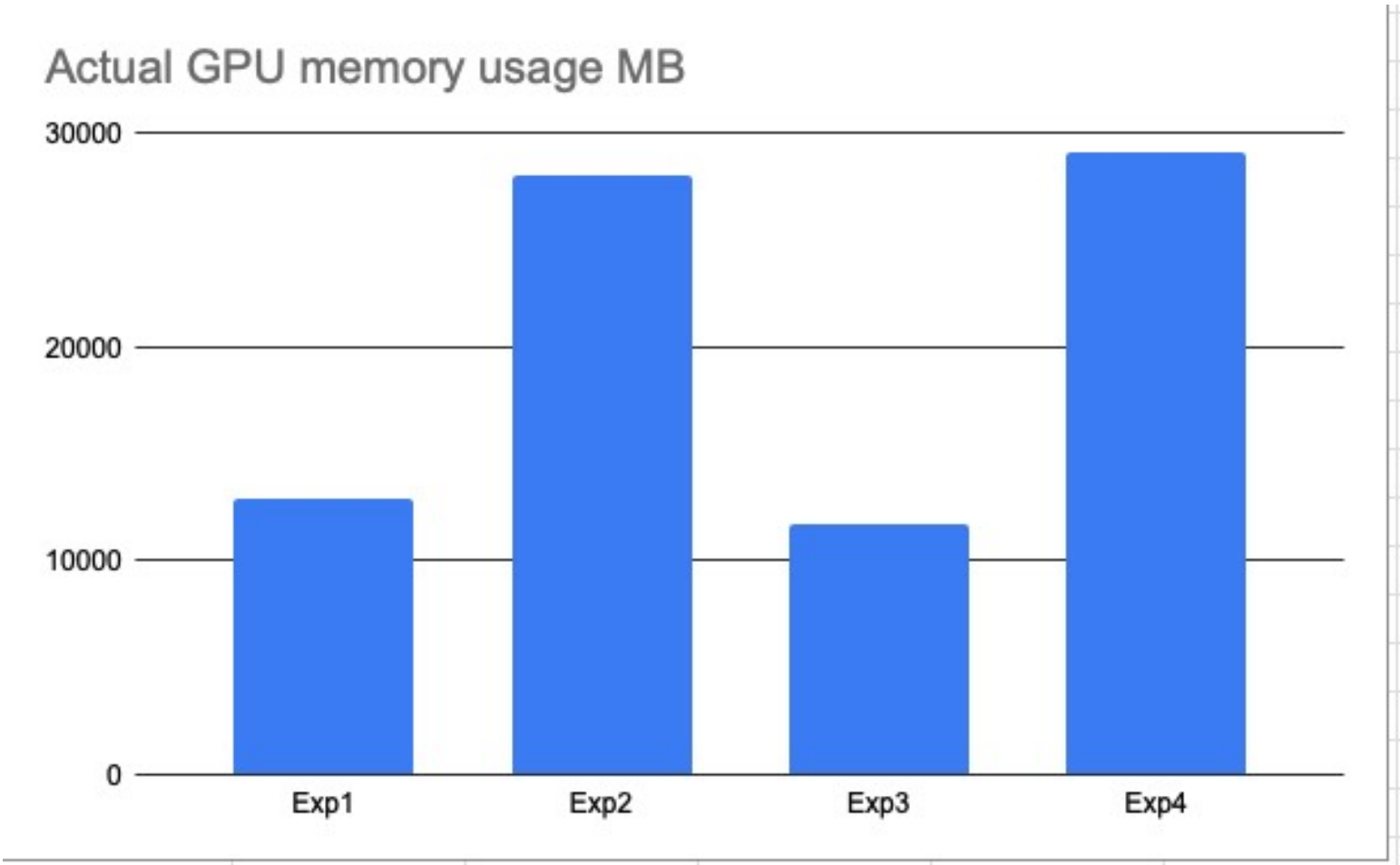
The pytorch summary gives the estimated number of parameters in the model, and we multiply it by 4 (float size)to get the theoretical memory.
This is the same value as given by the pytorch official website for resnet of certain size.

