CS265 Project: Building LLM Inference Like Clockwork (Part 1/2)

Spring 2026

1 Overview

In this project, we implement a Large Language Model (LLM) from scratch for inference, without using external libraries. LLMs are typically accessed through high-level languages and frameworks such as PyTorch and Hugging Face. Although these frameworks significantly increase productivity, they are not suitable for our purposes as systems researchers and performance engineers. Most physical operators, computations, and data movement are abstracted into high-level logical operators; without understanding these low-level details, it is impossible to reason about performance or improve it.

In this project, we use a pretrained model and implement its inference pipeline using `C++` as the controller and `CUDA` for compute kernels. The project is organized into three milestones, described in detail in Section 3.

In the first milestone, you will build the infrastructure required to load model parameters and convert an input string into its corresponding numerical representation. In the second milestone, you will implement the core computational components required to process these representations on the GPU. In the final milestone, you will complete the implementation of a full decoder block and integrate all components into a functional end-to-end inference pipeline, as well as monitor the performance of your LLM. By the end of the project, you will have constructed a working LLM inference engine capable of generating tokens autoregressively. See Figure 1, and you exactly know where time goes.

Comparing our implementation with `PyTorch`, you can view this project as tailoring a garment rather than purchasing one from a factory: the factory product is convenient and refined, but its construction is largely opaque. An early tailored garment may be rougher than a factory equivalent; it may also have lower quality, but it can ultimately fit better because it is built around precise requirements and can be iteratively refined based on direct measurement and understanding.

The overview is intended to provide a high-level description of the project. Additional details and clarifications are provided in Section 2 and Section 3.

The outcome of this project is:

- Develop a deep understanding of hardware behavior and data movement on GPUs and inference pipelines.
- Gain rigorous insight into LLM operators and their performance trade-offs, and build technical intuition for key concepts in modern ML systems such as KV caching, Fusion, ML compilers, and related optimization techniques.
- Develop the ability to read and critically understand research papers in ML systems, and to formulate new research ideas in this area.
- Having implemented an LLM that is prepared for advanced LLM performance optimizations, including paged attention, sampling methods, and quantization.

In the remainder of this document, we provide a detailed description of the project structure, expectations, and technical requirements. We begin with logistical information and repository organization, followed by guidelines for setting up the development environment and understanding the provided starter code. This includes the required functionality, suggested implementation guidance, and evaluation criteria. Required functionality is the logic we require in the milestone, which are steps for our inference to work. We also provide a suggested guideline, which is not binding, and you may proceed with your own design as long as you provide test functionality and follow the required guidelines for our testing. Students are expected to read these sections carefully before beginning the implementation.

This is part 1 of the project that is required for midway check-in. We release part 2 in early March.

2 Logistics

Initial Code Repo. You can clone the starter code from here.[1] The repository includes utility scripts for downloading the model, as well as starter files intended to guide your implementation. You are free to modify the structure and design of your solution as needed. However, any file marked at the top with the tag `DO NOT CHANGE THIS` must remain unmodified. These files are used for evaluation and may be replaced during assessment.

Language, Compiler, Libraries. Your implementation must use `C++` for the controller logic and `CUDA` for all compute kernels. After downloading the model weights using the provided script, you are required to dump the weights

[1] https://code.harvard.edu/mir593/CS265-llm-starter

into binary files so that they can be loaded by your `C++` controller. `Python` may be used exclusively for this weight-dumping step, as it is more convenient for handling the original model format. No other components of the project may be implemented in `Python`.

You may use any standard library available in `C++`. For the `Python` script used to dump the model weights, you may use the libraries listed in the project under `requirements.txt`.

All compute kernels must be implemented by you in `CUDA`. While matching state-of-the-art performance is not required, we expect few optimization efforts: shared-memory tiling, double buffering, and memory coalescing. Extra optimizations like tensore core usage and vectorization will give you bonus points. You are encouraged to study existing high-performance implementations, such as `GEMM` kernels, to enhance your kernels.

Model and Model Repository. We provide a script for downloading and storing the model weights. This script requires you to set your Hugging Face access token as an environment variable named `HF_TOKEN`. To obtain this token, you must register on the Hugging Face website and request access to the model from Meta at here.[2]

You must save the downloaded model files in `./assets/llama3/`. This is required. During assessment, we will not download the model again; instead, we will assume that the model files are located in this directory.

