ROOFLINE MODELING AND COMPARISONS OF PERFORMANCE OF TWO NEURAL NETWORKS ON A100 AND V100 GPUS.

1. Introduction

Evaluating a neural network's performance through profiling metrics is crucial for large-scale deployments and releases. It's essential for a model to demonstrate robust training and testing outcomes to be suitable for extensive applications. NVIDIA provides a range of predefined metrics that can be utilized for this purpose. The Roofline model offers insights into the performance predictions of applications, models, or kernels, factoring in the peak capabilities and bandwidth limitations of the underlying hardware or system. In this experiment, we will perform roofline analysis on 2 neural networks namely Resnet and Alexnet on Imagenet dataset on two different GPU flavors - NVDIA A100 and NVDIA V100.

2. Experiment design

We will be performing roofline analysis on two neural networks models - Resnet18 and Alexnet on 2 different GPU flavors - namely A100 and V100. Hypothesis stated in our experiments would be that the A100, with its Ampere architecture, is expected to offer superior performance and efficiency metrics compared to the V100's Volta architecture for our models.

3. Experiment Overview

3.1 Roofline Modeling

Roofline modeling is a visualization technique that helps us analyze the performance of models/algorithms in the context of underlying hardware.[8] This analysis helps in giving insights on scope of improvement and areas of optimization. The model is represented as a two-dimensional graph with X-axis representing operational intensity (operations per byte) and Y-axis represents FLOP/sec. The roof represents the peak computational performance of the hardware and slope represents the memory bandwidth. The models are plotted as points on this graph, with their position indicating whether they would be considered in memory-bound or in compute bound. Roofline analysis could be done based on the plots where they lie on the graph. Roofline modeling offers a high-level overview of how well an application or computation can be expected to perform on a given hardware platform, and where the main bottlenecks might lie.

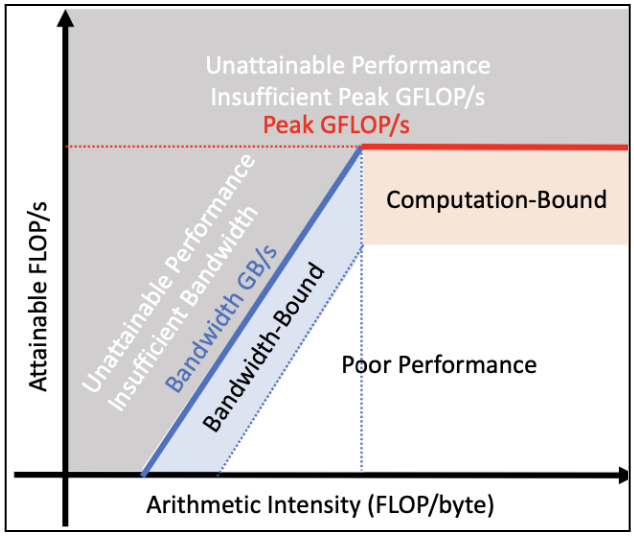


Fig.1 Roofline graph - five regions[2]

Computation-bound : the performance of the task is limited by the speed at which the processor can execute operations.
Bandwidth-bound : the performance of the task is limited by the rate at which data can be moved between processor-memory or memory hierarchies.
Arithmetic Intensity : Also known as operational intensity, is measured in (FLOP/byte) this is a measure of the number of operations performed per byte of data accessed. It's essentially the ratio of computation to the amount of data movement.High arithmetic intensity means more operations for each byte of data, suggesting the task might be compute-bound. Conversely, low arithmetic intensity implies the task could be more bandwidth-bound.

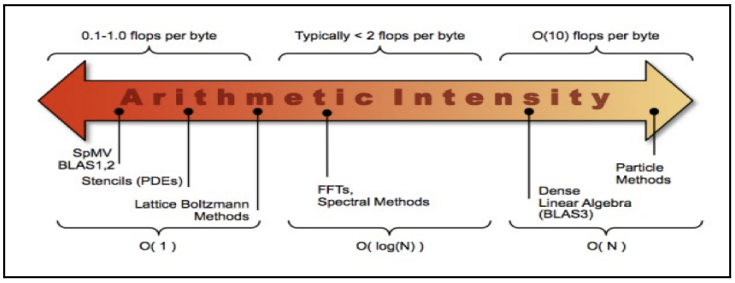


Fig.2 Arithmetic Intensity [2]

Attainable FLOP/s : Refers to the actual achievable FLOP/s on a given hardware. By comparing the attainable FLOP/s to the theoretical peak, one can gauge the efficiency of a computation on specific hardware.

