Homework 2 – Performance Study of Transformers

1. INTRODUCTION

As demand and innovation of deep learning models grows, we can see a growing trend of training these models on bigger datasets. Training of such deep models on such big datasets often takes days and is also very resource heavy. However, inefficient training and model implementations also adds to this.
Profiling is an invaluable tool which helps engineers determine performance bottlenecks and analyzing and improving these can help increase the performance of the entire software or model. Profiling is thus a very important part of a machine learning development pipeline and must be done. This report,created for *Homework-2 of Cloud & Machine Learning* course, aims to profile one particular block of a transformer and compare the results with results obtained by hand generation. I aim to analyze trends and variations in time and memory use as we vary batch size to help get a deeper understanding of GPUs and how it is being used by the deep learning model.

2. BACKGROUND

Transformers were first introduced by Vaswani et al.[1] in 2017 and revolutionized not only natural processing language tasks but has also led to innovations in neighboring fields, like computer vision like Dosovitskiy et al.[2] in 2020.
Transformers introduced the mechanism of self attention and multi-headed attention to measure the importance of various words in a text input. Self attention helps the model consider contextual information while multi–head attention captures attention from various parts of the text input which allows the model to understand the relationship and connection between various tokens or words of the input.
Transformers also contain an encoder-decoder unit. The encoder maps an input text sequence to a sequence of continuous representations. This representation is then put into the decoder. The decoder then uses the output of the encoder and the output of the decoder at the previous step to generate an output sequence of tokens. The encoder layers have multi-head attention, layers with ReLU activation and it also makes use of Layer Norm and residual connections. Positional encoding is added to the tokens to ensure that the transformer model understands the information about the tokens in the input sequence or text.