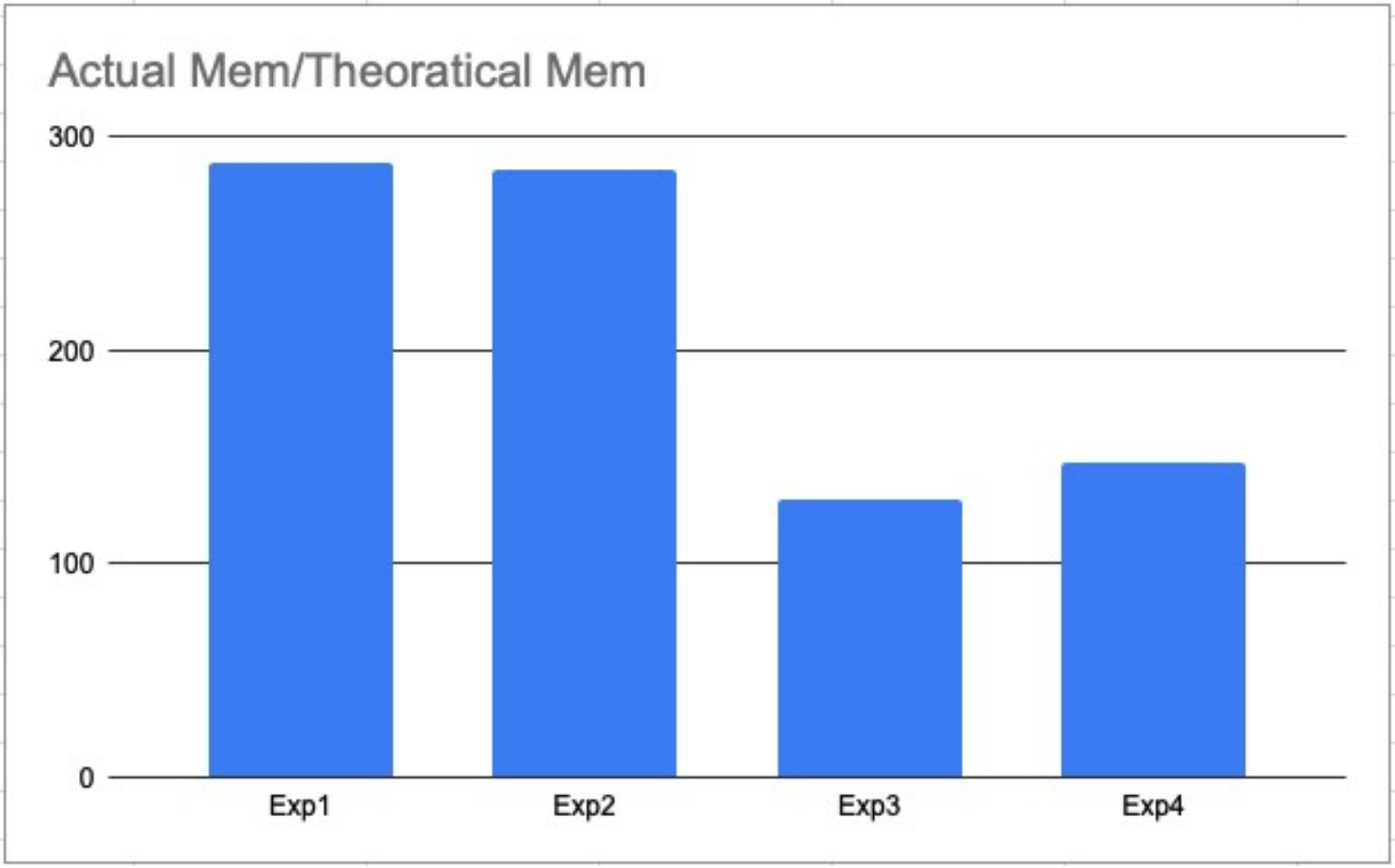
The data is as followed. Each column is also represented as a chart:

Trials	Theoratical Para	Theoratical par s	Actual GPU mem	Actual/Theoratical
Exp1	11,689,512	44.77	12882	287.7373241
Exp2	25,557,032	98.36	28000	284.6685645
Exp3	23,379,024	89.55	11688	130.519263
Exp4	51,114,064	196.72	29108	147.9666531






