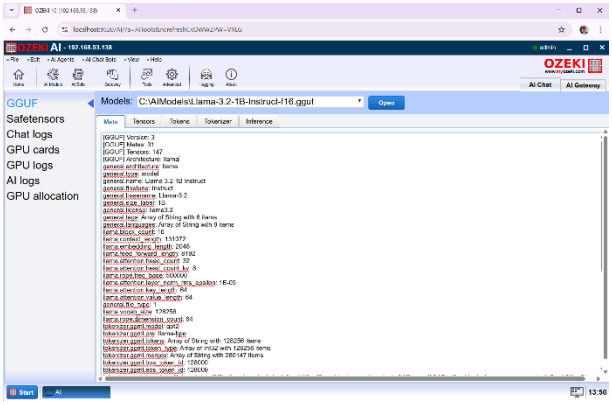
LLM Inference Engine

Introduction



In this project, a large language model inference engine is presented. This inference engine is capable of parsing GGUF files, tokenizing text, performing inference on the llama3 architecture and returning results. This project is still on-going with work currently focused on making the inference faster using hardware acceleration. The full project description is available at https://gyularabai.com/p_6919-ai-inference-engine.html.

How to Run?

Firstly, download a .gguf file that is in F16 and has the llama3.2 1 billion parameter architecture. Secondly, download the source-code from either my web-site (https://gyularabai.com/p_6919-ai-inference-engine.html) or my GitHub (https://github.com/mrgyularabai/AI-Inference-Engine). Finally, run the LLMInference project, and load the model you want to use to generate text using the console interface. The GUI solution was produced using proprietary software, so its source-code is not available.

Architecture

Model Construction

The model is constructed by first extracting the correct weights from the GGUF file, and then, constructing the model architecture from OzAIArchComp-s.

GGUF File Parsing

This involves first reading in the correct meta-data entries. Then, the headers for each tensor which contain the tensor's properties (dimensions, quantization etc.) are read. Finally, the tensors are identified and loaded into RAM.

Model Architecture

Next, a series of OzAIArchComp-s are chained together. These each contain either sub-components or calls to the ExecutionManager to do fundamental vector operations such

as vector addition or matrix multiplication (all tensor operations are broken down into vector and matrix operations). Together these architecture components form the architecture.

All of them have hyper-parameters and individual instance parameter, which are populated during the model construction phase.

Inference

Tokenization

First, the text is inputted into the model's tokenizer. This uses a novel approach, which approximates the true output token sequence instead of calculating it from merges. The approach is faster than many other implementations and is detailed further in this paper I wrote (https://gyularabai.com/p_9128-fast-inference-time-tokenization-through-approximating-bpe.html).

Embedding

This is a simple look up into the embedding matrix which is stored as a list of OzAIVector-s.

Other Architecture Components

It should serve as no surprise that an RMS Norm, followed by self-attention, followed by a residual connection, followed by a gated linear unit, followed by the next layer is repeated 16 times. Instead of detailing the architecture specifics, I will focus on a few unique aspects of my implementation. Firstly, all operations are performed on vectors not tensors. This mean that self-attention was broken down into groups, then heads, and finally the score calculations were all done in lower-and-lower-level architecture components instead of a few monolithic einsum calls to PyTorch.

Another interesting aspect of the implementation is that the same memory locations are reused, because no gradients have to be calculated and it would be wasteful to constantly reallocate MBs of memory.

Execution Manager

Finally, the way most vector calculations are done is they are given to the execution manager, which distributes all the operations amongst multiple executors running on different threads. The main reason this does not increase the performance drastically is because the architecture components are not assigning enough tasks at once for the parallelism to work. This is currently being worked on.

Motivation

The main motivation for this project was that I had been working on a similar project before large language modelling gained so much focus. I had a program where one would define a template for an article and define synonyms for different words or phrases. This program aimed at, someday, being able to generate text. Nonetheless, ChatGPT beat me to it. To understand how this novel technology works and make it more efficient, I decided to build my own inference engine.

Development

Unfortunately, the development journey cannot be fully captured as visually as it can for a game engine or a CPU design. References to currently existing files within the project will be made to illustrate the development process or changes made within each file. However, this project was not as strictly version controlled as the others.

My first attempt at making an engine that is capable of running neural networks was the neural network simulation on this GitHub (https://github.com/mrgyularabai/Neural-Network-Simulator). The first thing I did was I transferred this code into a separate project with the linear algebra library and got to work writing the parser for the GGUF file format (the file format of my choice, because it contained all the data in one file).

This work heavily involved analysing code from llama.cpp to understand the structure of the file format. Eventually, a representation of the file formed, but I encountered the problem of there already being at least twenty data-types for tensors form 8-bit K-mean quantization to brain-floats. I, therefore, made my a class to store all these different types of numbers. This attempt yielded the OzAINum class and all its inheritors. Here the properties for the different quantization types could be retrieved for each class, and each class could store its given data-type. Two functions necessarily implemented for all the OzAINum-s were casting to float32 and returning a byte array.

This was because my initial idea was to not implement all these different data types and to just cast them to a high precision data-type (float32) assuming no losses would be made. I also rewrote my linear algebra classes to store the data using the OzAINum classes, and each data-type had its own vector type.

The next thing I went on to create was the tokenizer. Implementing the trivial solution of remerging all the tokens in the text to get the desired output sequence was too slow, so I devised a new approach to BPE tokenization as detailed in the relevant previous section.

Afterwards, when I started implementing the architecture, I realized that each vector type would have to be able to operate with other data-types, and this was not going to be something I could implement in a reasonable time frame. Therefore, I abstracted the operations out away from the Vector classes into executor classes.

Anticipating the need for multithreading, I also made an execution manager class. This having happened, I removed the data storage from being done in an OzAINum for the vectors, and devised the current approach.

From here on, I built the architecture as detailed in the previous sections, however I encountered problems, when I first tried to run the model. The first problem was that the model simply could not execute without multithreading. It would take about an hour for one token to be generated. Therefore, before I could even test the architecture, I had to implement multi-threading.

The next error was that most vectors were slightly off from what they were supposed to be (+/- 0.4), when it came to unembedding. It turned out that casting everything to float32 introduced imprecision compared to what the model was trained on, which meant that it could only generate text if the same or a lower-precision data type is used (surprisingly). This meant I had to reimplement all the operations to take place in float16 (or float32 depending on what was used for the model.)

Last year, the first token was generated. Since, then I have been developing an approach where instead of each architecture component executing the components sequentially, the components build a compute graph of atomic vector operations, which can be executed with much greater parallelism.

Conclusion

Overall, this inference engine is still an on-going project with much more to learn still. However, along the way, I have gathered all forms of valuable skills such as understanding tensor operations and learning about multi-variate calculus. To help contribute to my community, I have also shared these skills with others in the forms of presentations and lectures. In the future, I hope to make the engine as performant as solutions like PyTorch, and learn even more along the way.