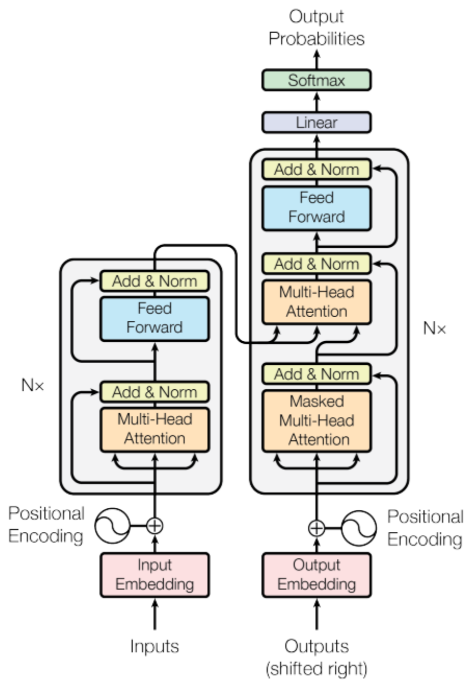


Figure 1: The Transformer Architecture

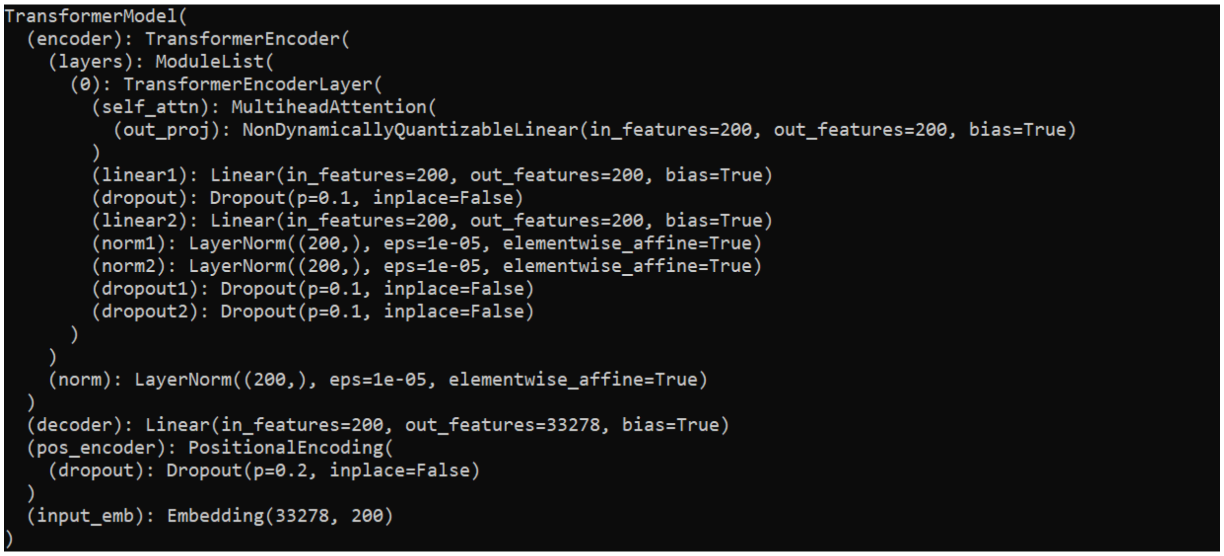
3. PURPOSE OF PROFILING

Profiling is an invaluable tool and has a plethora of benefits which help increase the general performance of the model.

- Performance Optimization- Profiling helps identify bottlenecks which can help in optimizing sections of the code resulting in better performance.
- Resource usage- Deep learning models make use of both CPU and GPU computing.Profiling plays a crucial role in understanding the allocation of memory resources. This helps engineers ensure that relevant parts of the code are running on the correct resources.
- Scalability- Profiling helps determine scalability and how the application will perform with increasing workloads.
- Cost optimization- If deep learning models are running on the cloud, profiling also helps estimate the compute and memory units required which helps determine the cost and bill of cloud services. This can help for businesses who are deploying models in commercial applications.

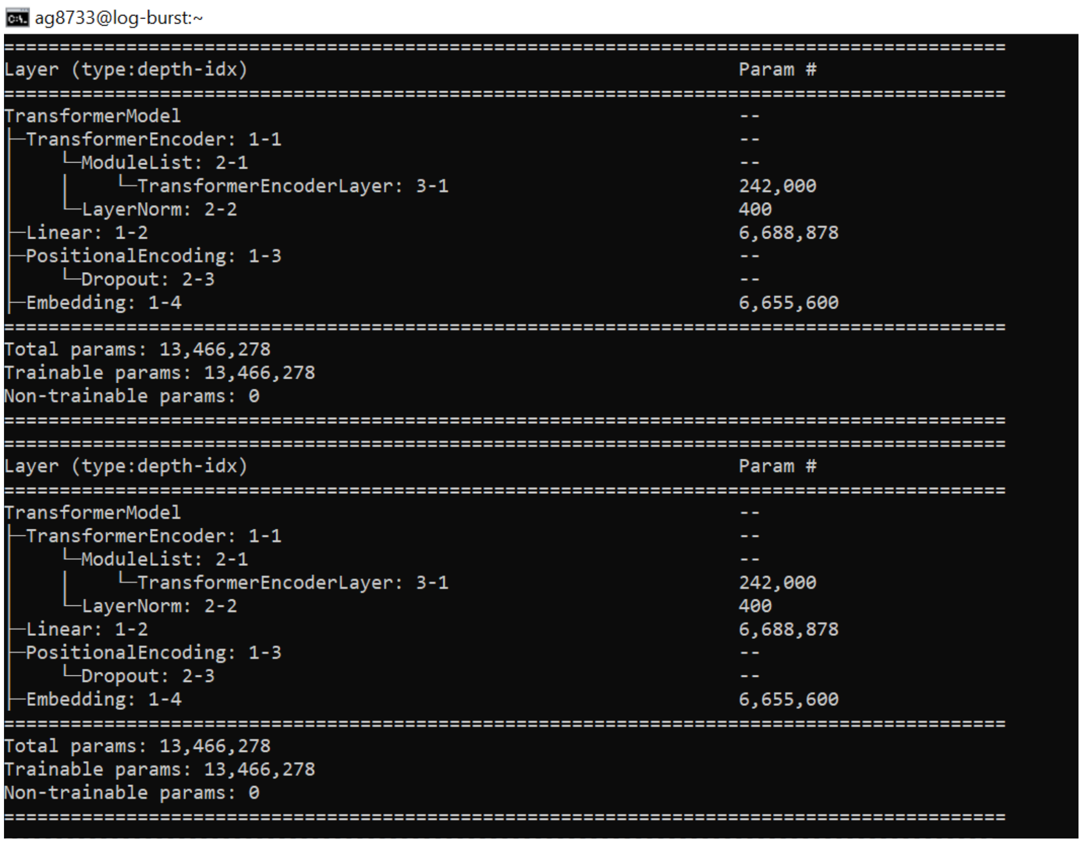
4. DESCRIPTION OF TRANSFORMER USED

I made use of the word-language model transformer . The model architecture and model summary are shown below.