Actual GPU memory usage MB
30000
20000
10000
0
Exp1
Exp2
Exp3
Exp4


Actual Mem/Theoratical Mem
300
200
100
0
Exp1
Exp2
Exp3
Exp4

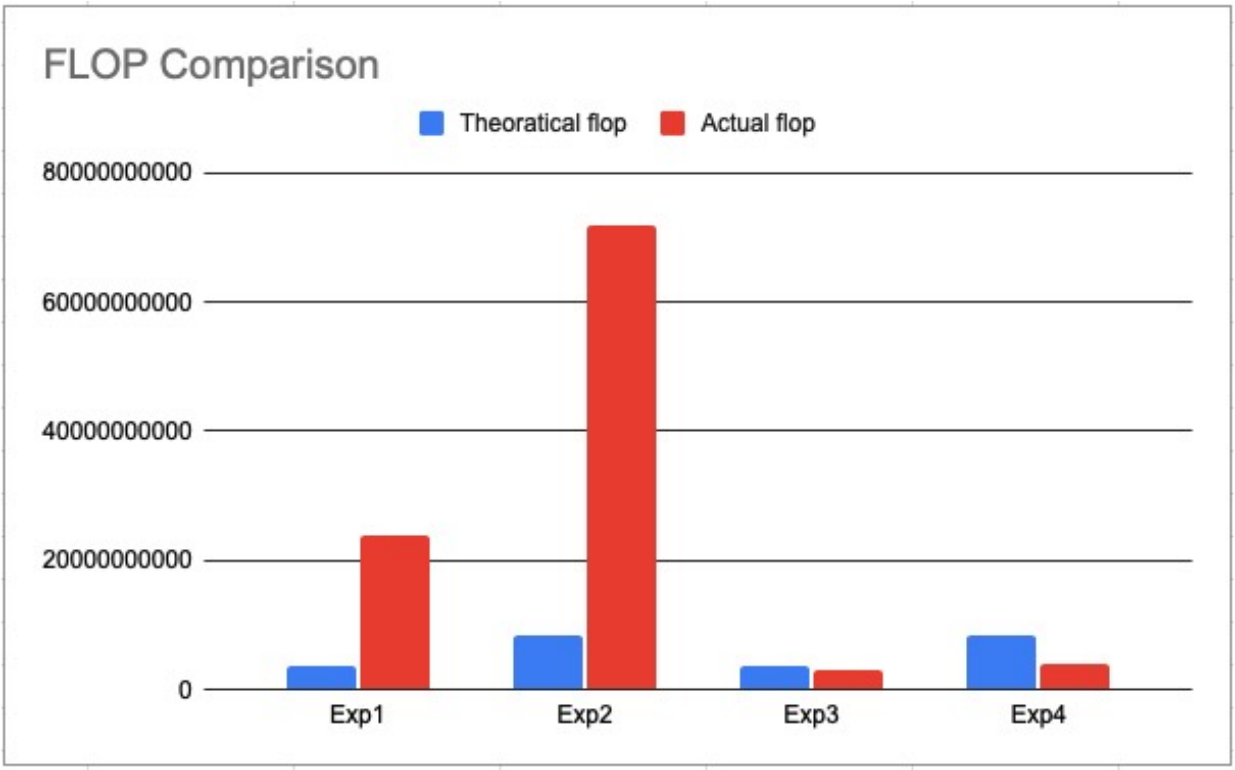
Discussion:
Graph 1 and 2 are linearly proportional, the only difference is the units.
We can clearly see that theoretically Resnet50's number of parameters is double that of Resnet18's. Also, pytorch profiler predicts that 2 GPU will have 2 times the parameters, which is not the case in reality.

