Optimizing LLM Inference with Speculative Decoding and Quantization

Ryan Huang
Computer Science Department
Columbia University, New York, NY
rh3129@columbia.edu

Michael Khanzadeh
Computer Science Department
Columbia University, New York, NY
mmk2258@columbia.edu

Niranjan Sundararajan
Computer Science Department
Columbia University, New York, NY
ns3888@columbia.edu

Abstract—**Large language models (LLMs) deliver impressive generative performance but incur high inference latency and memory cost, limiting real-world deployment. We study two complementary acceleration techniques, *speculative decoding* and *weight-only quantization*, on the open-weight LLaMA-3 family. Using an NVIDIA L4 GPU, we benchmark an 8-billion-parameter target model paired with 1B and 3B draft models, exploring confidence thresholds, assistant-token lengths, and sampling modes. We also evaluate pure 16-/8-/4-bit quantization and mixed-precision draft/target combinations.**

With full-precision weights, speculative decoding achieves up to 1.35× throughput speed-up at an optimal setting of $\tau = 0.2$, $k = 3$, and greedy decoding, while preserving output quality. The smaller 1B draft outperforms the 3B draft despite lower acceptance rates, confirming that draft efficiency outweighs marginal quality gains. Standalone 4-bit quantization yields a further 35% latency reduction and maintains pass@1 accuracy within 3.7% of FP16 on HumanEval. 8-bit shows minimal accuracy loss but under-performs in throughput on L4 hardware. Combining speculative decoding with low-precision drafts increases rollbacks and erodes speed-up, highlighting a quality–latency trade-off and inadequate INT8/INT4 kernel support on L4 GPUs.

Our results demonstrate that speculative decoding is a practical path to faster LLM inference, whereas quantization benefits are highly hardware-dependent. Future work on A100/H100 GPUs with larger draft/target gaps (e.g., 8B→70B) may unlock effective mixed-precision speculative decoding.

Index Terms—**Large language models, speculative decoding, weight quantization, inference acceleration, GPU computing.**

I. Introduction

A. Background and Motivation

Modern large language models (LLMs), with some exceeding over a trillion parameters, demand enormous computational resources. In particular, they require substantial VRAM to load and run efficiently, and their inference speed becomes increasingly limited as model size grows. One key bottleneck in LLM inference is the autoregressive nature of generation—each token is produced one at a time, with a full forward pass required per token. This introduces a clear opportunity for optimization.

Speculative decoding is one such method that addresses this inefficiency. It introduces a lightweight draft model to generate multiple tokens in a single step, which the target model then verifies. Because the target model only accepts tokens it would have generated itself, the quality of output is preserved while improving throughput. However, speculative decoding alone does not address the growing memory footprint of LLMs.

To reduce memory consumption, quantization lowers the precision of weights and activations, from 16-bit to 8-bit or 4-bit, allowing models to run with less VRAM. While quantization can sometimes degrade model accuracy, it also provides faster inference due to lower-precision arithmetic. There is opportunity for the exploration of mixed-precision speculative decoding, which combines the benefits of both approaches.

B. Problem Statement

Despite significant advancements in the capabilities of LLMs, inference remains a major bottleneck due to high memory consumption and latency. Most state-of-the-art models still generate outputs token-by-token using large memory-intensive weights which hinders practical deployment in real-world scenarios. There is a critical need to accelerate LLM inference and reduce memory requirements without significantly compromising model quality. This project addresses this challenge by investigating methods to speed up LLM inference and minimize memory usage while maintaining strong accuracy. In particular, we explore speculative decoding, quantization, and the combination of both techniques to assess their effectiveness in improving inference efficiency.

C. Objectives and Scope

This paper aims to optimize LLM inference by exploring speculative decoding and quantization techniques. Specifically, we:

- Explore how varying speculative decoding parameters (confidence thresholds, number of assistant tokens, and sampling) impact performance.
- Analyze the effect of quantizing the draft and/or target model in speculative decoding settings.
- Evaluate quantization alone (without speculative decoding) to establish a baseline for comparison.
- Measure latency, throughput, rollback behavior, acceptance rate, and output accuracy using the WikiText2-raw-v1 dataset and LLaMA3-based models.