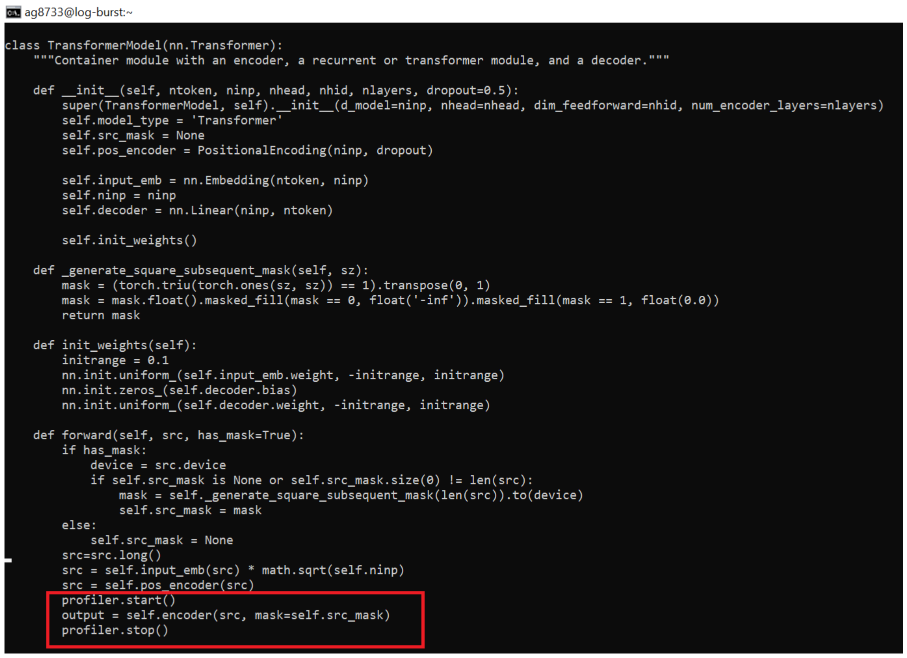
```
TransformerModel(
  (encoder): TransformerEncoder(
    (layers): ModuleList(
      (0): TransformerEncoderLayer(
        (self_attn): MultiheadAttention(
          (out_proj): NonDynamicallyQuantizableLinear(in_features=200, out_features=200, bias=True)
        )
        (linear1): Linear(in_features=200, out_features=200, bias=True)
        (dropout): Dropout(p=0.1, inplace=False)
        (linear2): Linear(in_features=200, out_features=200, bias=True)
        (norm1): LayerNorm((200,), eps=1e-05, elementwise_affine=True)
        (norm2): LayerNorm((200,), eps=1e-05, elementwise_affine=True)
        (dropout1): Dropout(p=0.1, inplace=False)
        (dropout2): Dropout(p=0.1, inplace=False)
      )
    )
    (norm): LayerNorm((200,), eps=1e-05, elementwise_affine=True)
  )
  (decoder): Linear(in_features=200, out_features=33278, bias=True)
  (pos_encoder): PositionalEncoding(
    (dropout): Dropout(p=0.2, inplace=False)
  )
  (input_emb): Embedding(33278, 200)
)
```

Figure 2: Model architecture using print(model)



```
ag8733@log-burst:~
==========================================================================================
Layer (type:depth-idx)                                                  Param #
==========================================================================================
TransformerModel                                                        --
├─TransformerEncoder: 1-1                                               --
│    └─ModuleList: 2-1                                                  --
│    │    └─TransformerEncoderLayer: 3-1                                242,000
│    └─LayerNorm: 2-2                                                   400
├─Linear: 1-2                                                           6,688,878
├─PositionalEncoding: 1-3                                               --
│    └─Dropout: 2-3                                                     --
├─Embedding: 1-4                                                        6,655,600
==========================================================================================
Total params: 13,466,278
Trainable params: 13,466,278
Non-trainable params: 0
==========================================================================================
==========================================================================================
Layer (type:depth-idx)                                                  Param #
==========================================================================================
TransformerModel                                                        --
├─TransformerEncoder: 1-1                                               --
│    └─ModuleList: 2-1                                                  --
│    │    └─TransformerEncoderLayer: 3-1                                242,000
│    └─LayerNorm: 2-2                                                   400
├─Linear: 1-2                                                           6,688,878
├─PositionalEncoding: 1-3                                               --
│    └─Dropout: 2-3                                                     --
├─Embedding: 1-4                                                        6,655,600
==========================================================================================
Total params: 13,466,278
Trainable params: 13,466,278
Non-trainable params: 0
==========================================================================================
```