3.2 Neural Network Models Analysis

Fig 3 : Resnet18 and AlexNet Summary

```
=> creating model 'resnet18'
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 64, 112, 112]           9,408
       BatchNorm2d-2         [-1, 64, 112, 112]             128
              ReLU-3         [-1, 64, 112, 112]               0
         MaxPool2d-4           [-1, 64, 56, 56]               0
            Conv2d-5           [-1, 64, 56, 56]          36,864
       BatchNorm2d-6           [-1, 64, 56, 56]             128
              ReLU-7           [-1, 64, 56, 56]               0
            Conv2d-8           [-1, 64, 56, 56]          36,864
       BatchNorm2d-9           [-1, 64, 56, 56]             128
             ReLU-10           [-1, 64, 56, 56]               0
       BasicBlock-11           [-1, 64, 56, 56]               0
           Conv2d-12           [-1, 64, 56, 56]          36,864
      BatchNorm2d-13           [-1, 64, 56, 56]             128
             ReLU-14           [-1, 64, 56, 56]               0
           Conv2d-15           [-1, 64, 56, 56]          36,864
      BatchNorm2d-16           [-1, 64, 56, 56]             128
             ReLU-17           [-1, 64, 56, 56]               0
       BasicBlock-18           [-1, 64, 56, 56]               0
           Conv2d-19          [-1, 128, 28, 28]          73,728
      BatchNorm2d-20          [-1, 128, 28, 28]             256
             ReLU-21          [-1, 128, 28, 28]               0
           Conv2d-22          [-1, 128, 28, 28]         147,456
      BatchNorm2d-23          [-1, 128, 28, 28]             256
           Conv2d-24          [-1, 128, 28, 28]           8,192
      BatchNorm2d-25          [-1, 128, 28, 28]             256
             ReLU-26          [-1, 128, 28, 28]               0
       BasicBlock-27          [-1, 128, 28, 28]               0
           Conv2d-28          [-1, 128, 28, 28]         147,456
      BatchNorm2d-29          [-1, 128, 28, 28]             256
             ReLU-30          [-1, 128, 28, 28]               0
           Conv2d-31          [-1, 128, 28, 28]         147,456
      BatchNorm2d-32          [-1, 128, 28, 28]             256
             ReLU-33          [-1, 128, 28, 28]               0
       BasicBlock-34          [-1, 128, 28, 28]               0
           Conv2d-35          [-1, 256, 14, 14]         294,912
      BatchNorm2d-36          [-1, 256, 14, 14]             512
             ReLU-37          [-1, 256, 14, 14]               0
           Conv2d-38          [-1, 256, 14, 14]         589,824
      BatchNorm2d-39          [-1, 256, 14, 14]             512
           Conv2d-40          [-1, 256, 14, 14]          32,768
      BatchNorm2d-41          [-1, 256, 14, 14]             512
             ReLU-42          [-1, 256, 14, 14]               0
       BasicBlock-43          [-1, 256, 14, 14]               0
           Conv2d-44          [-1, 256, 14, 14]         589,824
      BatchNorm2d-45          [-1, 256, 14, 14]             512
             ReLU-46          [-1, 256, 14, 14]               0
           Conv2d-47          [-1, 256, 14, 14]         589,824
      BatchNorm2d-48          [-1, 256, 14, 14]             512
             ReLU-49          [-1, 256, 14, 14]               0
       BasicBlock-50          [-1, 256, 14, 14]               0
           Conv2d-51            [-1, 512, 7, 7]       1,179,648
      BatchNorm2d-52            [-1, 512, 7, 7]           1,024
             ReLU-53            [-1, 512, 7, 7]               0
           Conv2d-54            [-1, 512, 7, 7]       2,359,296
      BatchNorm2d-55            [-1, 512, 7, 7]           1,024
           Conv2d-56            [-1, 512, 7, 7]         131,072
      BatchNorm2d-57            [-1, 512, 7, 7]           1,024
             ReLU-58            [-1, 512, 7, 7]               0
       BasicBlock-59            [-1, 512, 7, 7]               0
           Conv2d-60            [-1, 512, 7, 7]       2,359,296
      BatchNorm2d-61            [-1, 512, 7, 7]           1,024
             ReLU-62            [-1, 512, 7, 7]               0
           Conv2d-63            [-1, 512, 7, 7]       2,359,296
      BatchNorm2d-64            [-1, 512, 7, 7]           1,024
             ReLU-65            [-1, 512, 7, 7]               0
       BasicBlock-66            [-1, 512, 7, 7]               0
AdaptiveAvgPool2d-67            [-1, 512, 1, 1]               0
           Linear-68                 [-1, 1000]         513,000
================================================================
Total params: 11,689,512
Trainable params: 11,689,512
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.57
Forward/backward pass size (MB): 62.79
Params size (MB): 44.59
Estimated Total Size (MB): 107.96
----------------------------------------------------------------
```

```
=> creating model 'alexnet'
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 64, 55, 55]          23,296
              ReLU-2           [-1, 64, 55, 55]               0
         MaxPool2d-3           [-1, 64, 27, 27]               0
            Conv2d-4          [-1, 192, 27, 27]         307,392
              ReLU-5          [-1, 192, 27, 27]               0
         MaxPool2d-6          [-1, 192, 13, 13]               0
            Conv2d-7          [-1, 384, 13, 13]         663,936
              ReLU-8          [-1, 384, 13, 13]               0
            Conv2d-9          [-1, 256, 13, 13]         884,992
             ReLU-10          [-1, 256, 13, 13]               0
           Conv2d-11          [-1, 256, 13, 13]         590,080
             ReLU-12          [-1, 256, 13, 13]               0
        MaxPool2d-13            [-1, 256, 6, 6]               0
AdaptiveAvgPool2d-14            [-1, 256, 6, 6]               0
          Dropout-15                 [-1, 9216]               0
           Linear-16                 [-1, 4096]      37,752,832
             ReLU-17                 [-1, 4096]               0
          Dropout-18                 [-1, 4096]               0
           Linear-19                 [-1, 4096]      16,781,312
             ReLU-20                 [-1, 4096]               0
           Linear-21                 [-1, 1000]       4,097,000
================================================================
Total params: 61,100,840
Trainable params: 61,100,840
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.57
Forward/backward pass size (MB): 8.38
Params size (MB): 233.08
Estimated Total Size (MB): 242.03
----------------------------------------------------------------
```

3.2.1 Resnet18

ResNet18 is a deep neural network architecture with 18 layers, consisting of initial convolution and pooling layer followed by batch normalization and ReLU. It has residual blocks and skips connections. The pattern of Convolution blocks, batchnorm and relu is repeated over the layers. At the end, average pooling and linear layer with activation is applied.