By comparing all results to a non-optimized baseline (no speculative decoding, no quantization), we aim to understand the trade-offs involved and identify configurations that balance speed, memory efficiency, and model quality.

II. Literature Review

A. Review of Relevant Literature

Large language models (LLMs) have grown dramatically in scale and capability over recent years [1], but this growth has introduced significant challenges related to inference speed and memory efficiency. The autoregressive nature of generation, where each token depends on the previous context, makes inference inherently sequential and costly.

Speculative decoding, introduced by Chen et al. [2], accelerates generation by using a lightweight draft model to propose multiple tokens, which are then verified by a larger, more accurate target model. If all proposed tokens match the target's output, they are accepted at once, effectively collapsing multiple steps into one. In the event of a mismatch, the sequence is rolled back to the first incorrect token, and generation resumes from there. This allows for significant improvements in throughput while preserving output quality, as the final output distribution is still governed by the target model.

Recent works have refined the speculative decoding pipeline to make the acceptance process more efficient. Mamou et al. [3] introduced dynamic speculation lookahead, using a classification head to determine the optimal number of tokens to propose based on draft confidence. Other extensions, such as candidate-guided decoding and constrained speculation, continue to explore how we can maximize the benefits of speculative execution while reducing rollbacks. For example, multi-candidate speculative decoding [4], [5] proposes sampling multiple tokens from a draft model in parallel and verifying them in batches, significantly improving acceptance rates and throughput.

Quantization has emerged as a complementary technique to reduce memory footprint and improve runtime efficiency. Instead of using full 16-bit or 32-bit precision, quantized models operate at 8-bit or even 4-bit precision. This significantly reduces memory bandwidth usage and allows larger models to run on constrained hardware. The BitsAndBytes library [6] popularized 4-bit quantization in the transformer ecosystem, supporting formats like NF4 for training and inference.

To evaluate the quality of generation under these constraints, we rely on both language modeling and task-based metrics. HumanEval [7], a dataset of Python programming problems introduced by OpenAI, has become a standard for assessing code generation correctness. It measures functional accuracy by executing model-generated code against hidden test cases. For language modeling tasks, perplexity remains a standard metric for assessing how well the model predicts natural text.

III. Methodology

A. Data Collection and Preprocessing

For our experiments, we used the wikitext-2-raw-v1 [8] dataset from HuggingFace Datasets. This version of WikiText-2 contains the raw, unprocessed text from verified Wikipedia articles and is commonly used for language modeling tasks that require finer control over tokenization and preprocessing. The raw test set contains approximately 4,000 samples. To ensure consistency across all speculative decoding runs, we filtered the test set to include only samples with at least 128 tokens (as determined after tokenization with our model's tokenizer). If a sample exceeded 128 tokens, we truncated it to exactly 128 tokens. This preprocessing step allowed us to:

- Ensure that all prompts were exactly 128 tokens, maintaining uniform input lengths.
- Start generation from a consistent point (token 129) for every sample.
- Remove any variation in performance that could be caused by differing prompt lengths, which could affect both latency and the number of speculative decoding steps.

We chose wikitext-2-raw-v1 specifically because it is a known LLM benchmarking dataset and it gives us more control over tokenization, aligning better with the tokenizer used by LLaMA 3 [9] and ensuring a realistic, open-ended generation setup. Its Wikipedia-based content provides high-quality natural language while being lightweight enough for efficient experimentation.

B. Model Selection

We chose to use LLaMA 3 for our experiments due to its strong performance, open availability, and the existence of multiple model sizes within the same architecture family. LLaMA 3 is among the state-of-the-art open-weight LLMs as of 2024, with competitive accuracy across common language benchmarks and improved training stability and efficiency compared to LLaMA 2. Its availability in 1B, 3B, 8B, and 70B variants makes it particularly well-suited for speculative decoding, which requires both a large target model and smaller draft models that are architecturally compatible. We selected the 8B model as our target model, primarily due to hardware constraints — larger models like the 70B version would not fit within the memory and compute limits of our L4 GPU setup. Despite this limitation, the 8B model remains a compelling target: it's large enough for generation to be computationally expensive, creating room for speculative decoding to yield meaningful speedups. Critically, LLaMA 3's 1B and 3B variants make for ideal draft models. To benefit from speculative decoding, the draft model should be significantly smaller than the target model so that the cost of generating multiple speculative tokens is much lower than computing them directly on the target model. The architectural similarity across LLaMA 3 variants (tokenization, attention implementation, etc.) also ensures that speculative decoding remains stable and performant.

