Aneesh Pandoh

1.

Apple M1 SoC:
Online Specs
Peak Flops: 2.61 TFlops (INT8)
Memory Bandwidth: 67 GB/s

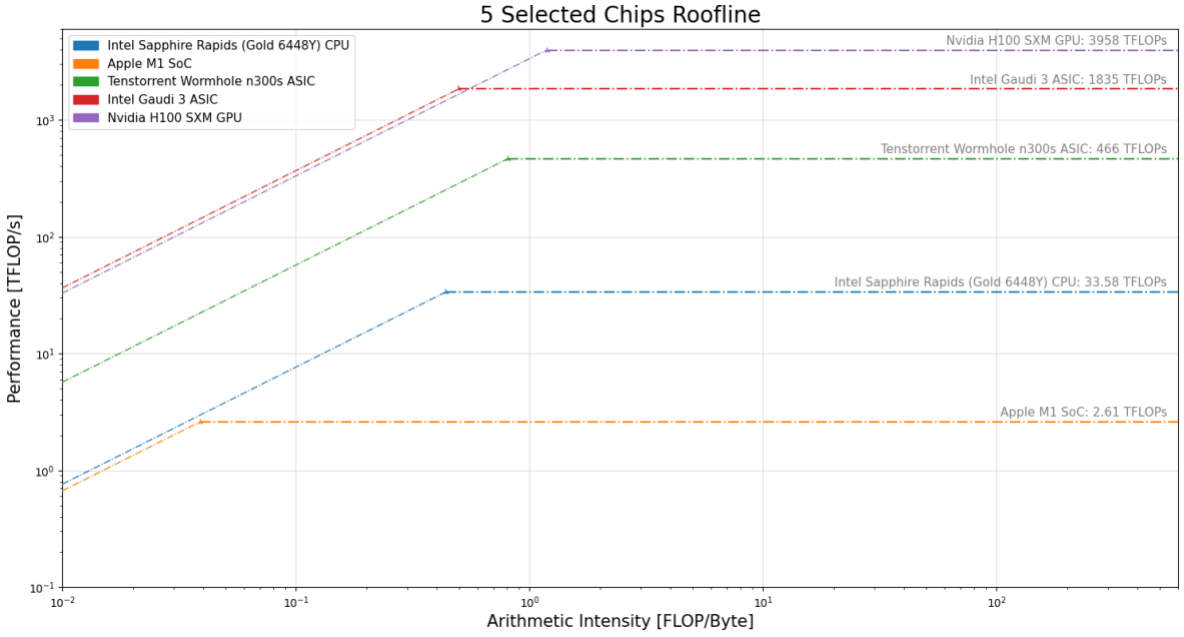
Intel Sapphire Rapids (Gold 6448Y) CPU:
Online Specs
AMX Specs
Peak Flops: 2 Cores x 4.1 GHz x 2048 x 2 (MAC) INT8 per cycle (AMX) = 33.58 TFlops
4400 MT/s x 8 bytes x 2 channels = 76.8 GB/s

Tenstorrent Wormhole n300s ASIC
Online Specs
Peak Flops: 466 TFlops (FP8)
Memory Bandwidth: 576 GB/s

Intel Gaudi 3 ASIC:
Online Specs
Peak Flops: 1835 TFlops (FP8/BF16)
Memory Bandwidth: 3.7 TB/s

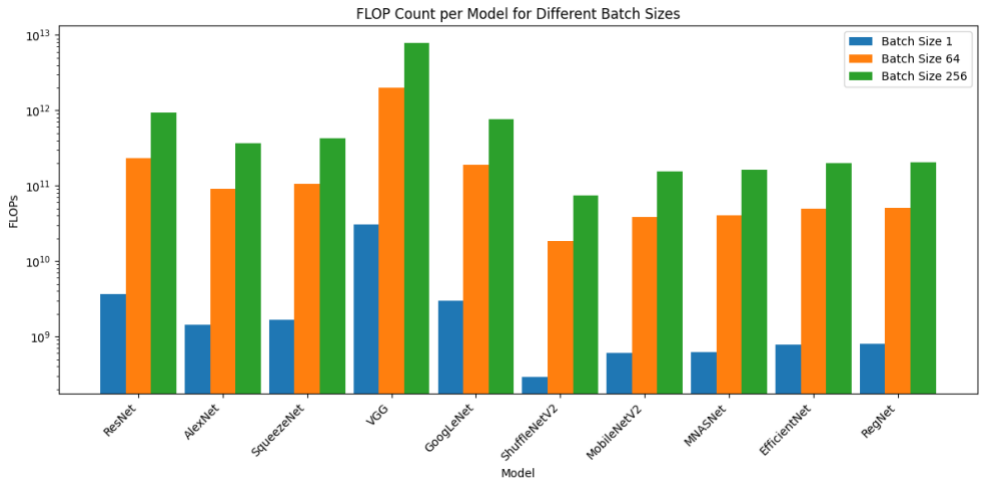
Nvidia H100 SXM GPU:
Online Specs
Peak Flops: 3,958 TFlops (FP8/INT8)
Memory Bandwidth: 3.35 TB/s

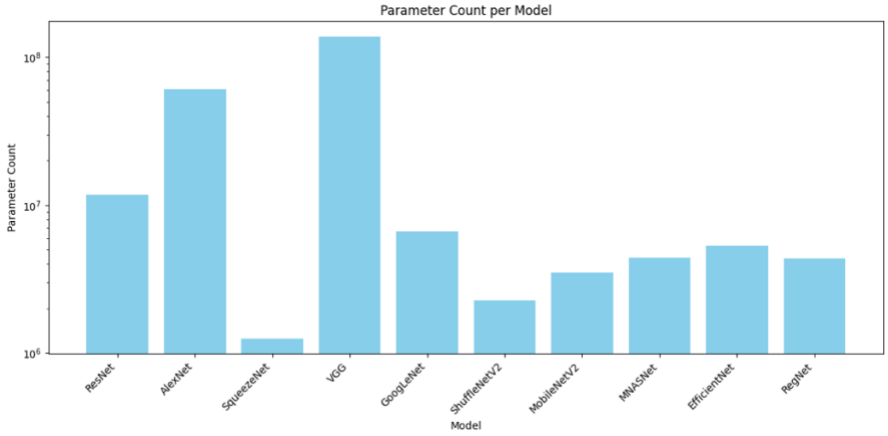
Roofline [Visualization script based of Rooflini repo]:



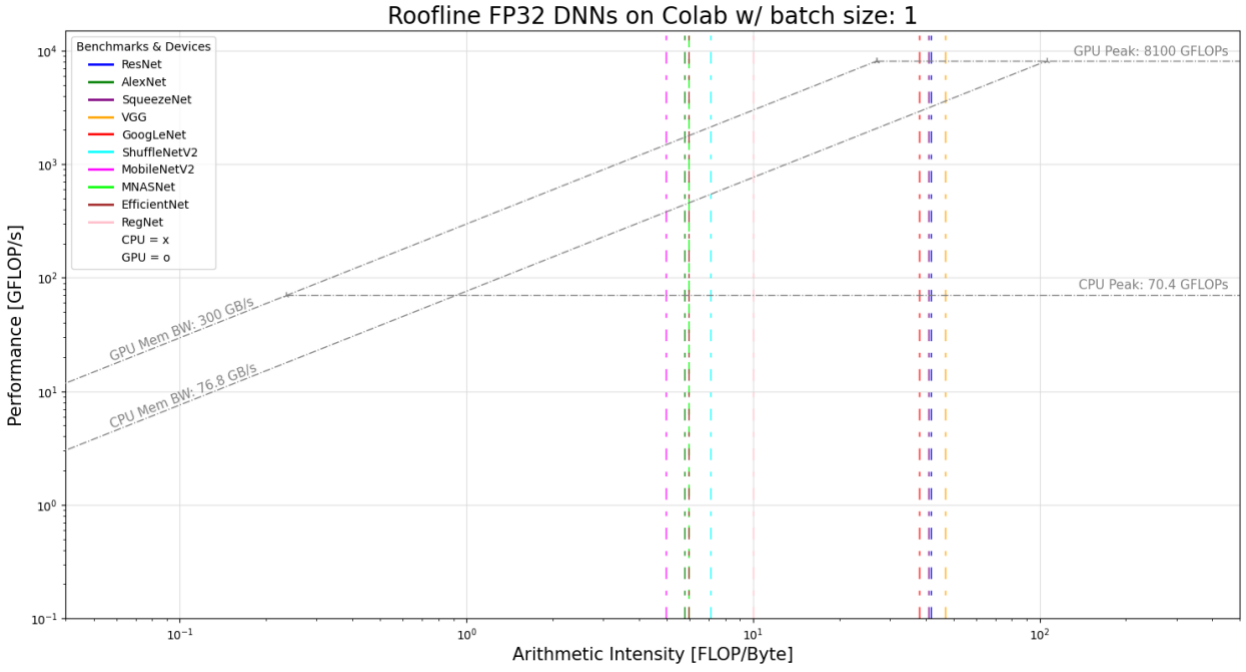
Comparison and Calculation Explanation:
The CPU flop calculation is done by multiplying the amount of cores by the clock speed by the amount of instructions that completed in a single cycle. For AMX, using the supporting information, which demonstrates that it can do 2048 INT8 MAC operations in a matrix multiply in a single cycle, which can then be multiplied by 2 due to the FMA option which combines a MAC into a single operation.
The roofline chart gives a good relative difference between the use cases for these chips, as the M1 is a SoC which all fit into a laptop relative to the other chips which are all meant for the data center compute. We can also see the different between CPUs and GPUs/ASICs as the sapphire rapids chip is orders of magnitude more compute and memory bounds, despite having Advanced Matrix Extensions allowing for matrix multiplication acceleration. Of the 2 AI accelerators we can see the Wormhole is more suited for inference as it does not have the flagship performance need for the extra computation and memory required for the backward pass. We can see the dominance of Nvidia's H100 as the more generalized GPU compared to ASIC of Intel is still considerably further ahead in performance.