3.2.2 Alexnet

It consists of convolution layers starting with 96 filters, each of size 11x11x3 with a stride of 4, followed by a ReLU activation function. The layer is then followed by a max-pooling operation and normalization. Next convolutional layer has 256 filters of size 5x5, followed by ReLU, max-pooling, and normalization. Similarly,384 filters of size 3x3, followed by a ReLU activation and 384 filters of size 3x3, followed by another ReLU. Last, 256 filters of size 3x3. This is followed by ReLU, and then max-pooling. Later there are 3 fully connected layers.

3.3 NVDIA GPU FLAVORS

3.3.1 NVDIA V100

NVIDIA® V100 is the most advanced data center GPU, based on the Volta architecture and features 5,120 CUDA cores and 640 Tensor cores. Has a memory bandwidth of up to 900 GB/s with its HBM2 memory. Delivers peak performance of 15.7 TFLOPS[4]

3.3.2 NVDIA A100

Uses the newer Ampere architecture, boasting 6,912 CUDA cores and with versatile 432 Tensor cores. Improved memory bandwidth, reaching up to 1.6 TB/s with HBM2e memory. Delivers peak performance of 19.5 TFLOPS..[3]

4. Pen-Paper Complexity Estimation

The input image size of Imagenet image is 3X224X224.

All the formulae, approximations for computing FLOPS and Bytes have been calculated referring to official document of NVIDIA[6]

FLOPs Estimation:

For Calculating Forward Pass:
For convolution layer it would be: 2 X Input channel X Output Height X Output Width X Output Channel X Kernel Shape [5]
For Linear Layer : 2 X Input X Output + Output

For Calculating Backward Pass:
To determine the number of floating-point operations (FLOPs) required for backward propagation, the same formula used for the convolution layer in the forward pass can be applied.[5] Empirical studies mentioned in the document suggest that in the initial layers, the ratio of backward to forward computations varies. However, as we progress to intermediate layers and as the batch size increases, this ratio tends to approach 2. Using this approximation, in combination with calculations from the convolution formula, it can be inferred that the FLOP count for backward propagation is roughly twice that of the forward pass, especially for deeper layers. Thus, a ratio of 2 is often used as an approximation for these higher layers.

So for the first layer we would be taking the same value, but for next layers we would be multiplying the values of forward pass with 2.

Bytes Estimation:

The bytes occupied by each element for Float32 could be 4 Bytes.

For forward pass:
For convolution layer = 4 X (Input channel X input Height X input width + Input Channel X Output Channel X Kernel Shape + Output Channel X Output Height X Output Width)
For Linear Layer = Input dimension X Output Dimension[5]

For Backward pass:
The parameters for forward and backward pass could be approximately the same. Similarly for linear layers as well.[5]

Following the formulae and approximation stated I have computed all the values for the excel sheet attached. Please refer to the excel sheet Project 1 - sxp8182 [7] for all pen-paper calculations. Also the other factors like Pooling layers, batch normalization layers and Relu layer have not been considered significant in the calculations since their values would be very less compared to convolution and linear layers.

The final values after computing everything in excel sheet are:

For pen paper estimation of Alexnet:

Total Forward FLOPS	**1428386152**
Total Backward FLOPS	**2268321104**

Total FLOPS	**4144604856**
Total Forward Bytes	**247689888**
Total Backward Bytes	**247689888**
Total Bytes	**495379776**

Based on the values computed, Arithmetic Intensity (AI) of Alexnet would be Totals FLOPs/Total Bytes = 3696707256 / 495379776 = **8.36 Flops/bytes**

For pen paper estimation of Resnet18:

Total Forward FLOPS	**3620020224**
Total Backward FLOPS	**4918296576**
Total FLOPS	**8538316800**
Total Forward Bytes	**123513504**
Total Backward Bytes	**123513504**
Total Bytes	**247027008**

Based on the values computed the Estimated Arithmetic Intensity(AI) of Resnet18 would be Total FLOPs/Total Bytes = 8538316800 / 247027008 = **34.56 FLOPs/Bytes**

5. Complexity measurement using NCU profiling tool

5.1 Steps performed for setup and execution:

1. Login to the CIMS server => `ssh sxp8182@access.cims.nyu.edu`
2. Login to the greene server => `ssh greene.hpc.nyu.edu`
3. To check the available modules in greene use => `module avail`
4. Loading the cuda and python [as mentioned in lecture 5][2] => `module load cuda/11.6.2 python/intel/3.8.6`
5. SSH into burst => `ssh burst`

   FOR V100

6. As mentioned in the lecture[2], using srun use the partition provided for this class => `srun --account=csci_3033_085-2023fa --partition=n1s8-v100-1 --time=2:00:00 --gres=gpu:1 --pty /bin/bash`
7. Inside the srun terminal, perform the following steps only once:
   7.1 Create the virtual environment => `python3 -m venv hw2`

7.2 Activate the created environment => `source hw2/bin/activate`

7.3 `pip install --upgrade pip`

7.4 Install the dependencies as mentioned in lecture 5 [2]using => `pip3 install torch torchvision torch summary`

7.5 Cloning the repository for word_language_model as mentioned in the homework description => `git clone` https://github.com/pytorch/examples.git