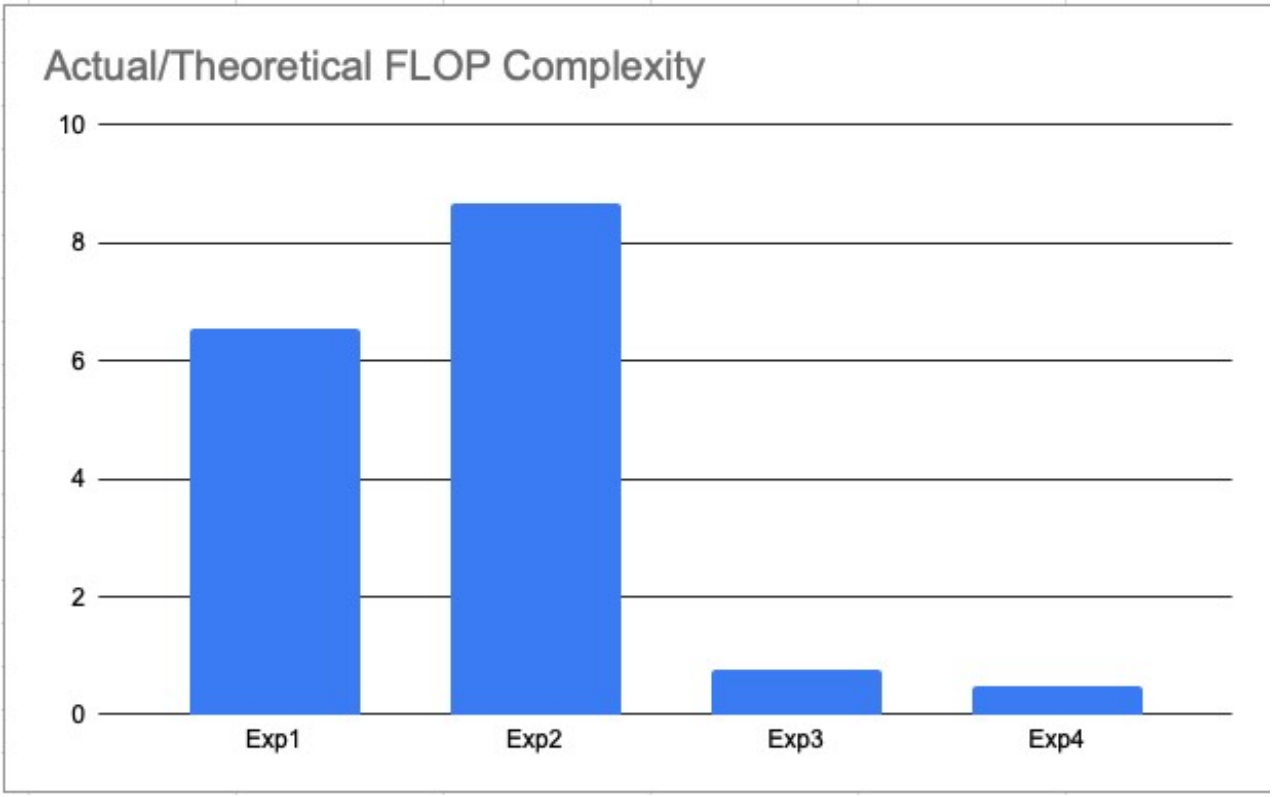
But the actual memory is about 130 to 290 times the theoretical prediction. There are a lot of reasons for this discrepency, and the primary reason is that our model is quite complex. Therefore, in order to execute efficiently pytorch definitely traded space for time, so we are not only storing the parameters but many other communication overheads. As indicated in the lecture slide, those extra operations that take space would be: memory for layer outputs, errors, parameter gradients, and hyperparameter as well as implementation overhead. Actual memory being more than 290 times the theoretical memory agrees with hw4 where a simpler model yields a 100 times memory difference between estimation and reality.

The actual memory is proportional to the theoretical memory when the GPU number is the same. But when we double the number of GPUs, we don't actually double the memory. Clearly, pytorch is doing sth in the background to increase communication in order to avoid duplicating data. The fact that 2 GPUs used roughly the same amount of total memory as 1 GPU further supported this idea of increased communication and constant memory.

This table shows the GPU FLOP comparison:

Trials	Theoratical flop	Actual flop	Actual/Theoratical
Exp1	3,653,586,920	23851649316	6.528282983
Exp2	8,286,749,672	71858381540	8.671479698
Exp3	3,653,586,920	2843954468	0.7784006595
Exp4	8,286,749,672	4014655508	0.4844668497





Recall that trial 1 and 2 uses v100 GPU, and trial 3 and 4 uses a100 GPU. Notice that both GPU's actual computation is the same order as the theoretical calculation. The older V100 GPU has a larger overhead compared to the later a100 GPU. The a100 GPU's flop is very close or even lower to the actual flop count.