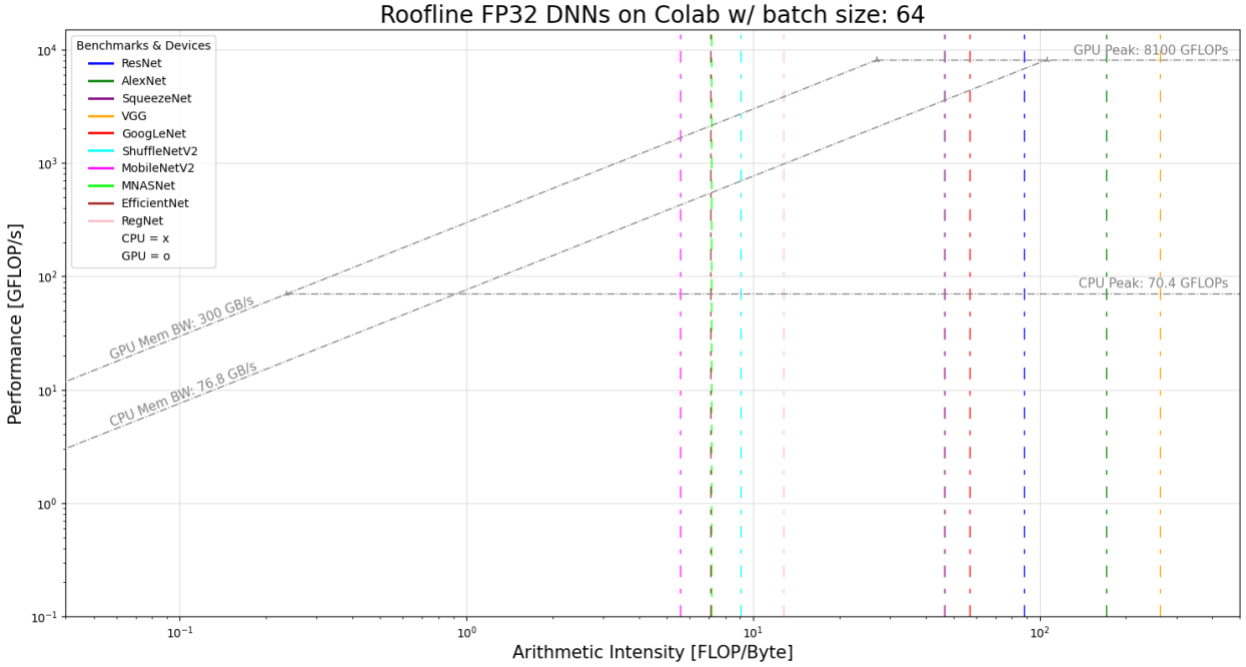
2.1
DNNS chosen from TorchVision: ResNet, AlexNet, SqueezeNet, VGG16, GoogLeNet, ShuffleNetV2, MobileNetV2, MNASNet, EfficientNet, RegNet

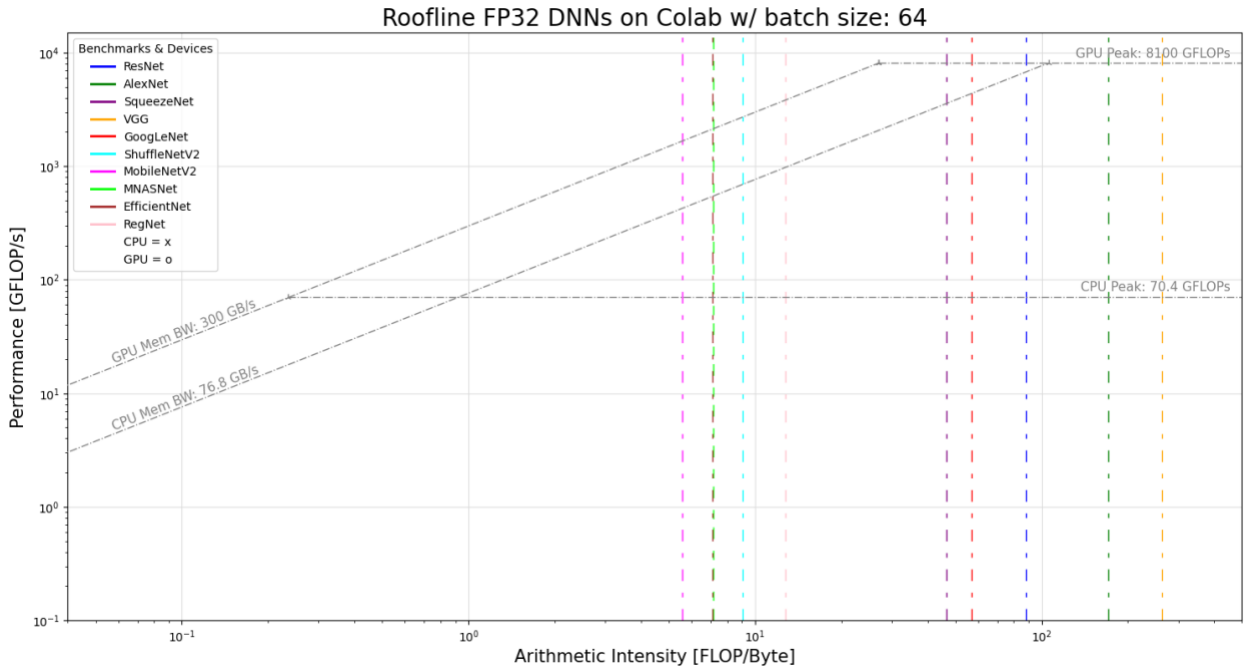
2.2




2.3


Roofline FP32 DNNs on Colab w/ batch size: 1
Benchmarks & Devices
ResNet
AlexNet
SqueezeNet
VGG
GoogLeNet
ShuffleNetV2
MobileNetV2
MNASNet
EfficientNet
RegNet
CPU = x
GPU = o
GPU Peak: 8100 GFLOPs
CPU Peak: 70.4 GFLOPs
GPU Mem BW: 300 GB/s
CPU Mem BW: 76.8 GB/s
Performance [GFLOP/s]
Arithmetic Intensity [FLOP/Byte]


Roofline FP32 DNNs on Colab w/ batch size: 64
Benchmarks & Devices
ResNet
AlexNet
SqueezeNet
VGG
GoogLeNet
ShuffleNetV2
MobileNetV2
MNASNet
EfficientNet
RegNet
CPU = x
GPU = o
GPU Peak: 8100 GFLOPs
CPU Peak: 70.4 GFLOPs
GPU Mem BW: 300 GB/s
CPU Mem BW: 76.8 GB/s
Performance [GFLOP/s]
Arithmetic Intensity [FLOP/Byte]



