Guys Performing Transformations GPT

Group Name: AnimeLovers42
Names: Jerry Li, Pranesh Naagamuthu

Project Idea:
With LLMs being the popular trend right now, we thought it would be interesting to recreate a miniGPT within CUDA with an additional twist of course. Since all of the lectures were recorded we found it to be a perfect way to harvest free data through the transcripts of the lectures. Using Professor Wong's text inscribed speech, we first train a miniGPT (basic char decoder transformer) and then load those weights into a CUDA version of that model for inference only.

Acceleration from GPU:

Note that since we wrote inference-only code, we kept all operations within matrices since we didn't need the batch dimension.

Key parallelized components:

- Matrix Multiplication (sgemm) - Function that multiplies two different matrices into one matrix, a basic function necessary in all programs in our code. Matrix multiplication was improved upon for GPU usage in our implementation by making it use shared memory instead of global memory and tiling the matrix multiplication. This was the most commonly used operation throughout the entire project as operations like positional encoding, self-attention, and linear layers all utilized it.

- Softmax - Function utilized to compute row-wise softmax of a matrix, a necessary function for attention mechanisms. To optimize the softmax function for GPU usage, we made it apply reduction and utilized shared memory for best performance.

- QKV Projection - typically, the QKV projections are done separately as linear layers. Using tiled matrix multiplication and indexing, we combine all the QKV matrices into one (Block size, 3*head_dim*num_heads. to simulate the projection across heads. This greatly reduces the time needed to do it sequentially and to parallelize it all.

- layer norm - Similar to softmax, we apply shared memory and reduction to reduce the time needed.

The stage of the software pipeline is parallelized: The stage where most of the software is being parallelized is all of the fundamental features necessary for the program. Parallelizing the base functionality of the program is important since the speedup of the functions will speed up all the other major functions of the program. In our case the matrix multiplication will speed up all of our functions due to it being a core function needed for transformers, softmax speeds up our multi-attention head and generate functions, and QKV projection helps with memory capacity with multi-head attention. Additionally, other methods like reduction and coalescing were also used in operations like layer norm and softmax.

Implementation Timeline-

Week 6:
- Coded miniGPT in Python

Week 7:
- Coded LoadMatrix and tested the function

Week 8:
- Coded LayerNorm, Sgemm and Softmax

Week 9:
- Coded Self-Attention Head

Week 10:
- Coded MultiHeadAttention, generate function, layer norm

Week 11:
- Connected Transformer Decoder modules, MHA fix, generate function

Python Implementation: We implemented a baseline miniGPT model on the tiny Shakespeare dataset by utilizing a video tutorial. This implementation was important as it gave us the weights and logits required for inference functions to work in CUDA. The Python miniGPT also served as a baseline program that was used as a reference to build CUDA functionality.

Load Matrix - Implemented a load matrix function that will take in a file location of a .txt matrix and output a matrix float* that contains all of the inputs. This is necessary to load all of the weights and matrices into arrays for computation.

Layer Norm - Coded Layer Norm is a function used to normalize all the layers for better and faster results for inference. This was implemented by following the implementation videos done in Python for layer normalization and then converting the Python code into CUDA.

Sgemm - A matrix multiplication method that multiplies two matrices together. Implemented by using tiling and shared memory. We referenced the matrix multiplication lab to implement this function.

Softmax - Normalizes a matrix or vector of values into probabilities on a 0-1 scale. Used mainly in attention.

Self-Attention Head - Technically embedded into the MHA, but the self-attention mechanism or the core implementation of the transformer is implemented.

Generate - Implemented by following the same Python function as its name. Generates new tokens based on given prompts. Multiple functions were inside it, such as multinomial, which picks a token at random in a uniform distribution, and text to tokens, which converts the given prompt into tokens for processing. The generate function works by putting tokens into multiple different layers to generate more new tokens till the sequence length.