Figure 3: Model Summary using torchinfo.summary(model,inputs)


ag8733@log-burst:~

```
class TransformerModel(nn.Transformer):
    """Container module with an encoder, a recurrent or transformer module, and a decoder."""

    def __init__(self, ntoken, ninp, nhead, nhid, nlayers, dropout=0.5):
        super(TransformerModel, self).__init__(d_model=ninp, nhead=nhead, dim_feedforward=nhid, num_encoder_layers=nlayers)
        self.model_type = 'Transformer'
        self.src_mask = None
        self.pos_encoder = PositionalEncoding(ninp, dropout)

        self.input_emb = nn.Embedding(ntoken, ninp)
        self.ninp = ninp
        self.decoder = nn.Linear(ninp, ntoken)

        self.init_weights()

    def _generate_square_subsequent_mask(self, sz):
        mask = (torch.triu(torch.ones(sz, sz)) == 1).transpose(0, 1)
        mask = mask.float().masked_fill(mask == 0, float('-inf')).masked_fill(mask == 1, float(0.0))
        return mask

    def init_weights(self):
        initrange = 0.1
        nn.init.uniform_(self.input_emb.weight, -initrange, initrange)
        nn.init.zeros_(self.decoder.bias)
        nn.init.uniform_(self.decoder.weight, -initrange, initrange)

    def forward(self, src, has_mask=True):
        if has_mask:
            device = src.device
            if self.src_mask is None or self.src_mask.size(0) != len(src):
                mask = self._generate_square_subsequent_mask(len(src)).to(device)
                self.src_mask = mask
        else:
            self.src_mask = None
        src=src.long()
        src = self.input_emb(src) * math.sqrt(self.ninp)
        src = self.pos_encoder(src)
        profiler.start()
        output = self.encoder(src, mask=self.src_mask)
        profiler.stop()
```

Figure 4: Torch.cuda.Profiler added around encoder block of Transformer in the Forward Pass of Transformer Class

5. ESTIMATION & PROFILING

FLOPS or Floating point Operation is any basic mathematical operation which includes addition,subtraction,multiplication or division.
MAC or Multiply accumulate operation is a multiplication and an addition operation. For example: a*b+c. A MAC counts as two floating point operations.

A. PEN AND PAPER ESTIMATION

We can define the MAC of a layer as follows-

$MAC_{\text{linear layer}} = batch\_size \ * \ seq\_len \ * \ embed\_size \ * \ embed\_size$

$MAC_{\text{LayerNorm}} = batch\_size \ * \ seq\_len \ * \ embed\_size \ * \ embed\_size$

$MAC_{\text{Multihead-attention}} = batch\_size \ * \ seq\_len \ * \ embed\_size \ * \ embed\_size \ * \ nheads$

$FLOP_{\text{layer}} = 2 \ * \ MAC_{\text{layer}}$

Hyper Parameters used are as follows:
Sequence length = seq_len = 35
Embedding size= embed_size = 200

Number of heads in muti attention layer = nhead =2
My encoder block consists of the following layers:Multihead attention, Linear1, linear2, Norm1,Norm2 followed by another LayerNorm. There are a total of six layers.

$MAC_{\text{total}} = 2 * MAC_{\text{linear layer}} + 3 * MAC_{\text{LayerNorm}} + MAC_{\text{Multihead-attention}}$
=
$= batch\_size * 7 * 35 * 200 * 200$
$= batch\_size * 98 * 10\text{^}5$

$FLOP_{\text{total}} = 2 * MAC_{\text{total}}$
$= 2 * batch\_size * 98 * 10\text{^}5$
$= batch\_size * 196 * 10\text{^}5$

(MAC is the memory used by the model)
Experiments were carried out for the following batch_sizes = 16,32,64,128,256

Batch_size	**MAC/Memory=batch_size *98 * 10^5 (bytes)**	**FLOPS= batch_size * 196 * 10^5**
16	156.8 * 10^6	313* 10^6
32	313.6 * 10^6	627.2* 10^6
64	627.2 * 10^6	1254.4* 10^6
128	1254.4 * 10^6	2508.8* 10^6
256	2508.8 * 10^6	5017.6* 10^6