Colab CPU Roofline Calculation:

```
%cat /proc/cpuinfo
```

CPU Online Specs

2.20 GHz (Colab does not overclock) x 1 core cpu x 32 FP32 FLOPs/cycle (AVX2)

Peak Flops: 70.4 GFlops (FP32)
Memory Bandwidth: 76.8 GB/s

Colab GPU Roofline Calculation:

```
!nvidia-smi
```

GPU Online Specs

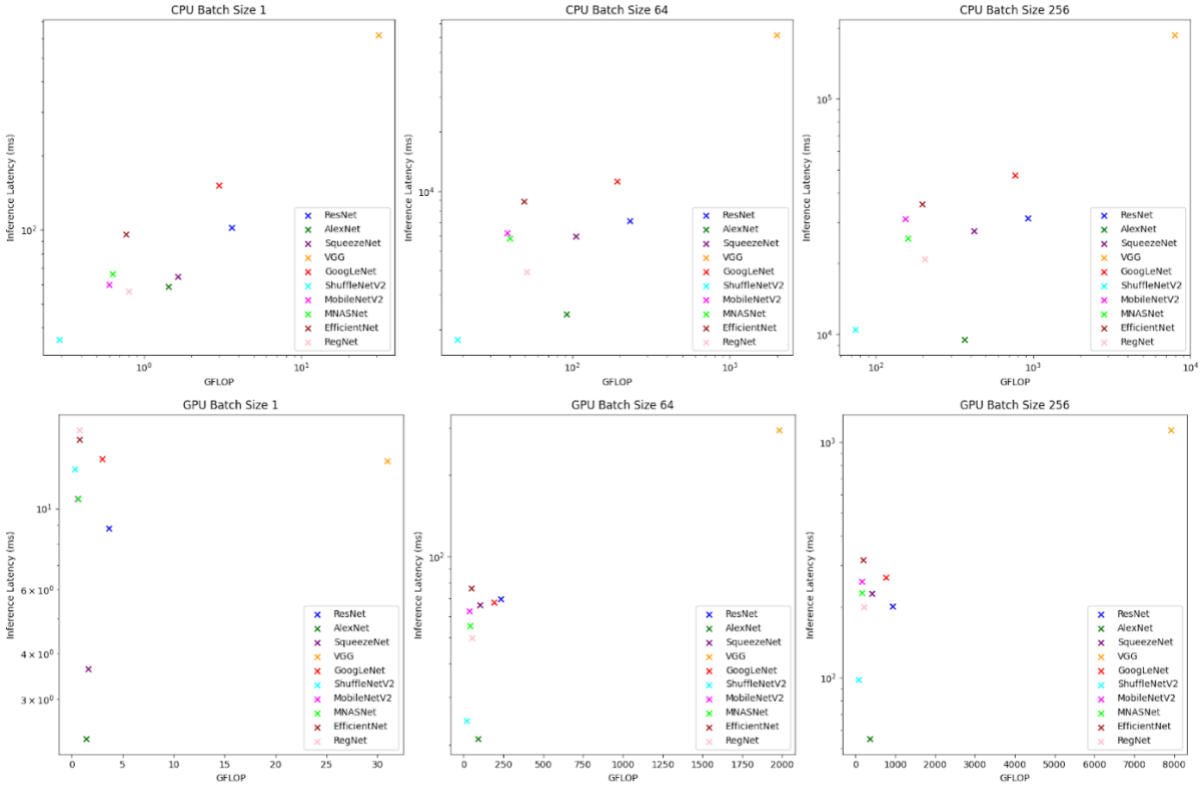
Peak Flops: 8.1 TFlops (FP32)
Memory Bandwidth: 300 GB/s

2.4:

We can see that almost all these models are compute bound on the CPU, which makes sense as these are image models which are primarily based on convolutions which have high operational intensity. However, it is important to note that the MobileNet, ShuffleNet, EfficientNet, RegNet, and MNASNet should all become memory bound on the GPU. AlexNet is interesting as the batch size drastically increases the arithmetic intensity due to a larger increase in FLOP count compared to memory (parameter memory doesn't change) which changes it from being memory bound to compute bound at higher batch sizes.

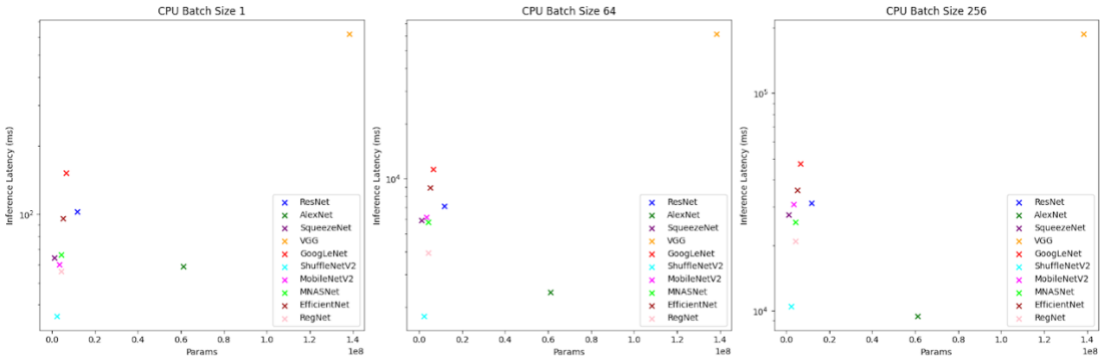
3.1

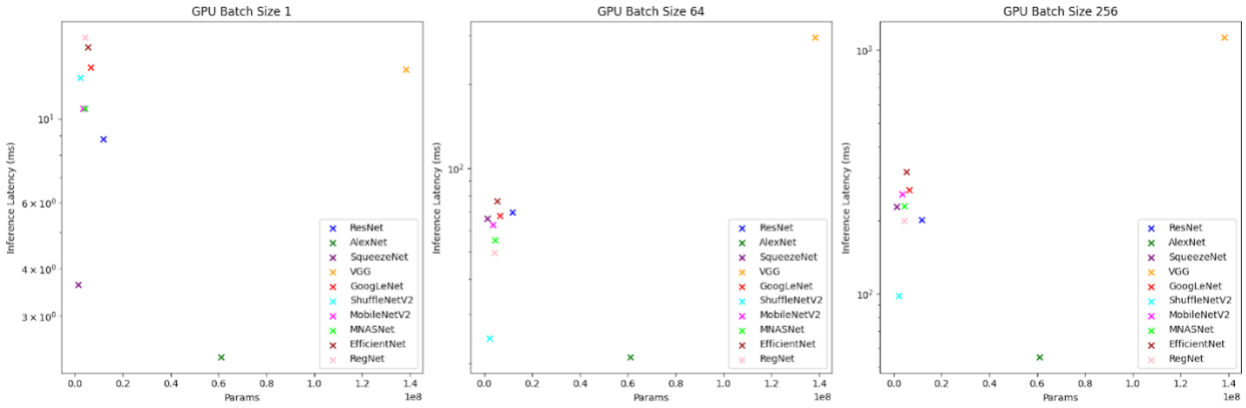
Used torch.cuda.event() for GPU time and time.time() for CPU timing



If we examine the y-axis, we can see the GPU takes less time by a couple orders of magnitude for inference. The VGG model consistently has a significant larger FLOP amount, which also explains why the larger batch sizes it still takes longer, due to it being severely compute bound.

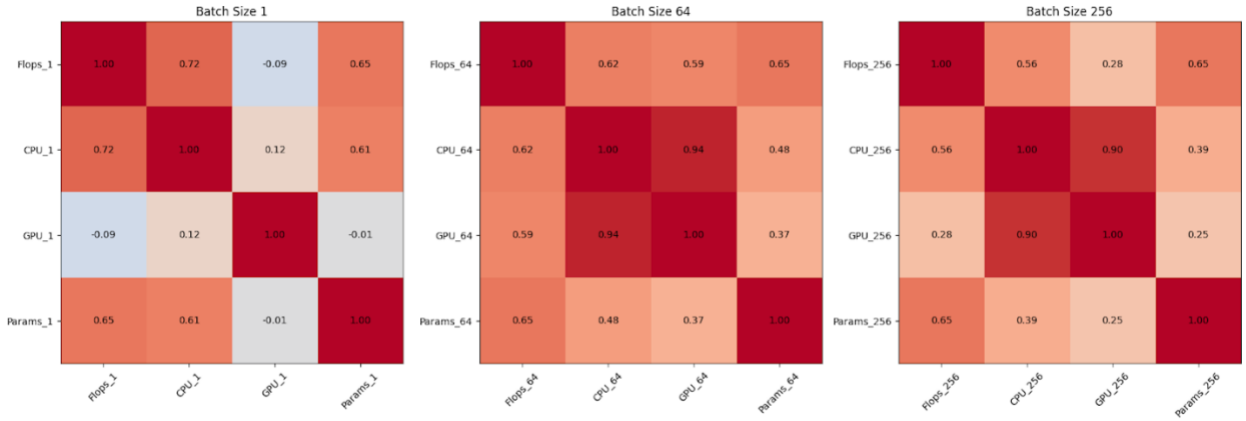
3.2





We can see here that for larger batch size, the inference latency significantly increases (pay attention to the order of magnitude) despite the parameter amount stays the same.