Multi-Head Attention - due to time constraints, we were not able to implement a more efficient method as it required batched sgemm and other clever indexing that required more testing. Ultimately, we took the combined projected matrix and then split it into separate Q, K, and V matrices and iterated through a for loop based on the number of heads. Afterwards, we concatenated the attention outputs together. This is essentially how the Python version works.

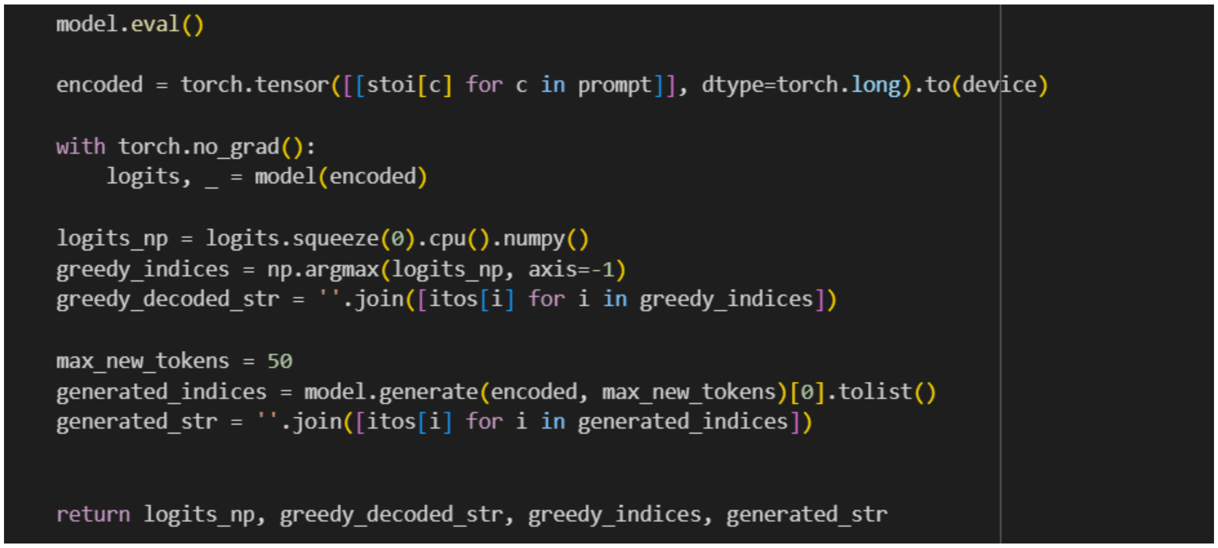
Transformer Decoder - Combines the layer norm, residual add, MHA, and linear project together as one module. This is not a CUDA kernel function but instead a normal function that calls multiple modules.

How to run the code:
Run the following two lines in the files :
./compile.sh

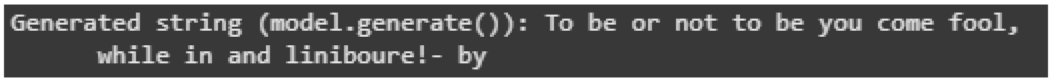
./minigptinfernce

Evaluation:
After implementing each feature, we systematically tested the correctness of the CUDA implementation by comparing its output to that of a referenced Python implementation. For each function, such as Layer Norm, positional encoding, and generate, we created dedicated test functions in both CUDA and Python. The functions accept the same inputs, and we compare the outputs are exactly or similar to each other. Apart from testing each of the functions separately, we connected each of the major functions to be tested if the output works properly. Below is an example of one test function we did for generate().