A100 finishes the task with fewer than expected number of FLOP. The majority of the decrease of FLOP comes from the decrease of multiply+add. It would not be surprising if NVIDIA's latest GPU like a100 optimizes multiply+add on a hardware level.

Resnet50 remains more computationally intensive than Resnet18 in all cases.

Roofline:

We followed Prof Yu's instruction to conduct roofline experiment.

The theoretical peak FLOPS for Tesla V100 for 32 bytes floating point is 15.7 Teraflop/second according to NVidia's own calculation. This is the computation roofline: 1.57*10^13 FLOPS=1.57*10^4 GFLOPS.

https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

The peak bandwidth between GPU-CPU using NVLink is 300 GB/sec. The peak bandwidth between GPU memory and multiprocessors is 900 GB/sec. By prof's instruction, it's clear that communication between GPU and global memory, 900GB/s, is what is asked.

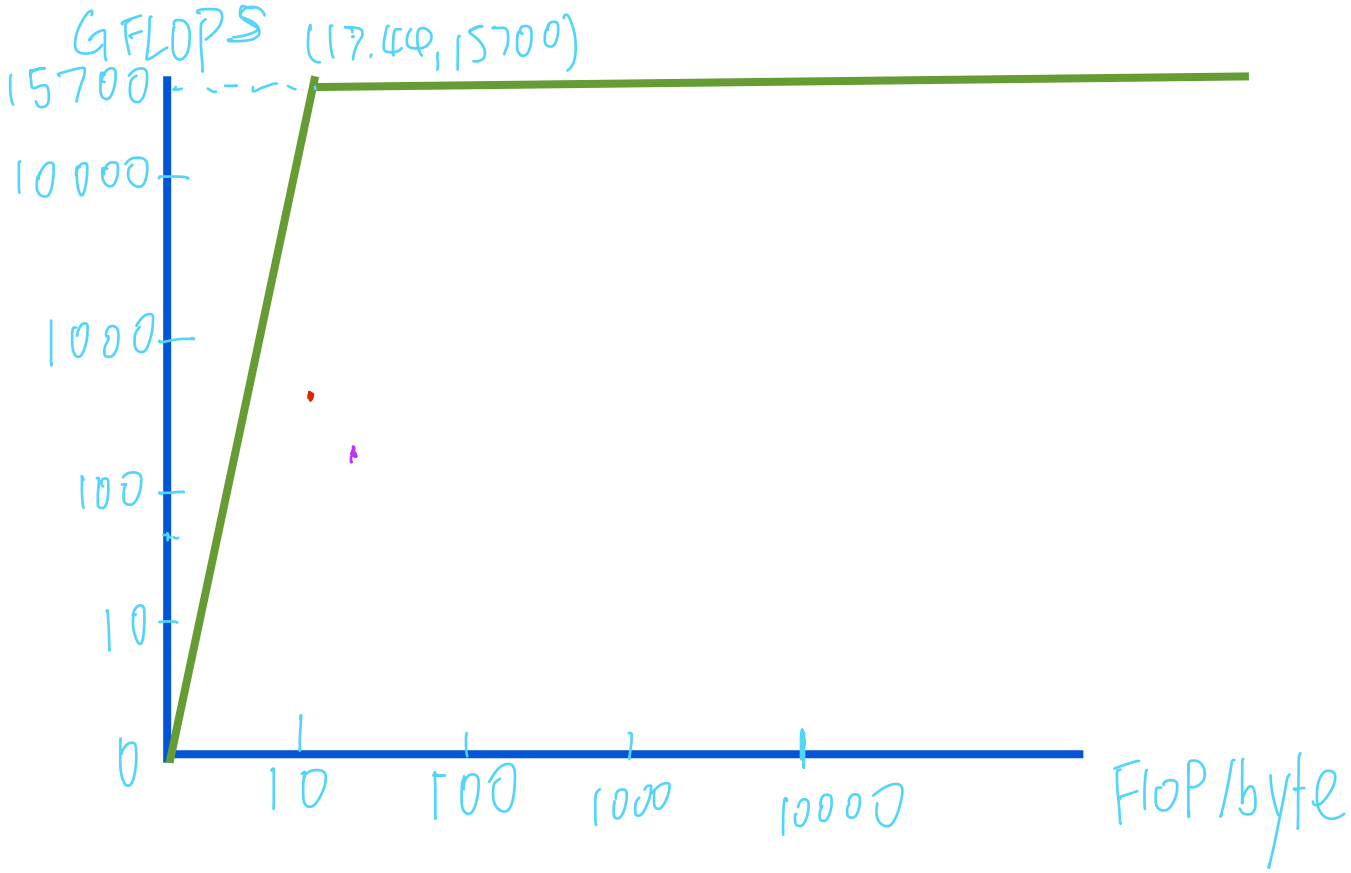
Now let's construct the green roofline: Assuming (0, random reasonable number) is the first point on the roofline. Point 2 is 1.57*10^13FLOP/s / 9*10^11Byte/s=17.44 FLOP/byte So (17.44, 15.7*10^4 GFLOPS) is the second point. Naturally, the third point is (infinity, 15.7*10^4 GFLOPS). So we get the upper bound graph below in green line for trial 1 and 2. The table is also listed here.