3.3

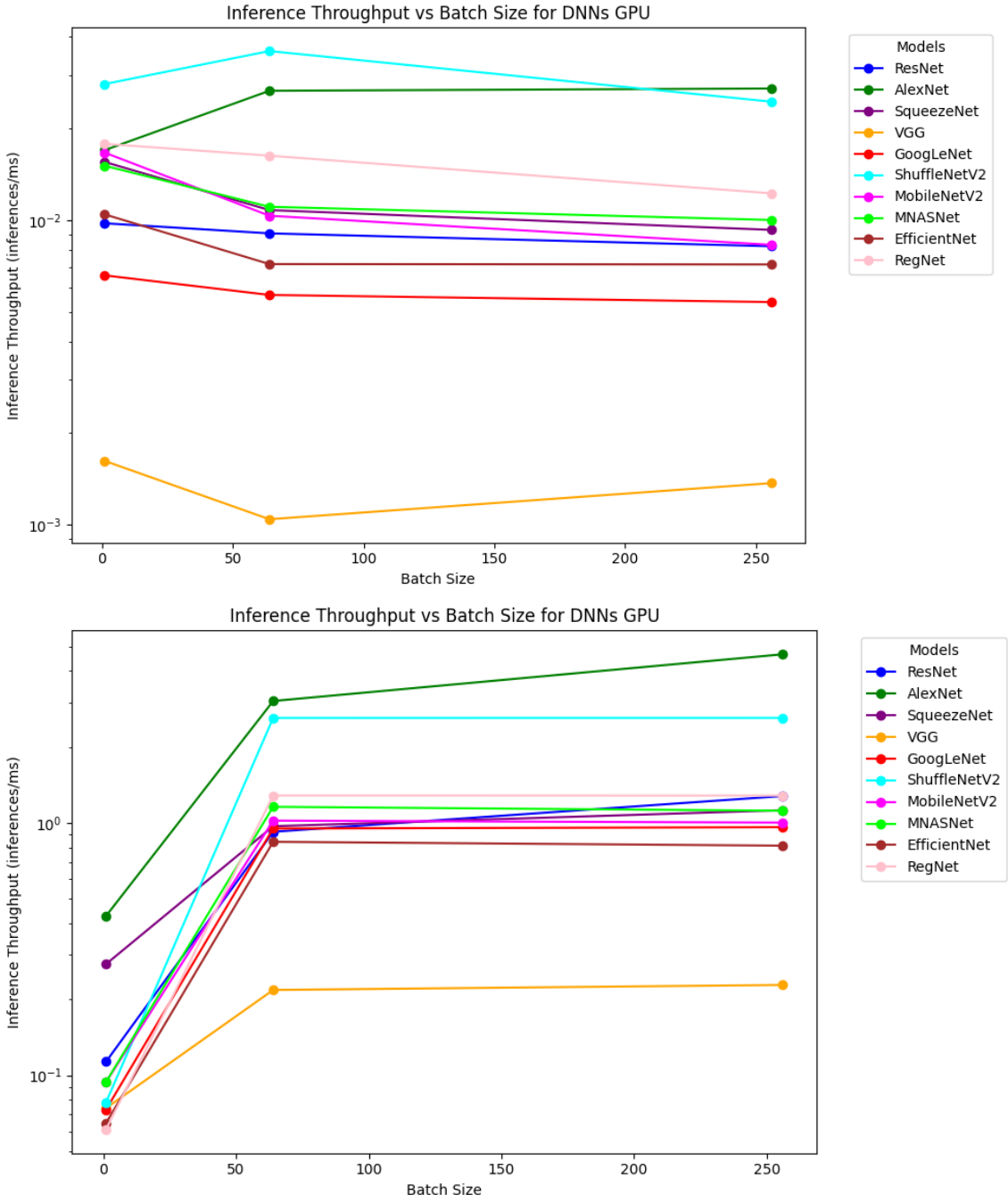


The correlation matrix depicts how a batch size of 64 and 256 are very similar for our models, however at a batch size of 1 we see a low monotonic relationship value for the GPU latency. Which could be due to the GPU not having a significant improvement at the smaller batch size.

3.4

Flops can generally be a good estimate of how fast a model runs, however it does not consider the memory movement requirements of a model, which can often be the limiting factor for inference speed. Parameters can be a good estimate when looking at the memory requirements to see if the model is bandwidth limited. But it is important to consider that the number of parameters does not change according to batch size which does not give the whole picture when it comes to memory requirements, especially if you look at model like AlexNet in which the parameter count is irrelevant compared to the total memory requirement at a higher batch size. VGG is a good example of knowing we are going to be hardware constraint by looking at the FLOP/Parameter requirements as it is significantly bigger than the other models.