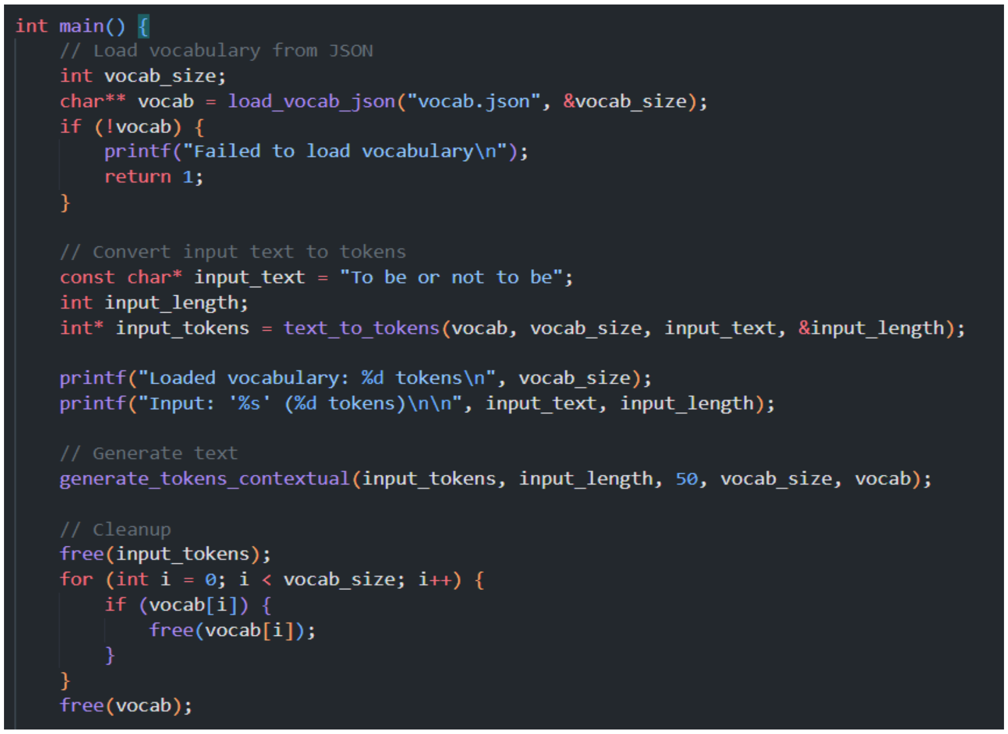
Example:



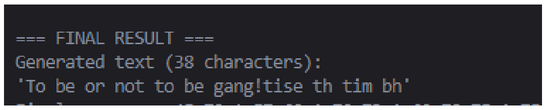
```python
    model.eval()

    encoded = torch.tensor([[stoi[c] for c in prompt]], dtype=torch.long).to(device)

    with torch.no_grad():
        logits, _ = model(encoded)

    logits_np = logits.squeeze(0).cpu().numpy()
    greedy_indices = np.argmax(logits_np, axis=-1)
    greedy_decoded_str = ''.join([itos[i] for i in greedy_indices])

    max_new_tokens = 50
    generated_indices = model.generate(encoded, max_new_tokens)[0].tolist()
    generated_str = ''.join([itos[i] for i in generated_indices])


    return logits_np, greedy_decoded_str, greedy_indices, generated_str
```



```
Generated string (model.generate()): To be or not to be you come fool,
     while in and liniboure!- by
```



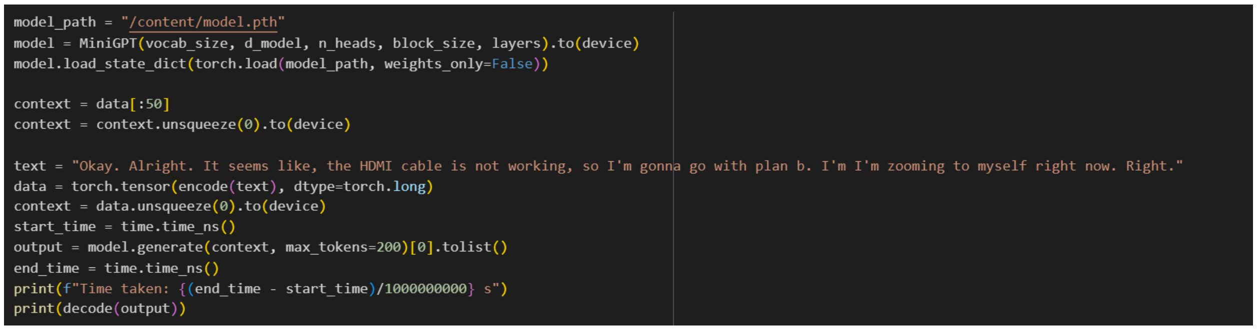
```c
int main() {
    // Load vocabulary from JSON
    int vocab_size;
    char** vocab = load_vocab_json("vocab.json", &vocab_size);
    if (!vocab) {
        printf("Failed to load vocabulary\n");
        return 1;
    }

    // Convert input text to tokens
    const char* input_text = "To be or not to be";
    int input_length;
    int* input_tokens = text_to_tokens(vocab, vocab_size, input_text, &input_length);

    printf("Loaded vocabulary: %d tokens\n", vocab_size);
    printf("Input: '%s' (%d tokens)\n\n", input_text, input_length);

    // Generate text
    generate_tokens_contextual(input_tokens, input_length, 50, vocab_size, vocab);

    // Cleanup
    free(input_tokens);
    for (int i = 0; i < vocab_size; i++) {
        if (vocab[i]) {
            free(vocab[i]);
        }
    }
    free(vocab);
```