Table 1: Estimated MAC and FLOPS

B. MEASUREMENTS USING NCU

- The Profiling was performed using Nvidia NCU tools, and the following arguments were given
  - --profile-from-start off : This ensures that the profiling doesn't start at application start time and instead when specified.
  - -- metrics : These are used to specify which metrics to measure when the kernels are launched. The metrics measured are 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 and dram__bytes.sum
    - --target-processes all
    - <command to run job> Here we provide the command to run main.py which executes the model on the Val data
      `python3 main.py --cuda --epochs 1 --nlayers 1 --model Transformer --lr 5 --batch_size 16`

- For Profiling with NCU the following command was used

```
/share/apps/cuda/11.1.74/bin/ncu --profile-from-start off --metrics
smsp__sass_thread_inst_executed_op_fadd_pred_on.sum,smsp__sass_thread_ins
t_executed_op_fmul_pred_on.sum,smsp__sass_thread_inst_executed_op_ffma_pr
ed_on.sum,dram__bytes.sum python main.py --cuda --epochs 1 --nlayers 1
--model Transformer --lr 5 --batch_size 16
```

- For each batch_ size, the logs of the NCU profiler were collected. (Logs can be found in folder)
- For each batch_size experiment the FLOPS and DRAM was calculated
    - $FLOP_{\text{kernel}}$ = 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

    - $Memory_{\text{kernel}}$ = dram__bytes.sum()
    - $FLOP_{\text{total}} = \sum_{all\ kernels} FLOP_{\text{kernel}}$
    - $Memory_{\text{total}} = \sum_{all\ kernels} Memory_{\text{kernel}}$

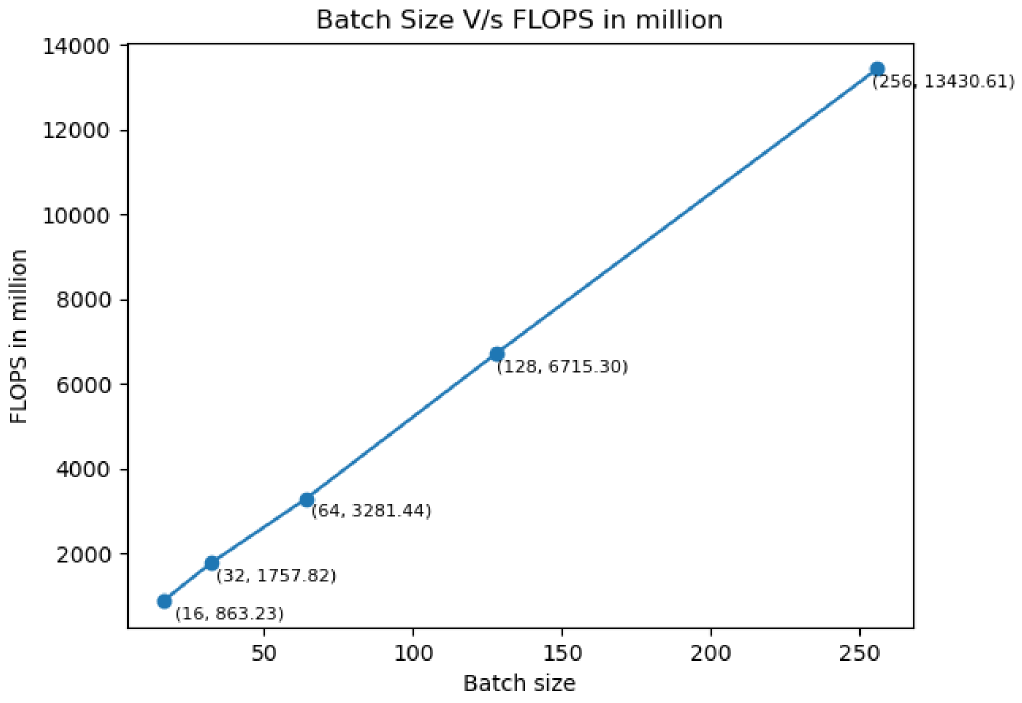
Using the logs and the above mentioned formulas the following Memory and FLOPS were calculated for each batch_size.