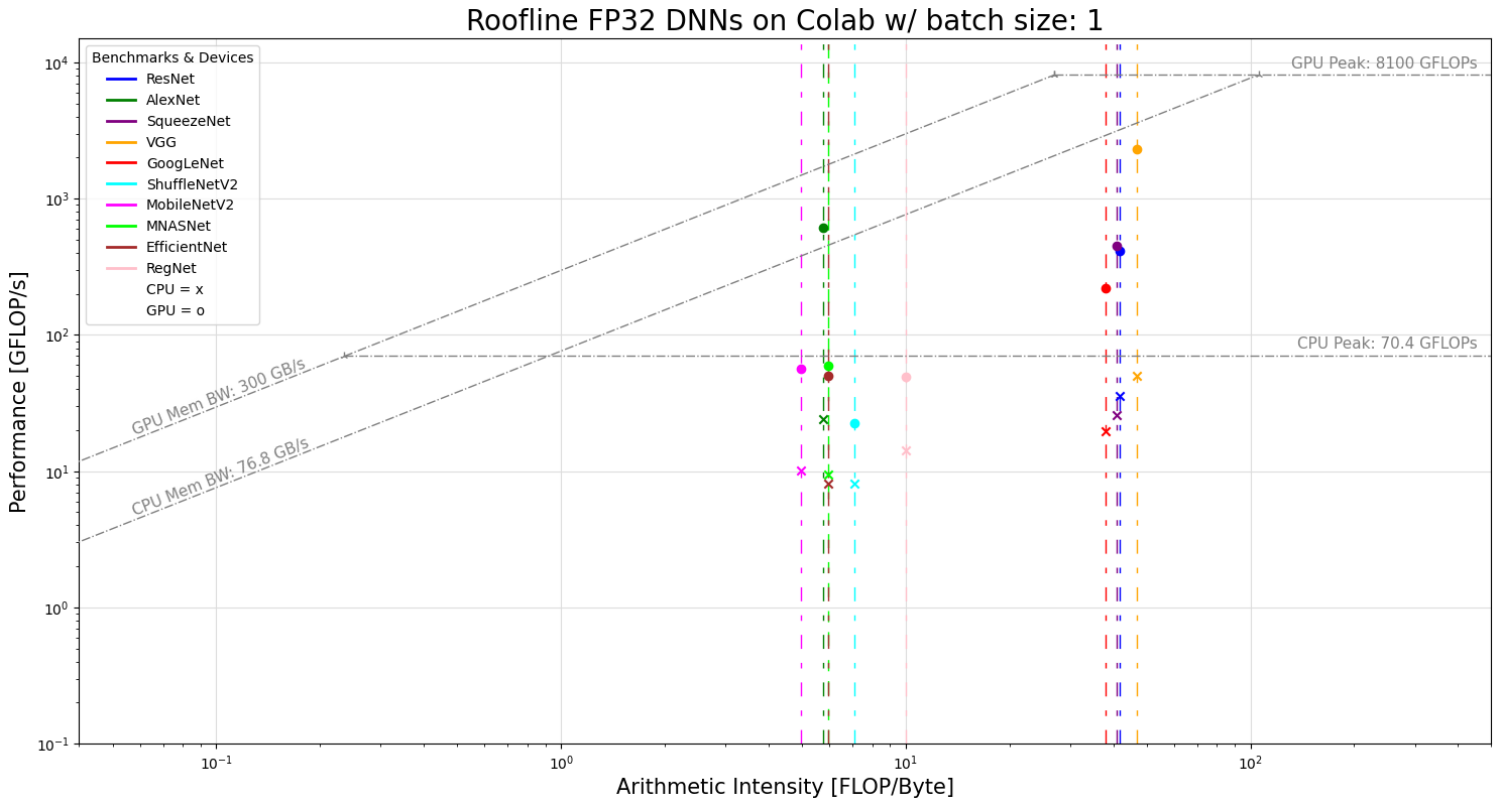
3.5

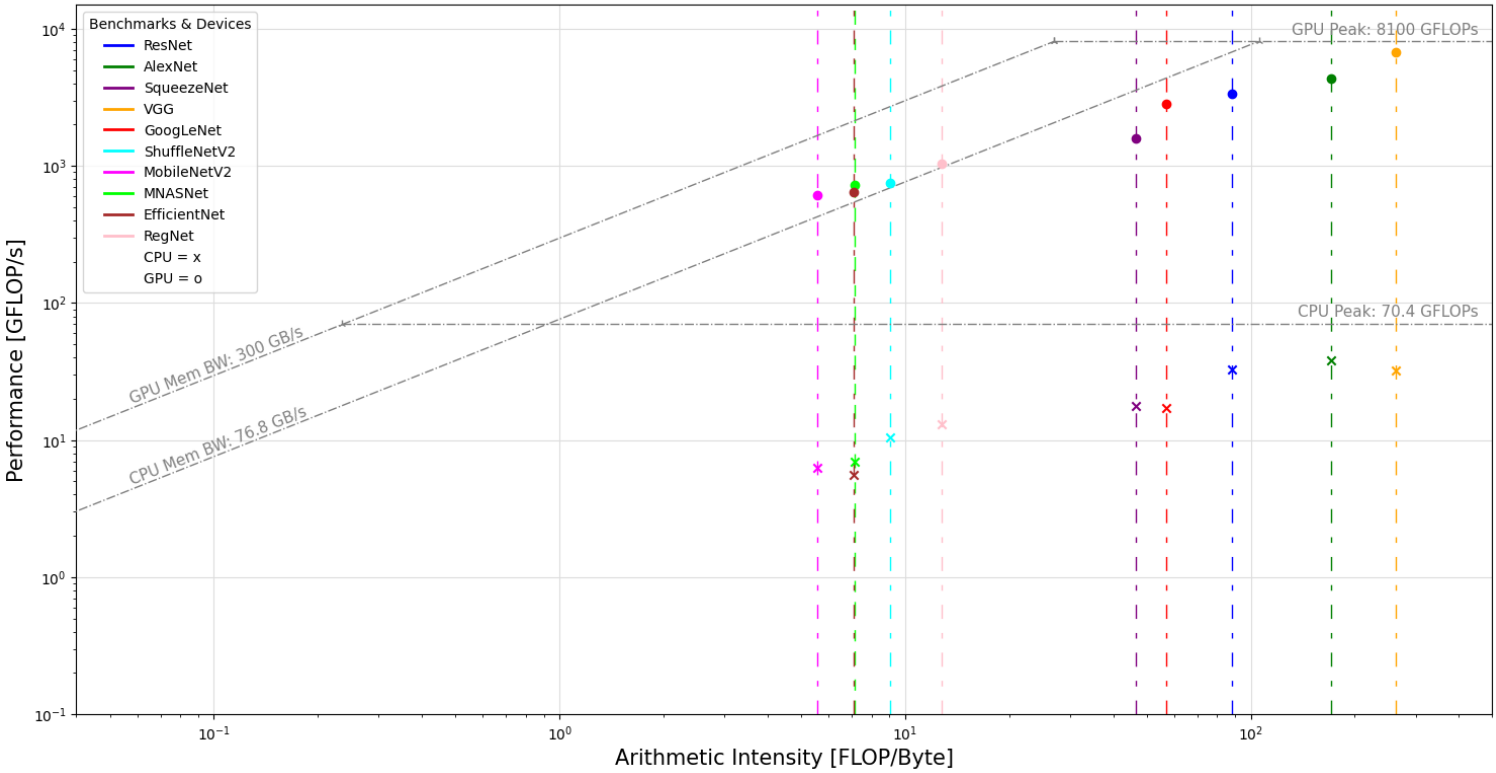


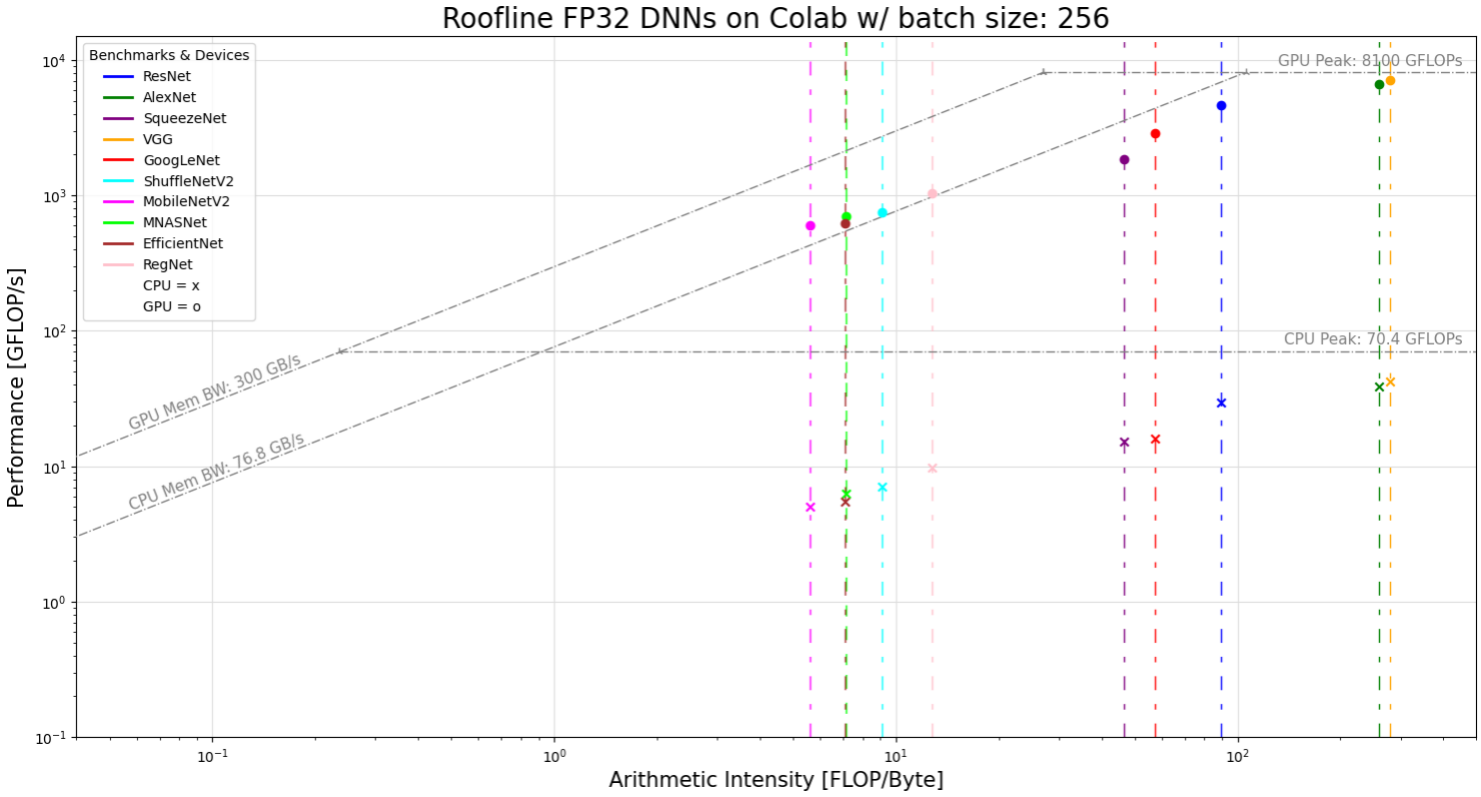
When we increase the batch size, the GPU can process more images simultaneously, improving throughput (the amount of data processed per second). This is because the GPU can handle multiple images/data at once by processing the batches in parallel, making better use of its resources.