7.6 cd /examples/imagenet

7.6 Adding the following lines in main.py

Importing the profiler => `import torch.cuda.profiler as profiler`

Adding start and stop, please refer to the code attached, since we are profiling against the training loop for 1 iteration, as well as considering forward and backward pass -> we would put `profiler.start()` and `profiler.stop()` against the call to the model in forward pass and loss.backward().

```
def train(train_loader, model, criterion, optimizer, epoch, device, args):
    batch_time = AverageMeter('Time', ':6.3f')
    data_time = AverageMeter('Data', ':6.3f')
    losses = AverageMeter('Loss', ':.4e')
    top1 = AverageMeter('Acc@1', ':6.2f')
    top5 = AverageMeter('Acc@5', ':6.2f')
    progress = ProgressMeter(
        len(train_loader),
        [batch_time, data_time, losses, top1, top5],
        prefix="Epoch: [{}]".format(epoch))

    # switch to train mode
    model.train()

    end = time.time()
    for i, (images, target) in enumerate(train_loader):
        # measure data loading time
        data_time.update(time.time() - end)

        # move data to the same device as model
        images = images.to(device, non_blocking=True)
        target = target.to(device, non_blocking=True)

        profiler.start()
        # compute output
        output = model(images)
        loss = criterion(output, target)

        # measure accuracy and record loss
        #acc1, acc5 = accuracy(output, target, topk=(1, 5))
        losses.update(loss.item(), images.size(0))
        #top1.update(acc1[0], images.size(0))
        #top5.update(acc5[0], images.size(0))

        # compute gradient and do SGD step
        optimizer.zero_grad()
        loss.backward()

        profiler.stop()
        optimizer.step()

        # measure elapsed time
        batch_time.update(time.time() - end)
        end = time.time()

        if i % args.print_freq == 0:
            progress.display(i + 1)

        break
```

Fig. 4 - Profiler code added to train loop

8. To run the code without profiler, we would use the following command

```
python main.py /home/sxp8182/imagenet/ -a resnet18 --epochs 1
-b 1 --gpu 0
```

9. The imagenet folder was made by copying images from the imagenet folder available on HPC.
10. Commands to be used for profiling we would be profiling for batch size 1,

    Computing FLOPs:

    - FOR RESNET => `/share/apps/cuda/11.1.74/bin/ncu --profile-from-start off --metrics gpu__time_duration.sum,dram__bytes_read.sum,dram__bytes_write.sum,smsp__sass_thread_inst_executed_op_fadd_pred_on.sum,smsp__sass_thread_inst_executed_op_fmul_pred_on.sum,smsp__sass_thread_inst_executed_op_ffma_pred_on.sum --csv --page raw --log-file resnet18-v100.csv --target-processes all python main.py /home/sxp8182/imagenet/ -a resnet18 --epochs 1 -b 1 --gpu 0`

    - FOR ALEXNET => `/share/apps/cuda/11.1.74/bin/ncu --profile-from-start off --metrics gpu__time_duration.sum,dram__bytes_read.sum,dram__bytes_write.sum,smsp__sass_thread_inst_executed_op_fadd_pred_on.sum,smsp__sass_thread_inst_executed_op_fmul_pred_on.sum,smsp__sass_thread_inst_executed_op_ffma_pred_on.sum --csv --page raw --log-file alexnet-v100.csv --target-processes all python main.py /home/sxp8182/imagenet/ -a alexnet --epochs 1 -b 1 --gpu 0`

      Metric used =>
      gpu__time_duration.sum,dram__bytes_read.sum,dram__bytes_write.sum,smsp__sass_thread_inst_executed_op_fadd_pred_on.sum,smsp__sass_thread_inst_executed_op_fmul_pred_on.sum,smsp__sass_thread_inst_executed_op_ffma_pred_on.sum

      As mentioned in the lecture slides 5[2], **flop_count_sp:**
      smsp__sass_thread_inst_executed_op_fadd_pred_on.sum +
      smsp__sass_thread_inst_executed_op_fmul_pred_on.sum +
      smsp__sass_thread_inst_executed_op_ffma_pred_on.sum * 2

      Gpu_time is given by: **gpu__time_duration.sum**
      Memory utilization is given by:
      dram__bytes_read.sum+dram__bytes_write.sum

FOR A100

11. As mentioned in the lecture[2], using srun use the partition provided for this class => `srun --account=csci_3033_085-2023fa --partition=c12m85-a100-1 --gres=gpu:1 --pty /bin/bash`
12. Going inside the singularity container, a100 requires a specific version of torch and the normal environment was giving me error, thus going inside the singularity container worked.
    /share/apps/pytorch/1.13.0/run-pytorch-imagenet.bash
13. **Computing FLOPs**:

    - FOR RESNET => `ncu --profile-from-start off --metrics gpu__time_duration.sum,dram__bytes_read.sum,dram__bytes_write.sum,smsp__sass_thread_inst_executed_op_fadd_pred_on.sum,smsp__sass_thread_inst_executed_op_fmul_pred_on.sum,smsp__sass_thread_inst_executed_op_ffma_pred_on.sum --csv --page raw --log-file resnet18-a100.csv --target-processes all python main.py /home/sxp8182/imagenet/ -a resnet18 --epochs 1 -b 1 --gpu 0`

    - FOR ALEXNET => `ncu --profile-from-start off --metrics gpu__time_duration.sum,dram__bytes_read.sum,dram__bytes_write.sum,smsp__sass_thread_inst_executed_op_fadd_pred_on.sum,smsp__sass_thread_inst_executed_op_fmul_pred_on.sum,smsp__sass_thread_inst_executed_op_ffma_pred_on.sum --csv --page raw --log-file alexnet-a100.csv --target-processes all python main.py /home/sxp8182/imagenet/ -a alexnet --epochs 1 -b 1 --gpu 0`
14. Please note that while computing for both A100 and V100, we have passed –csv as argument which gives us all the logs computed in csv files.