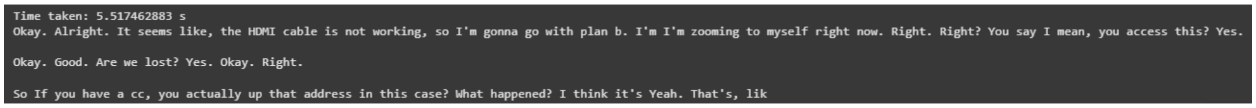
```
=== FINAL RESULT ===
Generated text (38 characters):
'To be or not to be gang!tise th tim bh'
```

Results:

Python Implementation: On average, the time it takes for the Python implementation to complete would be 5 seconds.

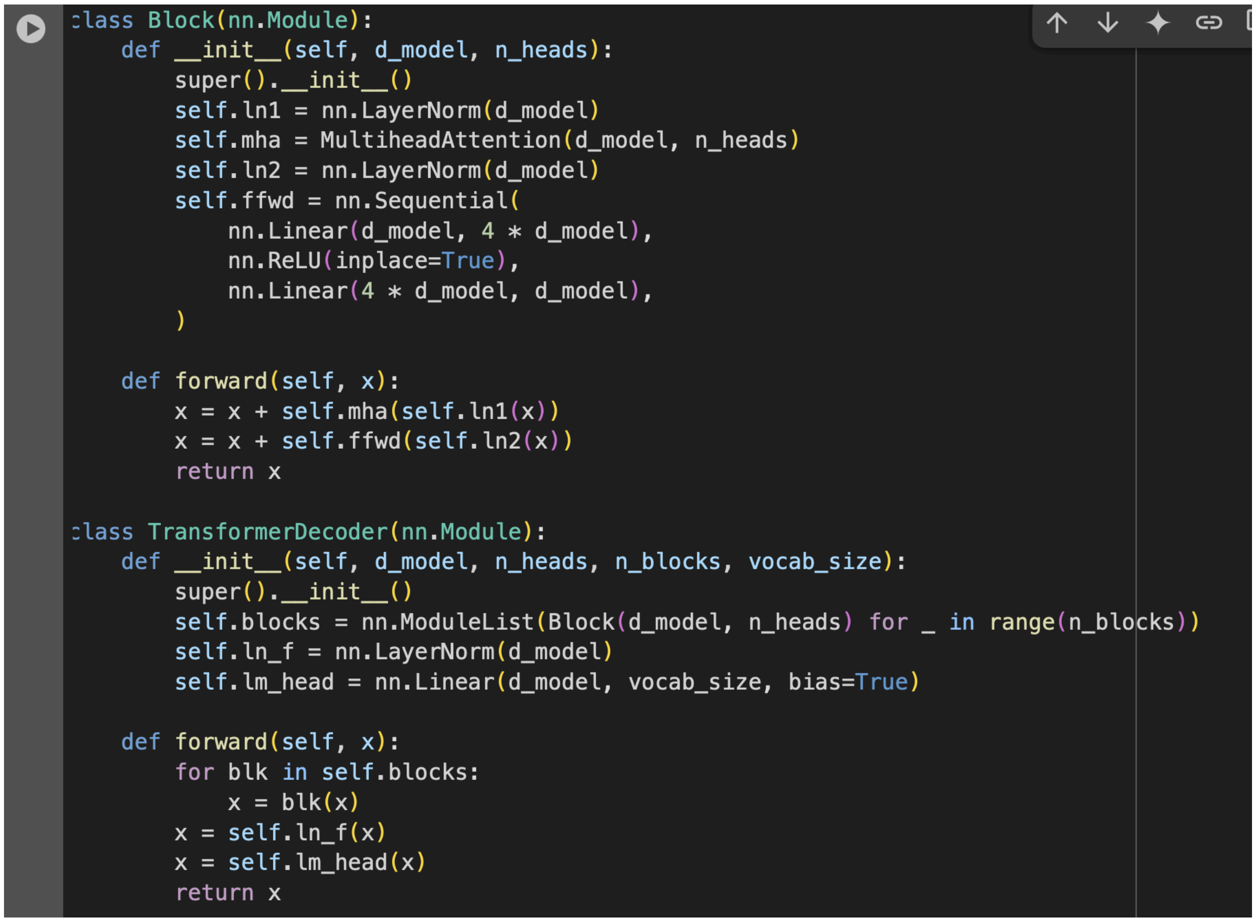


```python
model_path = "/content/model.pth"
model = MiniGPT(vocab_size, d_model, n_heads, block_size, layers).to(device)
model.load_state_dict(torch.load(model_path, weights_only=False))

context = data[:50]
context = context.unsqueeze(0).to(device)

text = "Okay. Alright. It seems like, the HDMI cable is not working, so I'm gonna go with plan b. I'm I'm zooming to myself right now. Right."
data = torch.tensor(encode(text), dtype=torch.long)
context = data.unsqueeze(0).to(device)
start_time = time.time_ns()
output = model.generate(context, max_tokens=200)[0].tolist()
end_time = time.time_ns()
print(f"Time taken: {(end_time - start_time)/1000000000} s")
print(decode(output))
```