For most of the models tested, we see that the ideal batch size is around 64, as going higher, especially to 256, doesn't result in much improvement. This could be since the GPU might not have enough cores or memory bandwidth (for extra activation calculations) to effectively distribute the work for such a large batch size.

4.1


Roofline FP32 DNNs on Colab w/ batch size: 1
Benchmarks & Devices
ResNet
AlexNet
SqueezeNet
VGG
GoogLeNet
ShuffleNetV2
MobileNetV2
MNASNet
EfficientNet
RegNet
CPU = x
GPU = o
GPU Peak: 8100 GFLOPs
CPU Peak: 70.4 GFLOPs
GPU Mem BW: 300 GB/s
CPU Mem BW: 76.8 GB/s
Performance [GFLOP/s]
Arithmetic Intensity [FLOP/Byte]


Roofline FP32 DNNs on Colab w/ batch size: 64
Benchmarks & Devices
ResNet
AlexNet
SqueezeNet
VGG
GoogLeNet
ShuffleNetV2
MobileNetV2
MNASNet
EfficientNet
RegNet
CPU = x
GPU = o
GPU Peak: 8100 GFLOPs
CPU Peak: 70.4 GFLOPs
GPU Mem BW: 300 GB/s
CPU Mem BW: 76.8 GB/s
Performance [GFLOP/s]
Arithmetic Intensity [FLOP/Byte]



4.2

I decided to overlay the Peak Flops and Memory Bandwidth over each other for the CPU and GPU to demonstrate situations where either the additional computational power of the GPU helped, the additional memory bandwidth helped, or both. We can see from these graphs that changing the batch size from 1 to 64 leads to a significant difference on how close the GPU inference performs to peak performance. In fact, the increased batch size allows for the less arithmetic intense models to be memory bound instead of compute bound. Despite them being memory bound on the GPU, they are less bottlenecked by the GPU which has a higher memory bandwidth and more computational power to pass the roofline of the CPU. We don't much of a difference between the 64 and 256 batch size rooflines, which is similar to the results at the throughput, suggesting that there is limited benefit to batching past 64 for any type of speedup. If we go back to looking at the batch size 1 roofline, we can see that there is a relatively small difference between the CPU and GPU again supporting the intuition that less parallelization is available. AlexNet as a model seems to standout at this batch size as it looks to be the only one that is memory bound here which is probably due to the simplicity of this model compared to the other as it just consistent of convolutions and feedforward layers which scale flops as significantly at higher batch sizes compared to smaller ones while disproportionally scaling memory requirements. We can also our suspicions about VGG16 confirmed from part 3 as it is consistently reach near peak flop count because with its large amount of convolutions.

5.1 [Used ChatGPT to help in writing some of these scripts for the layer by layer calculations]
AlexNet:
FLOPs Ratio (Forward/Backward): 1.03
Runtime Ratio (Forward/Backward): 0.38
Forward Pass Time: 0.0010 sec
Backward Pass Time: 0.0025 sec

VGG16
FLOPs Ratio (Forward/Backward): 1.01
Runtime Ratio (Forward/Backward): 0.49
Forward Pass Time: 0.0019 sec
Backward Pass Time: 0.0038 sec

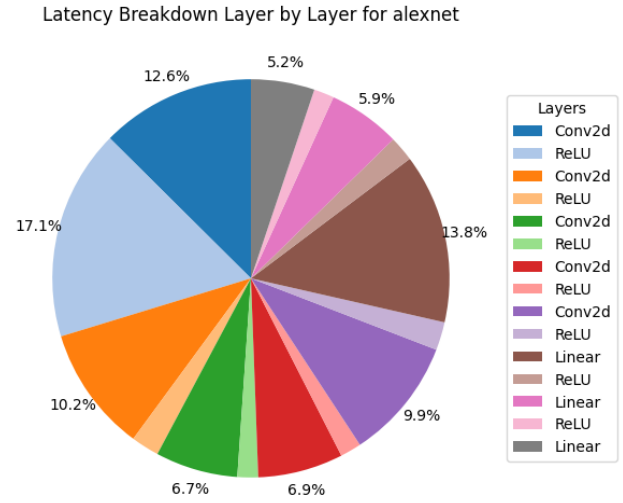
SqueezeNet
FLOPs Ratio (Forward/Backward): 1.00
Runtime Ratio (Forward/Backward): 0.60
Forward Pass Time: 0.0031 sec
Backward Pass Time: 0.0052 sec

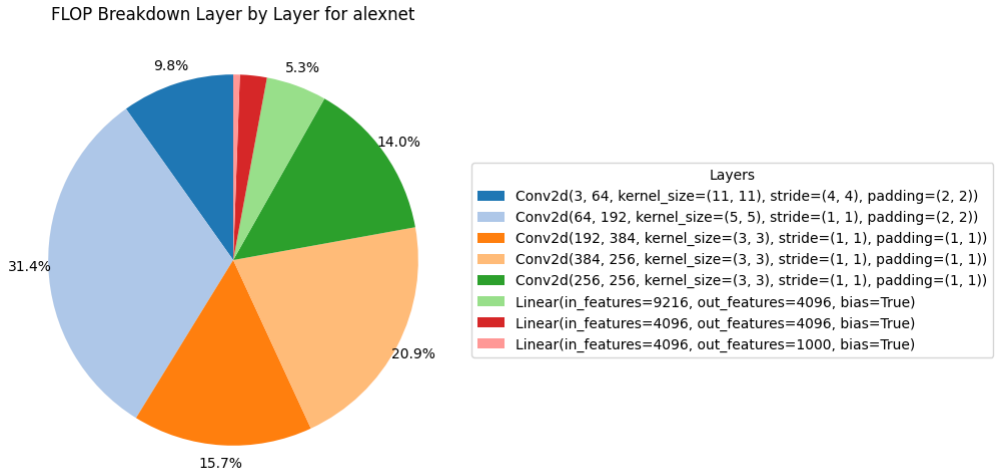
ResNet
FLOPs Ratio (Forward/Backward): 1.00
Runtime Ratio (Forward/Backward): 0.60
Forward Pass Time: 0.0034 sec
Backward Pass Time: 0.0056 sec

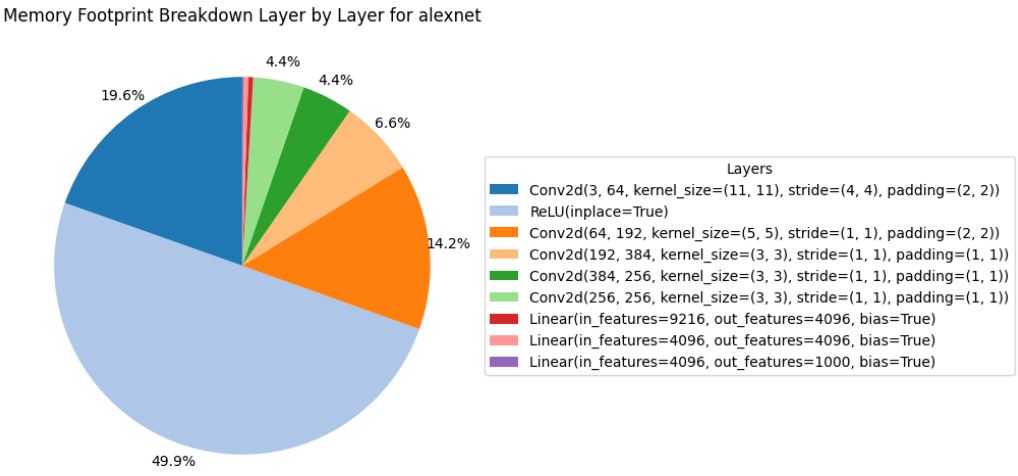
We can see from the 4 models tested that the runtime of the backwards pass takes approximately twice as long as the forward pass. With AlexNet taking longer while SqueezeBet and ResNet being less than double the time. I tried to measure the backwards pass computations by weight gradients, but did not account for activation gradients and accumulations, which is why the FLOP ratio is basically 1:1, take this number with a grain of salt.