Collaboration with Your Peers and AI Tools. You are expected to write and fully understand your own implementation. You may discuss ideas with your peers and use coding assistants or AI-based tools; however, you remain responsible for all design and implementation decisions in your code.

You must be able to explain any component of your implementation, including performance-related optimizations and specific language or syntax choices. During evaluation, we may ask you to justify or analyze any part of your code.

Grading & Evaluation. The project grade has three components: final code review (35%), midway check-in (10%), and automated testing (10%). The midway check-in is described in the following paragraph. At the end of the semester, we will conduct an individual code review meeting, in which you are expected to explain your code architecture, implementation details, and design decisions. You must demonstrate a clear understanding of both correctness and performance-related aspects of your implementation.

Testing . You can find examples of automated tests in the `test/` directory. You are required to implement all functionalities specified in `test/test_api.h`. Our test suite uses the primitives defined in this file for both unit and end-to-end testing. Correct functionality with reasonable performance is required

[2] https://huggingface.co/meta-llama/Meta-Llama-3-8B-Instruct

to receive credit. Additional performance optimizations may positively influence your evaluation during the code review. Passing the final code review is mandatory to receive credit for the automated testing component.

Midway Check-In. A midway check-in is scheduled to ensure that you begin the project early and make sufficient progress toward completion. During this check-in, you will have a one-on-one meeting with one of the TFs.

At this stage, you are expected to have completed all required functionality for Milestone 1, including model loading, tokenization, and embedding lookup. In addition, you must have implemented your matrix multiplication kernel. The purpose of this meeting is to verify progress, identify potential issues early, and provide feedback on your implementation.

References. To become familiar with GPU programming, we strongly encourage you to read the first six chapters and Chapter 10 of Programming Massively Parallel Processors [2]. In addition, numerous online resources are available to help you learn `CUDA` syntax and programming practices.

The original Llama 3 paper [1] provides background information about the model; however, it does not contain sufficient implementation detail for this project. For a precise understanding of the architectural structure and operator-level logic, the most reliable reference is the `Python` implementations available in `HuggingFace` and `PyTorch`. The papers assigned in this course will reinforce your understanding of the project, and the project will also deepen your understanding of the papers.

We will hold two lectures dedicated to LLM inference and the project on February 19 and February 24. The first lecture will focus on GPU architecture and performance considerations. The second lecture will cover LLM operators and related optimization techniques.

3 Milestones

3.1 Milestone 1: Loading Model, Tokenization, Token IDs to Embeddings

3.1.1 Desired Functionality

This milestone establishes the input and parameter pipeline used throughout the project. We use the following terminology:

- **Prompt:** The input string provided to the model.
- **Tokenizer:** The component that converts a prompt into a sequence of integer **token IDs**.
- **Token IDs:** Integers representing tokens in the model vocabulary. We represent size of the sequence by s as you see in Figure 1.