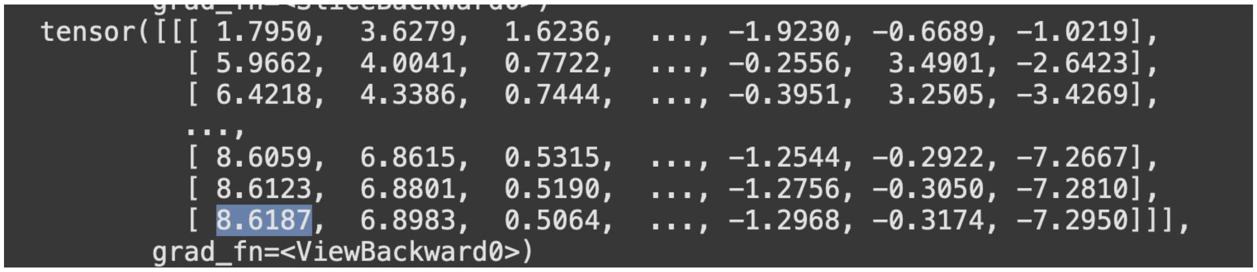


```
Time taken: 5.517462883 s
Okay. Alright. It seems like, the HDMI cable is not working, so I'm gonna go with plan b. I'm I'm zooming to myself right now. Right. Right? You say I mean, you access this? Yes.

Okay. Good. Are we lost? Yes. Okay. Right.

So If you have a cc, you actually up that address in this case? What happened? I think it's Yeah. That's, lik
```

Another example: we test the entire block by giving a hard set input on both Python and CUDA and see if the output are the same. Below is the correct logins which we matched in the cuda equivalent.