5.2

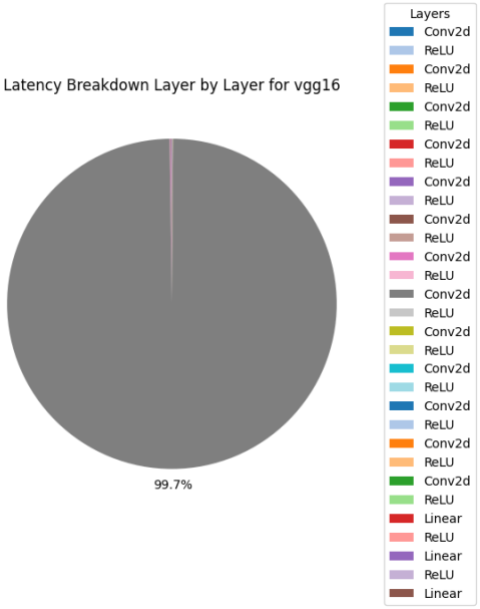
Alexnet:

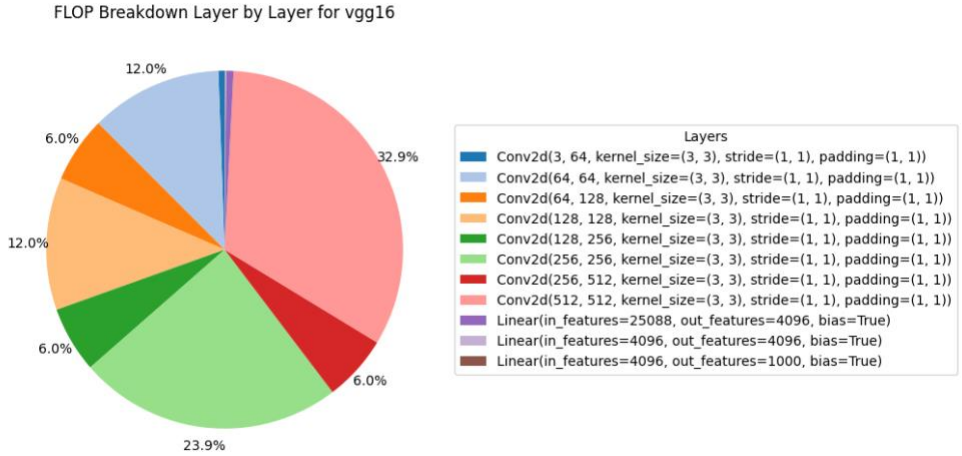

Latency Breakdown Layer by Layer for alexnet
12.6%
5.2%
5.9%
17.1%
13.8%
10.2%
9.9%
6.7%
6.9%
Layers
Conv2d
ReLU
Conv2d
ReLU
Conv2d
ReLU
Conv2d
ReLU
Conv2d
ReLU
Linear
ReLU
Linear
ReLU
Linear

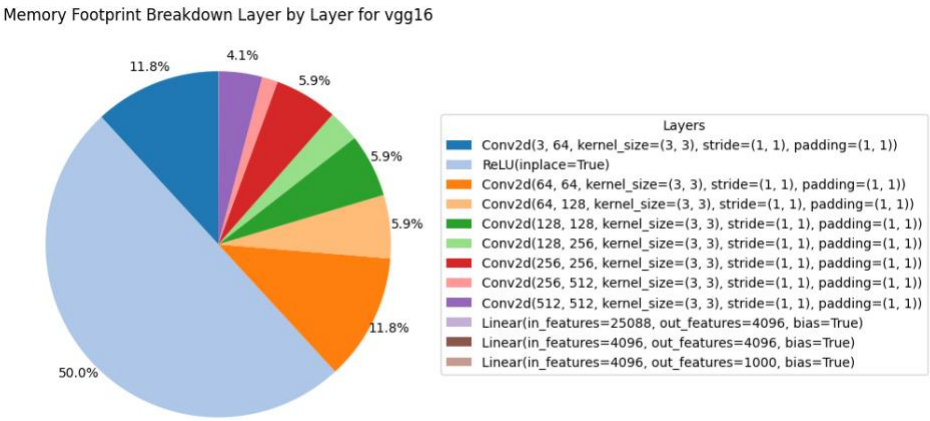

FLOP Breakdown Layer by Layer for alexnet
9.8%
5.3%
14.0%
31.4%
20.9%
15.7%
Layers
Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
Linear(in_features=9216, out_features=4096, bias=True)
Linear(in_features=4096, out_features=4096, bias=True)
Linear(in_features=4096, out_features=1000, bias=True)


Memory Footprint Breakdown Layer by Layer for alexnet
4.4%
4.4%
19.6%
6.6%
14.2%
49.9%
Layers
Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
ReLU(inplace=True)
Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
Linear(in_features=9216, out_features=4096, bias=True)
Linear(in_features=4096, out_features=4096, bias=True)
Linear(in_features=4096, out_features=1000, bias=True)

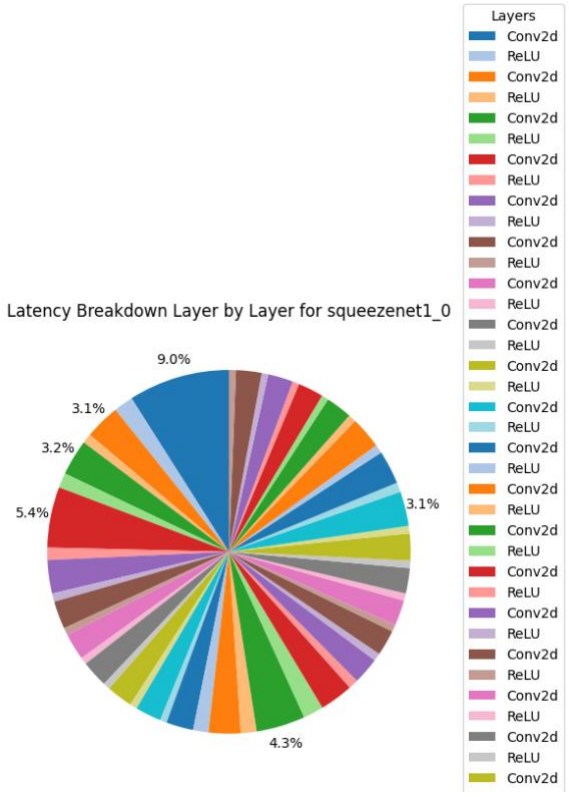
VGG16

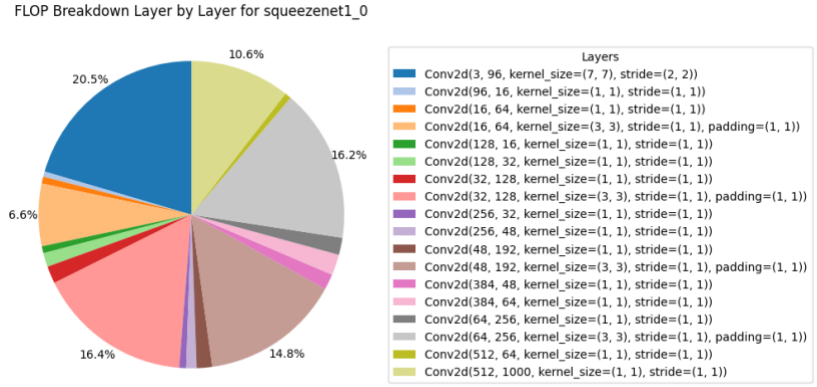

Latency Breakdown Layer by Layer for vgg16
99.7%
Layers
Conv2d
ReLU
Conv2d
ReLU
Conv2d
ReLU
Conv2d
ReLU
Conv2d
ReLU
Conv2d
ReLU
Conv2d
ReLU
Conv2d
ReLU
Conv2d
ReLU
Conv2d
ReLU
Conv2d
ReLU
Conv2d
ReLU
Linear
ReLU
Linear
ReLU
Linear