Trials	Time	Total Byte	ThroughPut (b/s)	FLOP	FLOP/Byte	GFLOPS
Exp1	0.202907231	1086870000	5356487271	23851649316	21.94526421	117.5495284
Exp2	0.211992785	3651000000	17222284240	71858381540	19.68183554	338.9661659
Exp3	0.223984715	639520000	2855194829	2843954468	4.447014117	12.69709171
Exp4	0.229978666	2787000000	12118515376	4014655508	1.440493544	17.45664317

From this graph, we see that trial 1's point is (22, 117)
Trial 2's point is (20, 339)

• in graph

• in graph



This outcome shows that our application is closer to be memory bound.

This makes sense because the problem size is so small since we are only using very limited number of images to speed up, so it's not likely computation bound.

Also, Nvidia's theoretical roofline is vastly exaggerated to make the products look good. They assume perfect cache hit, and those things are not possible in reality.

For trial 3 and 4, we know the theoretical peak FLOP is 19.5 TFLOPS=1.95*10^4 GFLOPS from NVIDIA's brochure: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-us-nvidia-1758950-r4-web.pdf

Bandwidth is 1,555GB/s.

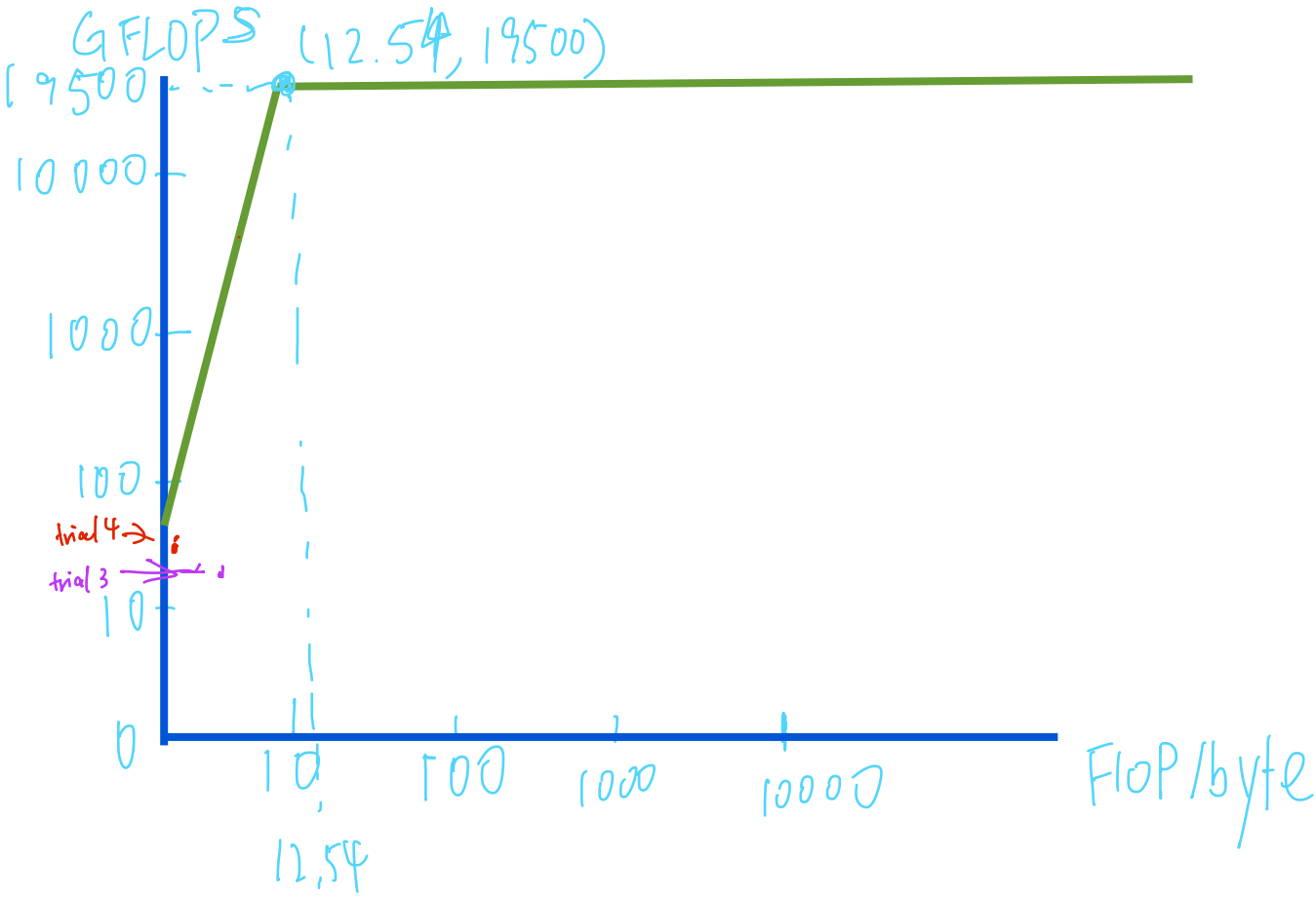
Therefore, for our roofline green line, we assume (0, some reasonable number), to be point 1.

Point two would be at x axis (1.95*10^13/(1555*10^9))=12.540. So (12.54, 1.95*10^4) is our 2nd point.

Point 3 would be (infinity, 1.95*10^4)

For trial 3, the purple point is at (4.45, 12.70).

For trial 4, the red point is at (1.44, 17.50)



Trial 3 and trial 4 result can be derived from the last 2 rows of the following table:

Trials	Time	Total Byte	ThroughPut (b/s)	FLOP	FLOP/Byte	GFLOPS