Batch_size	MEMORY/DRAM (Kbytes)	FLOPS
16	15524.78	863.23 * 10^6
32	30694.02	1757.81* 10^6
64	62967.54	3281.44* 10^6
128	1461010	6715.3* 10^6
256	362920	13430.61* 10^6

Table 2: Measured Memory and FLOPS

6. ANALYSIS

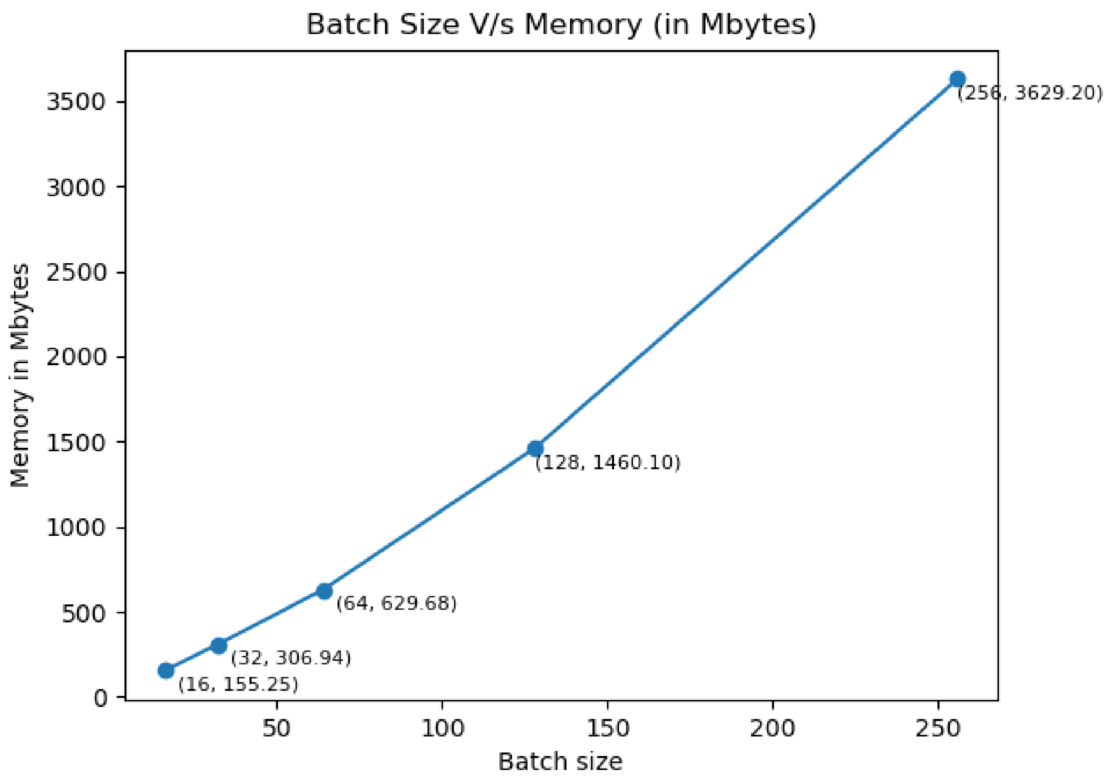
A. Trends of Batch Variations and Measured FLOPS



Graph 1: Batch Size V/s Measured FLOPS

Graph 1 shows the relationship between the batch-size and the measured FLOPS. From *Graph 1* we can see that the batch size and measured FLOPS increase linearly. In a transformer, we have linear layers,attention and also Layer Norm layers. As the batch_size increases the amount of data being processed in parallel increases. Hence, with an increase in batch_size , there is an increase in the matrix size and hence the floating point operations in the forward pass also increase. This means that with an increase in batch_size, in every second the number of computations increases. This shows that with an increase in batch_size, often the model evaluation happens quicker. However, we must also keep in mind the GPU capabilities.We can't infinitely keep on increasing the batch_size as we might hit GPU capability thresholds in which case the GPU won't be able to do everything parallelly. In this case the GPU will schedule these computations and so the FLOPS will again start to decrease for extremely big batch_size values.