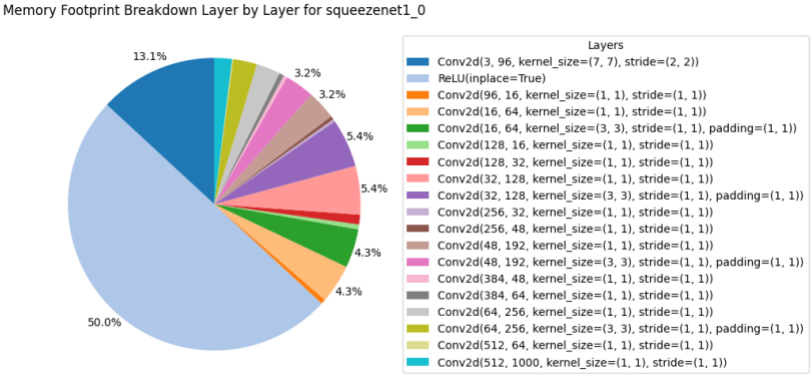

FLOP Breakdown Layer by Layer for vgg16
12.0%
6.0%
32.9%
12.0%
6.0%
6.0%
23.9%
Layers
Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
Linear(in_features=25088, out_features=4096, bias=True)
Linear(in_features=4096, out_features=4096, bias=True)
Linear(in_features=4096, out_features=1000, bias=True)


Memory Footprint Breakdown Layer by Layer for vgg16
11.8%
4.1%
5.9%
5.9%
5.9%
11.8%
50.0%
Layers
Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
ReLU(inplace=True)
Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
Linear(in_features=25088, out_features=4096, bias=True)
Linear(in_features=4096, out_features=4096, bias=True)
Linear(in_features=4096, out_features=1000, bias=True)

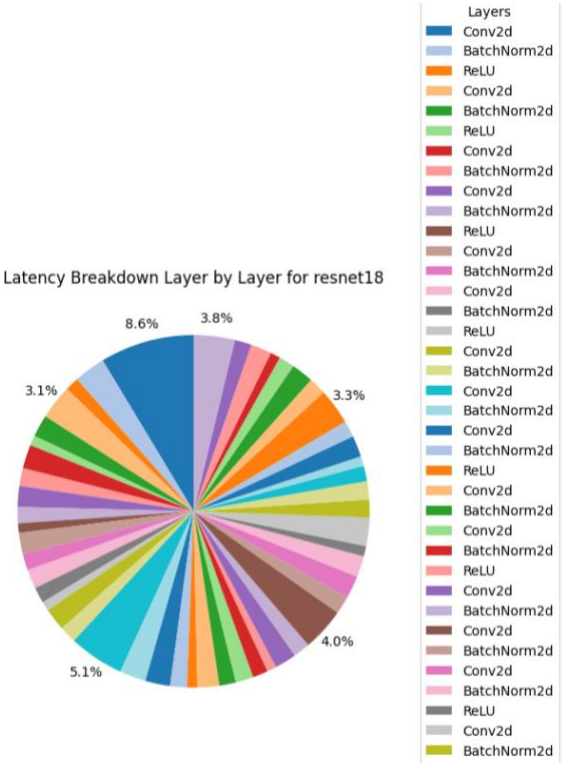
Squeezenet

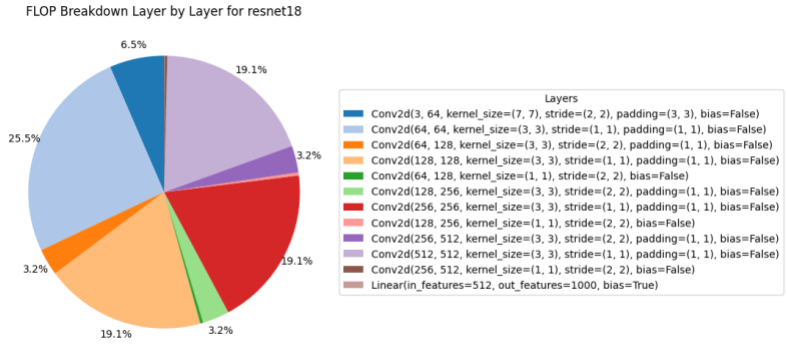

Latency Breakdown Layer by Layer for squeezenet1_0
9.0%
3.1%
3.2%
5.4%
3.1%
4.3%
Layers
Conv2d
ReLU
Conv2d
ReLU
Conv2d
ReLU
Conv2d
ReLU
Conv2d
ReLU
Conv2d
ReLU
Conv2d
ReLU
Conv2d
ReLU
Conv2d
ReLU
Conv2d
ReLU
Conv2d
ReLU
Conv2d
ReLU
Conv2d
ReLU
Conv2d
ReLU
Conv2d
ReLU
Conv2d
ReLU
Conv2d
ReLU
Conv2d

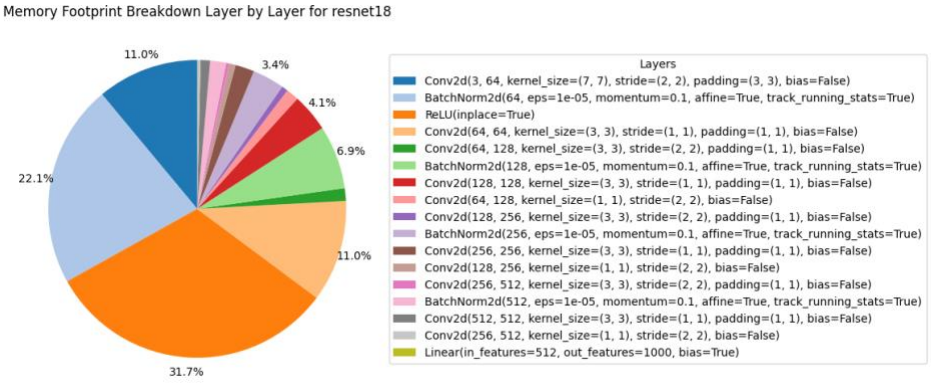

FLOP Breakdown Layer by Layer for squeezenet1_0
10.6%
20.5%
16.2%
6.6%
16.4%
14.8%
Layers
Conv2d(3, 96, kernel_size=(7, 7), stride=(2, 2))
Conv2d(96, 16, kernel_size=(1, 1), stride=(1, 1))
Conv2d(16, 64, kernel_size=(1, 1), stride=(1, 1))
Conv2d(16, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
Conv2d(128, 16, kernel_size=(1, 1), stride=(1, 1))
Conv2d(128, 32, kernel_size=(1, 1), stride=(1, 1))
Conv2d(32, 128, kernel_size=(1, 1), stride=(1, 1))
Conv2d(32, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
Conv2d(256, 32, kernel_size=(1, 1), stride=(1, 1))
Conv2d(256, 48, kernel_size=(1, 1), stride=(1, 1))
Conv2d(48, 192, kernel_size=(1, 1), stride=(1, 1))
Conv2d(48, 192, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
Conv2d(384, 48, kernel_size=(1, 1), stride=(1, 1))
Conv2d(384, 64, kernel_size=(1, 1), stride=(1, 1))
Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1))
Conv2d(64, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
Conv2d(512, 64, kernel_size=(1, 1), stride=(1, 1))
Conv2d(512, 1000, kernel_size=(1, 1), stride=(1, 1))


Memory Footprint Breakdown Layer by Layer for squeezenet1_0
13.1%
3.2%
3.2%
5.4%
5.4%
4.3%
4.3%
50.0%
Layers
Conv2d(3, 96, kernel_size=(7, 7), stride=(2, 2))
ReLU(inplace=True)
Conv2d(96, 16, kernel_size=(1, 1), stride=(1, 1))
Conv2d(16, 64, kernel_size=(1, 1), stride=(1, 1))
Conv2d(16, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
Conv2d(128, 16, kernel_size=(1, 1), stride=(1, 1))
Conv2d(128, 32, kernel_size=(1, 1), stride=(1, 1))
Conv2d(32, 128, kernel_size=(1, 1), stride=(1, 1))
Conv2d(32, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
Conv2d(256, 32, kernel_size=(1, 1), stride=(1, 1))
Conv2d(256, 48, kernel_size=(1, 1), stride=(1, 1))
Conv2d(48, 192, kernel_size=(1, 1), stride=(1, 1))
Conv2d(48, 192, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
Conv2d(384, 48, kernel_size=(1, 1), stride=(1, 1))
Conv2d(384, 64, kernel_size=(1, 1), stride=(1, 1))
Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1))
Conv2d(64, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
Conv2d(512, 64, kernel_size=(1, 1), stride=(1, 1))
Conv2d(512, 1000, kernel_size=(1, 1), stride=(1, 1))

Resnet







Takeaways from 5.2:
We can see just how complicated the SqueezeNet and resnet models are compared to AlexNet and VGG. These are all compute bound, but we can verify VGG16 being more compute intensive due to the domination in convolutions compared the other models