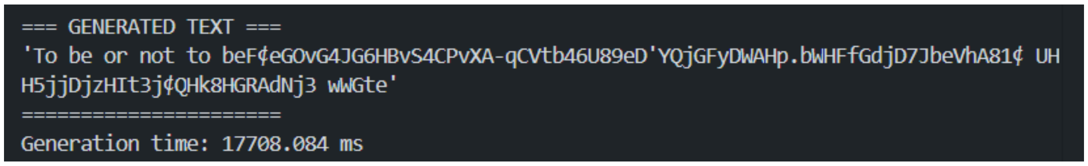


```
class Block(nn.Module):
    def __init__(self, d_model, n_heads):
        super().__init__()
        self.ln1 = nn.LayerNorm(d_model)
        self.mha = MultiheadAttention(d_model, n_heads)
        self.ln2 = nn.LayerNorm(d_model)
        self.ffwd = nn.Sequential(
            nn.Linear(d_model, 4 * d_model),
            nn.ReLU(inplace=True),
            nn.Linear(4 * d_model, d_model),
        )

    def forward(self, x):
        x = x + self.mha(self.ln1(x))
        x = x + self.ffwd(self.ln2(x))
        return x

class TransformerDecoder(nn.Module):
    def __init__(self, d_model, n_heads, n_blocks, vocab_size):
        super().__init__()
        self.blocks = nn.ModuleList(Block(d_model, n_heads) for _ in range(n_blocks))
        self.ln_f = nn.LayerNorm(d_model)
        self.lm_head = nn.Linear(d_model, vocab_size, bias=True)

    def forward(self, x):
        for blk in self.blocks:
            x = blk(x)
        x = self.ln_f(x)
        x = self.lm_head(x)
        return x
```



```
tensor([[[ 1.7950,  3.6279,  1.6236,  ..., -1.9230, -0.6689, -1.0219],
         [ 5.9662,  4.0041,  0.7722,  ..., -0.2556,  3.4901, -2.6423],
         [ 6.4218,  4.3386,  0.7444,  ..., -0.3951,  3.2505, -3.4269],
         ...,
         [ 8.6059,  6.8615,  0.5315,  ..., -1.2544, -0.2922, -7.2667],
         [ 8.6123,  6.8801,  0.5190,  ..., -1.2756, -0.3050, -7.2810],
         [ 8.6187,  6.8983,  0.5064,  ..., -1.2968, -0.3174, -7.2950]]],
       grad_fn=<ViewBackward0>)
```

Our Cuda implementation runs in 17 seconds, 13 seconds slower than the python implementation



```
=== GENERATED TEXT ===
'To be or not to beF¢eGOvG4JG6HBvS4CPvXA-qCVtb46U89eD'YQjGFyDWAHp.bWHFfGdjD7JbeVhA81¢ UH
H5jjDjzHIt3j¢QHk8HGRAdNj3 wWGte'
======================
Generation time: 17708.084 ms
```

Problems Faced:

- Generate function was a bit of trouble and took a long time due to not understanding the fundamental structure of it. The structure being, one of the functions is making randomness, resulting in new outputs even if using the same logits for the operations.
- Layer Norm was tricky since I found a blog that describes an optimized version of Layer Norm, which was tried to be implemented first but it was too advanced for implementation.
- The MultiHeadAttention was tricky since we had to turn it into a combined matrix to parallelize all at once. The main issue we had with figuring out the batched sgemm since we simply couldn't use the combined Q, K, and V matrices together, as that would produce a combined (block_size, block_size) matrix without the n_heads dimension.
- Combining the modules was tricky as each module had to be tested beforehand and the shapes had to be ensured to be correct between modules so they could be connected.

Function	Jerry Li	Pranesh Naagamuthu
Python Implementation	50%	50%
Sgemm.cu	100%	0%
Layer Norm.cu	0%	100%
softmax.cu	100%	0%
Transformer Block.cu	100%	0%
tools.cu	100%	0%
Generate	0%	100%
Positional Encoding	0%	100%
Forward	100%	0%
Report	25%	75%

Video
https://app.screencastify.com/watch/lVtGAZbRpPiI9vxzIkhr