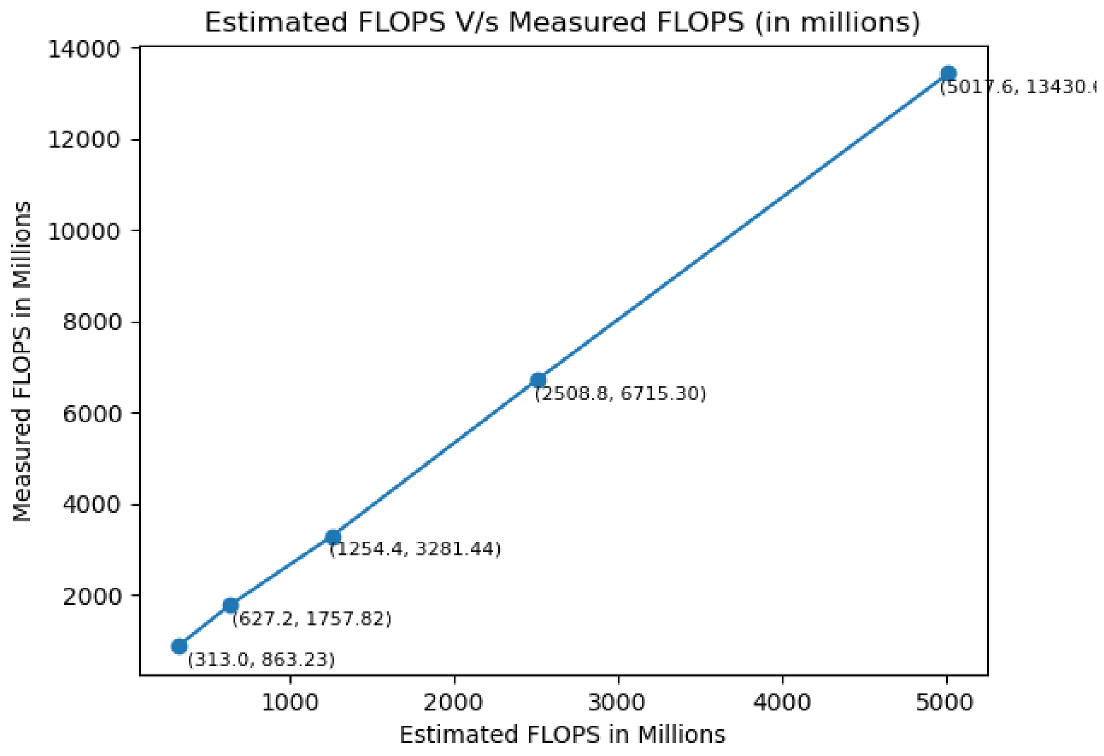
B. Trends of Batch Variations and Measured Memory



Graph 2: Batch Size V/s Measured Memory (DRAM)

Graph 2 shows the relationship between the batch-size and the measured DRAM. From *Graph 2* we can see that the batch size and measured memory increase linearly. As the batch_size increases the size of each matrix increases and hence the space required to store them also increases linearly.

C. Trends of Estimated FLOPS and Measured FLOPS



Graph 3: Estimated FLOPS V/s Measured FLOPS

Graph 3 shows the variations of estimated FLOPS and measured FLOPS. We can see a linear trend in these.
The slope of the line = m =
$(y2 - y1)/(x2 - x1) = (1757.82 - 863.23)/(627.2 - 313) \approx 2.84$

If the estimated and measured values had been equal the slope of this line would have been 1. Slope is 2.84 which shows the measured FLOPS are approximately 2.84 times the estimated FLOPS. Since the order of magnitude of difference of measured and estimated is 2.84, we can say that the pen and paper estimation and measured metrics from the NCU Profiler are similar and can be compared.

7. CONCLUSION

This report helped understand the benefits of profiling deep learning models using tools like Torch Profiler, Nvidia Nsight and Nvidia Nvprof. This also focused on understanding the encoder block of the transformer and hand estimating the complexity of problems. We analyzed that the measured metrics are comparable to the estimated metrics. Furthermore, as the batch size increases the measured FLOPS and memory also increases. While increasing FLOPS is a good thing, a balance must be set between the GPU capabilities and the batch size.

8. REFERENCES

1. Attention Is All You Need (https://arxiv.org/abs/1706.03762)
2. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale (https://arxiv.org/abs/2010.11929)
3. Transformers Explained (https://microsoft.github.io/code-with-engineering-playbook/machine-learning/ml-profiling/ https://towardsdatascience.com/transformers-explained-visually-part-1-overview-of-functionality)
4. Calculating MAC & FLOPS (https://medium.com/@pashashaik/a-guide-to-hand-calculating-flops-and-macs-fa5221ce5ccc)
5. NCU Documentation (https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html)
6. Pytorch Word-language-model (https://github.com/pytorch/examples/blob/main/word_language_model/model.py)