Documentation: FastAPI Service for Optimized LLM with Medusa Head

Introduction

This documentation outlines the implementation of a FastAPI service for an optimized Language Model (LLM) with a Medusa head, using the lmsys/vicuna-7b-v1.3 model. The project aimed to enhance inference speed through model compilation, implement speculative decoding with a Medusa head, and utilize dynamic batching for efficient request handling.

Model Compilation Libraries Explored

1. Llama.cpp

Llama.cpp was successfully used to inference the Vicuna 7B v1.3 model and create a FastAPI serving application. This library was chosen for its efficiency in model quantization and optimization, particularly for running large language models on consumer hardware.

Advantages:

- Efficient quantization techniques
- Reduced memory footprint
- CPU-based inference capabilities

Limitations:

- Lack of support for speculative decoding with a Medusa head

2. TensorRT LLM

Attempts to use TensorRT LLM were unsuccessful due to dependency issues and memory constraints during the checkpoint conversion process. Able to run with a docker image, but lack of memory to run the checkpoint conversion script, and couldn't find pretrained checkpoints/

3. vLLM

vLLM was successfully implemented on Google Colab due to local memory constraints. This library was chosen for its efficient serving of LLMs and support for advanced features like speculative decoding.

Advantages:

- Support for quantization
- Batch inferencing capabilities
- Compatibility with Medusa head for speculative decoding

Limitations:

- High memory requirements, necessitating cloud-based GPU resources

Speculative Decoding Implementation and Advantages

Speculative decoding is implemented by using a smaller, faster "draft" model to generate multiple token predictions in parallel, which are then verified or corrected by the main LLM. This process works as follows:

1. The draft model generates a batch of tokens based on the input prompt.
2. The main LLM verifies these tokens in parallel, checking their probability against its learned distribution.
3. Accepted tokens are used in the final output, while rejected ones are corrected by the main LLM.

Advantages of speculative decoding include:

- Significant speedup in inference times (up to 2-3x improvement)
- Reduced computational load on the main LLM
- Identical output distribution to traditional decoding, maintaining quality
- Lower memory requirements, enabling on-device inference

Implementation Details

Model Compilation and Optimization

The Vicuna 7B v1.3 model was successfully quantized and optimized using vLLM on Google Colab. This process significantly reduced the model's memory footprint while maintaining performance.

Medusa Head Implementation

The Medusa head was implemented on top of the base model using vLLM. Speculative decoding with the Medusa head was achieved on an A100 40GB GPU. This implementation enhances performance by predicting multiple tokens in parallel, reducing the number of forward passes required.

Dynamic Batching

Dynamic batching was implemented to efficiently handle multiple concurrent requests. This approach allows for better utilization of GPU resources by processing multiple inputs simultaneously, leading to improved throughput.

Dynamic Batching Approach and Benefits for LLMs

Dynamic batching, also known as continuous batching, optimizes LLM inference by adjusting batch sizes in real-time based on current workload and request patterns. The approach involves:

- Adaptive batch formation based on system load and request patterns
- Concurrent processing of multiple requests
- Token-level batching instead of request-level batching

Benefits of dynamic batching for LLMs include:

- Improved throughput (up to 23x increase reported in some cases)
- Reduced latency, especially for varying output lengths
- More efficient resource utilization, particularly GPU memory
- Better handling of real-time, variable workloads

FastAPI Service

A FastAPI service was created for the Llama.cpp implementation, demonstrating the ability to serve the optimized LLM. However, due to time constraints and hardware limitations, a complete FastAPI service incorporating all features (Medusa head, dynamic batching) could not be fully realized.

Performance Benchmarks

Contrary to the expected performance improvements, our batched inferencing experiments showed an increase in inference time instead of a reduction. Specifically:

- Single prompt inferencing: approximately 50 seconds
- Batched inferencing with a concurrency of 5 prompts: approximately 150 seconds

This unexpected result highlights the complexity of implementing efficient batching strategies and the importance of careful optimization for specific hardware and model configurations. It suggests that further investigation and tuning of the batching implementation may be necessary to achieve the performance gains typically associated with dynamic batching techniques.

Challenges and Limitations

1. **Hardware Constraints:** Local hardware limitations necessitated the use of cloud resources (Google Colab) for certain implementations.

2. **Time Constraints:** The project was completed within a 3-day timeframe during working days, limiting the extent of feature implementation and testing.

3. **Compatibility Issues:** Integrating all desired features (model optimization, Medusa head, dynamic batching) into a single system proved challenging within the given timeframe.

Future Work

With additional time and resources, the following improvements could be made:

1. Full integration of all features (optimized model, Medusa head, dynamic batching) into a single FastAPI service.

2. Comprehensive performance benchmarking comparing different configurations.

3. Exploration of alternative quantization techniques to reduce GPU memory requirements.

4. Implementation of more advanced dynamic batching strategies for improved efficiency.

Conclusion

This project demonstrated the potential for significant performance improvements in serving large language models through optimization techniques, speculative decoding, and efficient request handling. While not all features were fully integrated due to time and resource constraints, the groundwork has been laid for a highly efficient LLM serving system.

The implemented components and exploration of various libraries provide valuable insights into the challenges and opportunities in optimizing LLM inference and serving. With further development, this system has the potential to offer state-of-the-art performance in LLM deployment.