5.2 Calculating values obtained from NCU.

The flops are computed using the csv file values, the csv file is imported as dataframe for model and then values are computed using the following formula.
All the calculations have been done in colab notebook attached => FLOP_SUM.ipynb [8]

```
Flops
=(2*subsetted_dataframe['smsp__sass_thread_inst_executed_op_ffma_pred_on.sum'])+subsetted_dataframe['smsp__sass_thread_inst_executed_op_fmul_pred_on.sum']+subsetted_dataframe['smsp__sass_thread_inst_executed_op_fadd_pred_on.sum']
Time = (subsetted_dataframe['gpu__time_duration.sum'].sum())/1000000000
Total Bytes = (subsetted_dataframe['dram__bytes_read.sum'] + subsetted_dataframe['dram__bytes_write.sum']).sum()
```

Operational Intensity(OI)= flops/total_bytes

Flop per sec = flops/time

Code snippet for the same:

```
[ ] def print_flops_for_dataframe(columns):
    subsetted_dataframe=columns[["dram__bytes_read.sum", "dram__bytes_write.sum", "gpu__time_duration.sum", "smsp__sass_thread_inst_executed_op_ffma_pred_on.sum", "smsp__sass_thread_inst_exe
    subsetted_dataframe=subsetted_dataframe.iloc[1:,:]
    for col in subsetted_dataframe.columns:
        subsetted_dataframe[col] = subsetted_dataframe[col].str.strip().str.replace(',', '').astype(float)
    flops=(2*subsetted_dataframe['smsp__sass_thread_inst_executed_op_ffma_pred_on.sum']) + subsetted_dataframe['smsp__sass_thread_inst_executed_op_fmul_pred_on.sum'] + subsetted_dataframe['s
    flops=flops.sum()
    print("flops.sum:",flops)
    time = (subsetted_dataframe['gpu__time_duration.sum'].sum())/1000000000
    print("time in secs:",time)
    total_bytes = (subsetted_dataframe['dram__bytes_read.sum'] + subsetted_dataframe['dram__bytes_write.sum']).sum()
    print("total_bytes:",total_bytes)
    OI=flops/total_bytes
    print("OI:",OI)
    Flop_sec = flops/time
    print("Flops/sec:",Flop_sec)
```

Fig.5 - Code snippet of the function calculating values

5.3 Calculating values for V100

RESNET18

CSV file generated:

resnet18-v100.csv

1 to 10 of 251 entries Filter

num_warps_avg_per_active_cycle	smsp__sass_thread_inst_executed_op_fadd_pred_on.sum	smsp__sass_thread_inst_executed_op_ffma_pred_on.sum	smsp__sass_thread_inst_executed_op_fmul_pred_on.sum
	inst	inst	inst
	0	0	0
	802,816	122,028,032	802,816
	0	0	0
	3,528,896	2,441,536	3,076,416
	0	0	0
	0	0	0
	16,384	98,304	24,576
	4,472,832	59,981,824	0
	0	0	0

Fig.6 - Resnet CSV file generated

By calling the function defined in the code snippet => the output values are:

```
dataframe=pd.read_csv("resnet18-v100.csv")
print_flops_for_dataframe(dataframe)
```

```
flops.sum: 8297434213.0
time in secs: 0.005030304
total_bytes: 484320736.0
OI: 17.13210605337369
Flops/sec: 1649489615935.7366
```

Fig.7 Resnet output values

ALEXNET

CSV file generated:

alexnet-v100.csv

1 to 10 of 90 entries Filter

n_warps_avg_per_active_cycle	smsp__sass_thread_inst_executed_op_fadd_pred_on.sum	smsp__sass_thread_inst_executed_op_ffma_pred_on.sum	smsp__sass_thread_inst_executed_op_fmul_pred_on.sum
	inst	inst	inst
	0	71,598,080	193,600
	0	193,792	0
	0	0	0
	0	0	0
	0	226,099,200	139,968
	0	140,032	0
	0	0	0

Fig.8 ALEXNET CSV file

By calling the function defined in the code snippet => the output values are:

```
dataframe=pd.read_csv("alexnet-v100.csv")
print_flops_for_dataframe(dataframe)

flops.sum: 4117546463.0
time in secs: 0.002397472
total_bytes: 839343936.0
OI: 4.905672497763777
Flops/sec: 1717453410509.0696
```

Fig. 9 - ALEXNET output values

5.4 Calculating values for A100

RESNET18

CSV file generated:

resnet18-a100.csv

1 to 10 of 329 entries Filter

active_cycle	smsp__sass_thread_inst_executed_op_fadd_pred_on.sum	smsp__sass_thread_inst_executed_op_ffma_pred_on.sum	smsp__sass_thread_inst_executed_op_fmul_pred_on.sum
	inst	inst	inst
	0	0	150528
	0	0	9408
	0	0	802816
	0	0	802816
	0	0	0
	3592384	2441536	3064128
	0	0	0
	0	0	0
	0	0	200704

Show 10 per page

1 2 10 30 33

Fig.10 - Resnet CSV file

By calling the function defined in the code snippet => the output values are:

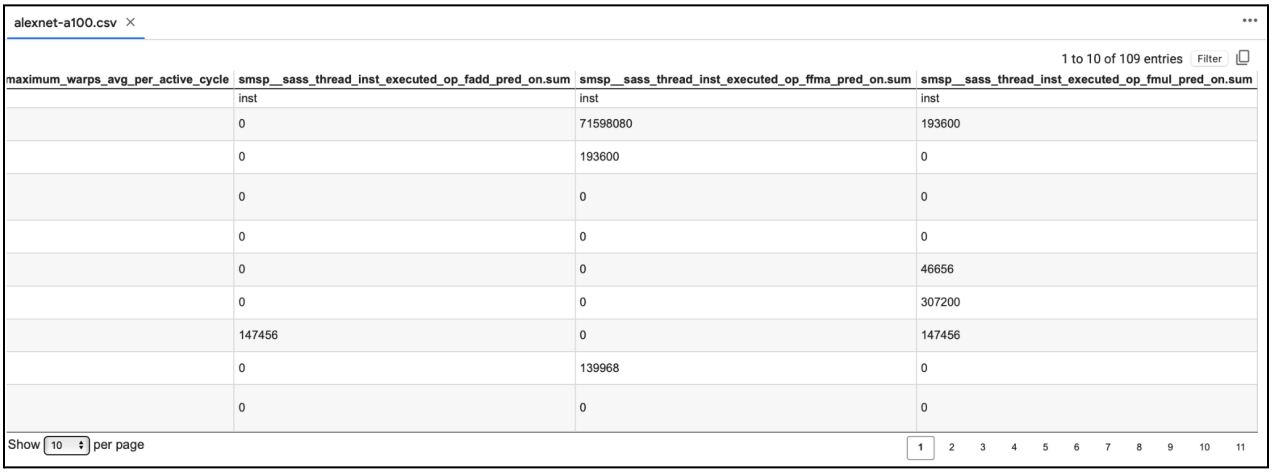
```
dataframe=pd.read_csv("resnet18-a100.csv")
print_flops_for_dataframe(dataframe)

flops.sum: 1214372889.0
time in secs: 0.003620512
total_bytes: 383364864.0
OI: 3.1676687225045224
Flops/sec: 335414684166.21735
```

Fig.11 - Resnet outputs generated

ALEXNET

CSV file generated:


alexnet-a100.csv

1 to 10 of 109 entries Filter

maximum_warps_avg_per_active_cycle	smsp__sass_thread_inst_executed_op_fadd_pred_on.sum	smsp__sass_thread_inst_executed_op_ffma_pred_on.sum	smsp__sass_thread_inst_executed_op_fmul_pred_on.sum
	inst	inst	inst
	0	71598080	193600
	0	193600	0
	0	0	0
	0	0	0
	0	0	46656
	0	0	307200
	147456	0	147456
	0	139968	0
	0	0	0

Show 10 per page

1 2 3 4 5 6 7 8 9 10 11

Fig. 12 - Alexnet CSV file generated

By calling the function defined in the code snippet => the output values are:

```
dataframe=pd.read_csv("alexnet-a100.csv")
print_flops_for_dataframe(dataframe)
```

```
flops.sum: 1392806057.0
time in secs: 0.001568768
total_bytes: 735346944.0
OI: 1.8940801595280716
Flops/sec: 887834311383.2001
```

Fig. 13 - ALEXNET output generated.

6. Roofline modeling

Based on the values computed in the last section, the comparative analysis table would look like the following:

		V100	A100
RESNET18	FLOPS	8297434213.0	1214372889.0
	BYTES	484320736.0	383364864.0
	TIME in secs	0.005030304	0.003620512
	OI(AI)	17.13210605337369	3.1676687225045224

	FLOPs/byte		
	FLOP/Sec	1649489615935.7366	335414684166.21735
ALEXNET	FLOPS	4117546463.0	1392806057.0
	BYTES	839343936.0	735346944.0
	TIME	0.002397472	0.001568768
	OI(AI) FLOPs/byte	4.905672497763777	1.8940801595280716
	FLOP/Sec	1717453410509.0696	887834311383.2001

NVDIA V100 and A100 specs

	V100	A100
Single Precision FP32 - Peak Computation Performance	15.7 TFLOPs/Sec	19.5 TFLOPs/sec
Memory Bandwidth	900 GB/s	1555 GB/s
Flops/byte	17.4	12.540

6.1. Roofline Analysis for V100

Attainable Performance = min {PeakPerformance, AI*PeakBandwidth}

Based on the values obtained in the table:
Attainable Performance for Resnet18 = 1.649 TFLOPs/sec

AI (FLOPs/Bytes) = 17.13

The point to plot for Resnet18 {17.13, 1.6 }

Attainable Performance for Alexnet = min { 1717453410509,(839343936.0/0.002397472)*4.905 }
= 1.717 TFLOPs/sec