Exp1	0.202907231	1086870000	5356487271	23851649316	21.94526421	117.5495284
Exp2	0.211992785	3651000000	17222284240	71858381540	19.68183554	338.9661659
Exp3	0.223984715	639520000	2855194829	2843954468	4.447014117	12.69709171
Exp4	0.229978666	2787000000	12118515376	4014655508	1.440493544	17.45664317

Also, because of the program set-up overhead, the time it takes to finish does not decrease much when we switch GPU. So GFLOPS actually appears to be much lower than expected. Meanwhile, total byte decreases by about 20 to 40 percent, while flop decreases by more than 90 percent. As previously explained, this is likely due to Nvidia's optimization of multiply+add in their latest product.

At least resnet50 is much more computational intensive than resnet18 as expected.

This computation is closer to be memory bound than computation bound as expected. It's hard to tell since the dataset size is too small.

Conclusion and re-evaluation of the hypothesis:

1. As expected, the actual memory usage turns out to be much higher(100 times more) than the theoratical parameter memory size because we didn't account for overhead.

2. The actual FLOP is of the same order as the theoratical FLOP as expected in HW4. The latest GPU, a100, performs the task with much lower number of flop overhead and is very close to the estimated flop.

3. As for roofline modeling, I expect all 4 trials to be communication bound. We see that the points are closer to memory bound than computation bound as expected.

4. Resnet50 runtime, bandwidth and flop are much larger than Resnet18 in all cases.