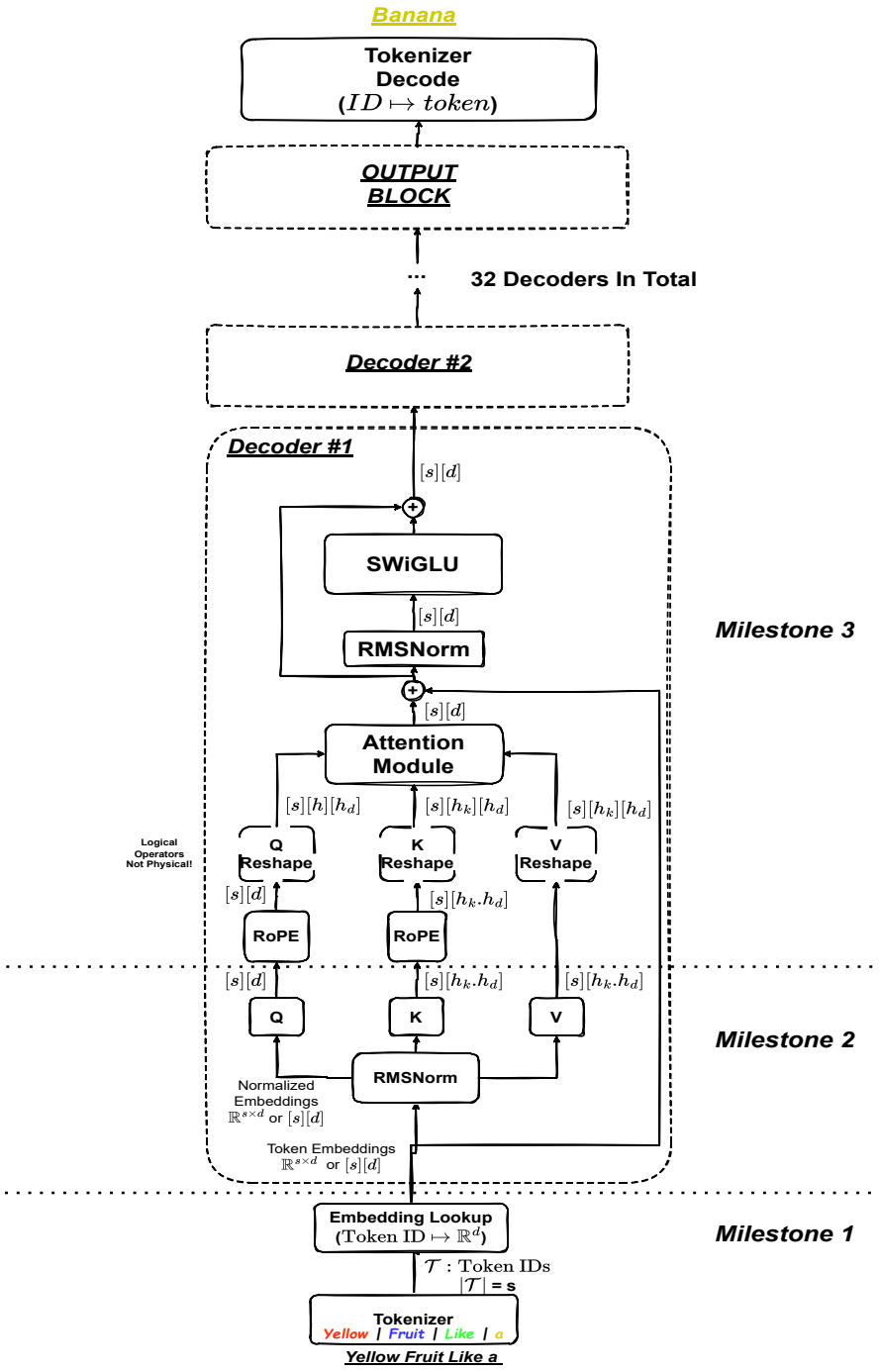

Figure 1: Architecture of Language Model for our Project. Your Matrix Multiplication Kernel is also included in the Midway check-in.

- **Embedding Table:** A matrix that maps each token ID to a token embedding vector. You can view this as a lookup table.
- **Token Embeddings:** The vectors used as the initial input to the model computation. In our model embedding size is 4096, and in Figure 1, we denote it by d.

First, use the provided script to download the model weights and tokenizer into `assets/llama3/`. In this milestone, you must complete the following steps:

1. **Step 1: Download assets.** Ensure that the model weights and tokenizer files are downloaded and stored in `assets/llama3/`.
2. **Step 2: Tokenize prompts.** Implement tokenization such that a prompt (e.g., `Hello world`) is converted into a sequence of token IDs (e.g., `[128000 , 9906, 1917]`). You are not required to insert special tokens in this milestone; special-token handling will be specified later.
3. **Step 3: Dump and load weights.** Implement a `Python` dumper script (the only `Python` component in the project) and the corresponding loader in the `C++` controller. The goal is to extract the embedding table and all required model parameters from the `.safetensors` files and make them available to the `C++` implementation.
4. **Step 4: Embedding lookup.** Given the token IDs from Step 2, implement an embedding lookup to produce the corresponding sequence of token embedding vectors.
5. **Step 5: First `CUDA` Kernel, Matrix Multiplication.** After producing token embeddings, implement a `CUDA` matrix multiplication kernel. Although the first operator used in the model is `RMSNorm`, you must implement matrix multiplication first because it will be reused extensively throughout later milestones. Your kernel is expected to include basic optimizations such as tiling, shared-memory reuse, and coalesced global-memory(hbm) accesses. These optimizations are mandatory, and any other optimization will give you bonus points.

By the end of this milestone, the system should accept a prompt, tokenize it into token IDs, and produce the corresponding token embeddings. These token embeddings serve as the input to the GPU computation in subsequent milestones. You also implemented your matrix multiplication, which is the backbone of most of the operators.