AI (FLOPs/Bytes) = 4.9

The point to plot for Alexnet {4.9, 1.717 }

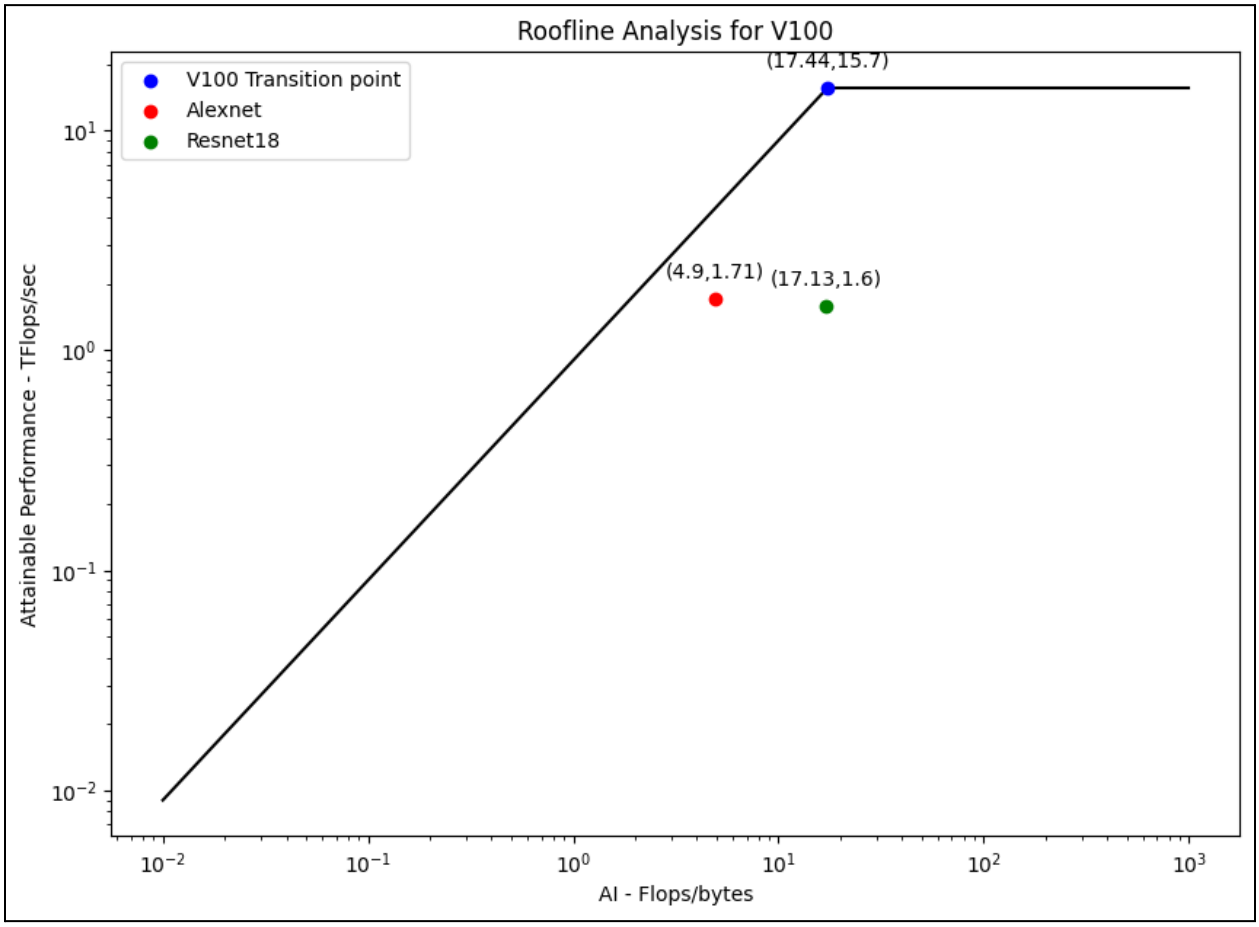


Fig 14 - Roofline analysis of Alexnet and Resnet18 on V100

6.2. Roofline Analysis for A100

Attainable Performance = min {PeakPerformance, AI*PeakBandwidth}

Based on the values obtained in the table:
Attainable Performance for Resnet18 = `0.3353` TFLOPs/sec

