Assignment 1: Optimizing LLMs Inference with KV-Caching

In this assignment, you will extend the nanoGPT codebase by Andrej Karpathy to optimize the performance of transformer inference through KV-caching.

Step 1: Understanding Transformer and Self-Attention Implementations

First, we will explore the implementation of transformer-based models. Please watch the let's build GPT: from scratch video, and answer the following questions:

- **Q1. (2 points):** In Karpathy's implementation of "self-attention" for one head, what are the dimensions of `q`,`k`,`v` matrices? Answer in-terms of `B`,`T`, `C`, `vocab_size`, `block_size`, `num_heads`
- **Q2. (2 points):** Complete the following code block to calculate the attention scores from the q and k tokens for all tokens. Recall that for "causal" attention, an "attention mask" matrix ensures that attention scores are only calculated for previous tokens and not future tokens.

```
tril = torch.tril(torch.ones(block_size, block_size))
mask = tril[:T,:T] == 0
wei = q @ k.transpose(-2,-1) * C**(-0.5)
```

- **Q3. (2 points):** What do you expect the value of mask to be when calculating the token at position i?
- **Q4. (4 points):** Write code to extract the value of mask from tril for calculating the token at position i.

For the rest of the assignment, use the provided colab to write code and run experiments. While the code can run either using the CPU or the free GPU version. Initially, use the CPU version for debugging and experimentation. Use the GPU version only for generating final graphs to avoid exceeding GPU usage limits.

Step 2: Benchmarking Token Generation (10 points)

Read through and run the code in step 2 of the colab. Submit your answers to the following questions:

- **Q5. (2 points):** Submit the generated graph.
- **Q6. (8 points):** What key trends do you observe in the per-token generation time?

Step 3: Implementing and Benchmarking KV caching (50 points)

What is KV-Caching?

The following is a snippet from the `generate` method in nanoGPT:

```
def generate(self, idx, max_new_tokens...):
    for _ in range(max_new_tokens):
        # Forward the model to get logits for the sequence
        # Extract the logits at the final step
        logits, _ = self(idx)
        logits = logits[:, -1, :]

        # Apply softmax to convert logits to probabilities
        # Sample from the distribution
        idx_next = torch.multinomial(F.softmax(logits, ...)...)

        # Append sampled index to the sequence and continue
        idx = torch.cat((idx, idx_next), ...)
    return idx
```

Here, the model computes logits for all previous tokens at every step, leading to an increase in computational load as the sequence grows. Only the logits from the final step are needed, while the rest are discarded. In this assignment, you will implement KV-caching to reuse intermediate values from previous steps to avoid recalculations and allow the model to compute logits for the latest token directly.

How Does KV-Caching Work?

Transformer models process input tokens by computing key (K), value (V), and query (Q) vectors for each token, which are used in the self-attention mechanism. Without KV-caching, the model recalculates K, V, and Q for the entire sequence at every step, even though only the newly generated token has changed. KV-caching resolves this by storing the K and V vectors for all previously generated tokens.

- Initial Step: When generating the first token, the model calculates K, V, and Q for the entire input sequence. The K and V values are stored in a cache.
- Subsequent Steps: For each new token, the model only computes K, V, and Q for the new token and appends them to the cached K and V values.
- Self-Attention: The cached K and V vectors are used for all previous tokens, while only the new K, V, and Q vectors are computed for the current token.

By reusing the cached K and V vectors, KV-caching ensures that the per-token generation time remains constant, regardless of the length of the input sequence. This makes token generation much more efficient, especially for long sequences.

Implementing KV-caching

You are provided with a code template containing placeholders marked as:

```
##### Solution Block #####
##### End Solution Block #####
```

Your task is to implement the missing functionalities within these blocks to enable KV caching in the transformer model. Each solution block has a unique identifier to help you navigate and organize your work effectively. Follow the detailed comments within each block to implement the required functionality. Hint: Implement and use `output_check` to validate the correctness of your code.

- **Q7. (40 points):** Submit your solutions for each block, along with brief explanations.

Benchmarking KV-caching

- **Q8. (5 points):** Run the benchmarking code `GPTWithCaching.generate_with_cache` and submit the generated graph?
- **Q9. (5 points):** Interpret the results based on KV-caching and discuss trade-offs compared to the original implementation?

Step 4: Submission & Private Evaluation (30 points)

- Submit the modified .ipynb file with changes only in the solution blocks, along with a report containing your answers to the above questions. Remember to generate the graphs using the T4 GPUs on colab.
- **(30 points)** We will run your submitted code on our GPUs to test the correctness and performance of your implementation.