Assumption and Simplifications: As illustrated in Figure 1, the implementation in this project operates on a single input sequence (prompt). You are not required to implement batching or KV caching later in part2. Both batching and KV cache support are considered optional extensions and may receive up to 5% additional credit. You may assume that the input sequence length will not exceed 1000 tokens.

3.1.2 Suggested Guideline (Not Mandatory)

To download the model weights, register on Hugging Face and request access to Llama. Ensure that the downloaded files are stored in the directory specified by the downloader script. The tokenizer API is defined in include/tokenizer.h. We also provide a partial tokenizer implementation in src/. The missing component that you must implement is indicated in the starter code and should require approximately ten lines of code. After completing this step, you should be able to convert a prompt into token IDs using encode() and convert token IDs back into text using decode()[3].

After downloading the model weights and setting up the tokenizer, you must implement loading for the model parameters and the embedding table in your C++ controller. The model consists of parameters for multiple layers and operators, which must be loaded and transferred to the GPU for computation. These parameters are provided in .safetensors files. The file model.safetensors.index.json lists the parameter names and the corresponding .safetensors shards that store them.

You may implement a native C++ loader for .safetensors; however, this is not required. Instead, you may write a single-file Python dumper that extracts the required tensors and writes them to one or more binary files in a format of your choice. Your C++ controller should then load these binary files, for example, using mmap (as in CS165). Your dumper must be located at tools/dumper.py, and it will be executed prior to assessment. We recommend designing a simple binary format that begins with a fixed-size header containing metadata (e.g., tensor name, shape, dtype) followed by the raw tensor data. You may choose to dump parameters per operator or per layer, and you should dump the embedding table separately.

Note that the model parameters are stored in BF16. You may dump and load parameters in FP32 for simplicity, and optionally convert to half precision before transferring to the GPU. You may implement the project using either BF16 or FP32, but BF16 is encouraged.

After implementing the dumper and loader, you should be able to load the embedding table, which contains one vector in R^d per token ID. For our Llama-3-8B-Instruct, $d = 4096$. Given a 1D sequence of token IDs, your embedding lookup should produce the corresponding 2D array of token embeddings. This output will serve as the input to your RMSNorm, which we do in the next part.

As the final step of this milestone (and as part of the midway check-in), you must implement a general matrix multiplication (GEMM) kernel as your first CUDA kernel. At a minimum, your implementation should use tiling, shared-memory reuse, and coalesced global-memory accesses. For guidance on baseline CUDA GEMM design patterns, see Programming Massively Parallel Processors [2]. We also discuss in the GPU lecture how to reason about GPU hardware behavior and performance bottlenecks (e.g., memory bandwidth, occupancy, and data movement), which should inform your optimization choices.

[3] This functionality is used later when decoding the model output into a text response.

You are strongly encouraged to verify correctness incrementally. Small errors in early components can propagate and become difficult to diagnose later. When debugging, you probably want to compare intermediate outputs against a reference implementation in PyTorch.

3.1.3 Bonus Thought Experiments & Questions[4]

- In tokenizer code, you see some special tokens such as <|start_header_id|> or <|begin_of_text|>. Check what these tokens are, why they are necessary, and how to benefit from each token.
- Our tokenization determine our sequence length and hence directly affect our time and memory. You may notice that our tokenizer may even use multi-tokens for a single word. What are the benefits and drawbacks of fine-grained tokenization? Why we do not use one token id for each word?
- Loading all model weights can be time-consuming. Although it is a one time job, do you know application that are bottlenecked by this and also think about ways to speed up this step.
- Think of where is the best place for storing embedding table. CPU or GPU? What are the trade-offs?
- Moving weights from disk to CPU and from CPU to GPU seems suboptimal. Do you know any tricks/hack that can make this process faster?

References

[1] W. Bouaziz, X. Constable, X. Tang, X. Wu, X. Xia, Y. Gao, Y. Chen, Y. Hu, Y. Jia, Y. Qi, Y. Adi, Y. Nam, Y. Wang, et al. The llama 3 herd of models, 2024.

[2] Wen mei W. Hwu, David B. Kirk, and Izzat El Hajj. *Programming Massively Parallel Processors: A Hands-on Approach*. Elsevier, 4th edition, 2022.

[4] Extra bonus of 5% of the project grade per milestone.