AI (FLOPs/Bytes) = 3.167

The point to plot for Resnet18 {3.167, 0.3353}

Attainable Performance for Alexnet =`0.887` TFLOPs/sec

AI (FLOPs/Bytes) = `1.89`

The point to plot for Alexnet {1.89, 0.887 }

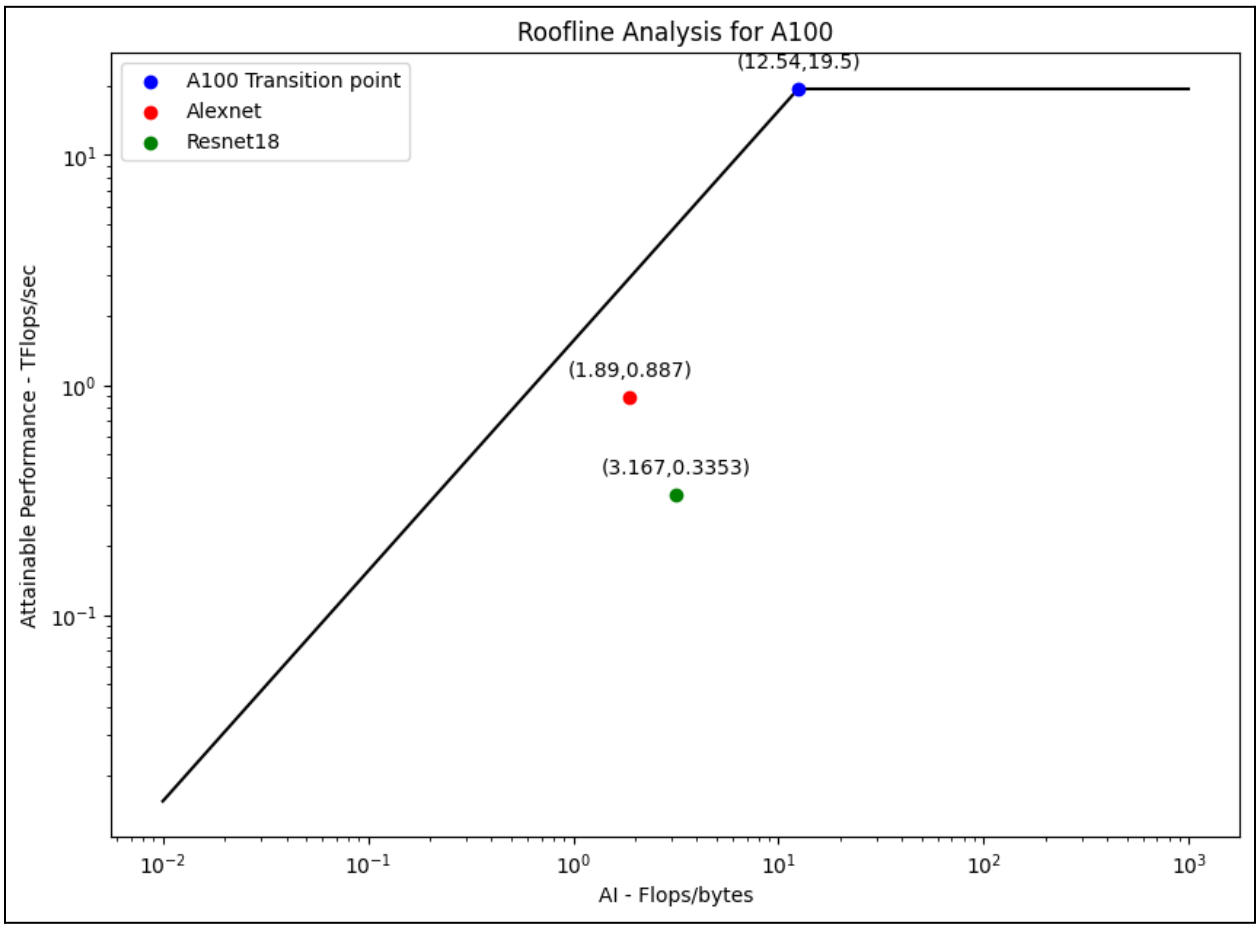


Fig 15 - Roofline analysis of Alexnet and Resnet18 on A100

7. Discussions

Roofline analysis for V100:

Alexnet is plotted at (4.9, 1.71), it lies considerably below the roofline but is much farther from the memory bandwidth line compared to the A100's plot. This suggests that while Alexnet's performance on the V100 is still somewhat memory-bound, it's closer to being compute-efficient than on the A100.

Resnet18 is positioned at (17.13, 1.6), it's closer to the transition point but still below the roofline. While its performance is also memory-bound, its operational intensity is considerably higher than Alexnet, making it more compute-efficient on the V100.

Both models are operating at a performance level far below the peak performance of the V100 hardware (15.7 TFLOPs/sec). This suggests significant room for optimization.

Alexnet and Resnet18 are both operating in the memory-bound region, but they are notably farther from the memory-bound line than in the A100's plot. This suggests that the V100 offers better memory performance for these models compared to the A100.
For Alexnet exploring methods to optimize memory access patterns and data movement to approach the transition point could be done. For Resnet18, given its proximity to the transition point, optimizing both compute and memory aspects would be beneficial.

Roofline analysis for A100:
Alexnet is Plotted at (1.89, 0.887), it lies below the roofline and closer to the memory bandwidth line. This suggests that Alexnet's performance on the A100 is more memory-bound than compute-bound. To optimize Alexnet, focusing on improving memory access patterns might yield better performance gains. This could also be happening dude to the batch size given. Utilization of the memory bandwidth of the underlying hardware is not happening.
Resnet18 is Located at (3.167, 0.3353), Resnet18 is also positioned below the roofline and near the memory bandwidth line, indicating that it too is memory-bound on the A100. Like Alexnet, Resnet18 could benefit from memory access optimization. Resnet needs more memory optimization than Alexnet to fully utilize the hardware capabilities. Both Alexnet and Resnet18 are operating at a performance level considerably below the peak performance of the A100 hardware (19.5 TFLOPs/sec).
It seems the main bottleneck for both models on the A100 is memory bandwidth rather than compute capacity. Methods that reduce data movement, optimize memory access patterns, leverage effective use of cache, and maybe even utilize techniques like data quantization or pruning could be used.

However, the models seem to perform more compute-efficiently on the V100 compared to the A100, especially Resnet18, which is close to the transition point, and thus roofline plot helps to actually reject the hypothesis claimed initially that performance of models are expected to be better on A100 than V100.

8. Conclusion

The most important takeaway from this project is understanding the significance of roofline modeling, being able to compute roofline models for different GPU flavors and then plotting the points for different models, helped me analyze the performance of models on different GPUs. The pen-paper analysis matches reasonably well with the NCU values; the differences experienced could be because of the kernel's internal computations or our approximations. This also helped me understand the scope of improvement in the models while running on different GPUs as well as making models compatible to utilize the underlying GPUs fully.

9. References

[1] Imagenet:
https://github.com/pytorch/examples/blob/main/imagenet/main.py

[2] Lecture 6 Slides :
https://brightspace.nyu.edu/content/enforced/300311-FA23_CSCI-GA_3033_1_085/lecture6_fall23.pdf

[3] Nvidia A100:
https://www.nvidia.com/en-us/data-center/a100/

[4]NVIDIA V100:
https://www.nvidia.com/en-us/data-center/v100/

[5] FLOPs computation blog:
https://medium.com/@pashashaik/a-guide-to-hand-calculating-flops-and-macs-fa5221ce5ccc

[6] Official NVDIA doc for calculating AI:
https://docs.nvidia.com/deeplearning/performance/dl-performance-convolutional/index.html

[7] Excel sheet of pen-paper calculations:
https://docs.google.com/spreadsheets/d/18cJxXc8pjZ5ohkV3utSvuNVhS0VrfvoHvK8dAGBApm4/edit?usp=sharing

[8] Google colab for calculating values of CSVs generated:
https://colab.research.google.com/drive/1Q7HH0_fh_mPPFvG5-LILhHT7JQkPZCOZ?usp=sharing

[9] Roofline modeling Wikipedia:
https://en.wikipedia.org/wiki/Roofline_model

***** Please Note:***
The Imagenet Code, ncu csv files, excelsheet and colab notebook all are attached along with the report in the HW submission.