C. Optimization Procedure(s)

Our focus was exclusively on optimizing inference rather than training. We aimed to reduce latency and memory usage by tuning speculative decoding parameters—specifically, the confidence threshold, number of assistant tokens, and sampling strategy. Additionally, we applied post-training quantization (8-bit and 4-bit) using the BitsAndBytes library to compress model weights and further improve efficiency. All configurations were selected to balance speedup, accuracy, and GPU utilization on constrained hardware.

D. Profiling Tools and Methods

To profile our experiments, we use Weights and Biases [10] and NVIDIA Nsight Systems [11].

Weights and Biases (W&B) is used for logging high-level training and inference metrics, such as loss, throughput (tokens/sec), latency per token, memory usage, and acceptance rates. It allows us to track speculative decoding performance across multiple configurations in real time and compare results interactively.

NVIDIA Nsight Systems provides fine-grained, system-level GPU profiling capabilities. It collects kernel-level traces and memory transfer information, helping us identify performance bottlenecks and inefficiencies during inference. Specifically, Nsight Systems allowed us to:

- Track CUDA kernel execution times and overlap with memory operations.
- Visualize compute utilization and GPU occupancy across different speculative decoding configurations.
- Measure Host-to-Device and Device-to-Host memory transfer times.
- Identify underutilized GPU cycles caused by frequent rollbacks or model imbalance.

Using Nsight Systems, we observed that certain draft model combinations led to suboptimal GPU utilization. For instance, when using a quantized 4-bit draft model, the compute kernels became shorter but memory-bound, leading to high memory transfer overheads and low occupancy. Similarly, larger draft models like LLaMA3-3B showed higher compute latency during speculative steps, but their benefit in acceptance rate was often nullified due to longer verification delays and kernel queuing.

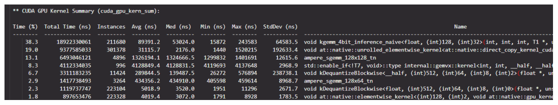
These insights guided our decisions to choose draft models that are not only architecturally compatible but also well-aligned with the GPU's compute/memory balance. In particular, we used Nsight to validate that increasing draft model size beyond a certain point (e.g., from 1B to 3B) gave diminishing returns in speedup due to latency bottlenecks in the verification step.

Overall, Nsight Systems served as a crucial tool in diagnosing performance trade-offs that were not apparent from high-level metrics alone.

E. Evaluation Metrics

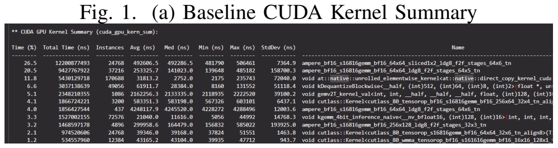
Speculative Decoding Metrics:

For spec decoding the key metrics that need to be tracked are latency/throughput, total number of rollbacks, and acceptance rate. Latency/throughput will give metrics for overall speed performance allowing us to compare it to the baseline throughput to calculate how much speculative decoding has sped up



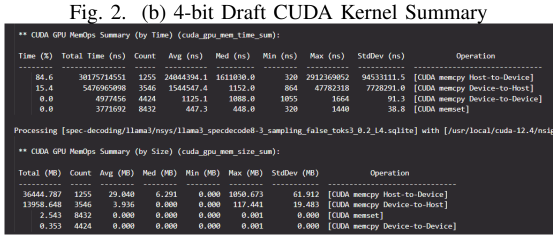
```
** CUDA GPU Kernel Summary (cuda_gpu_kern_sum):

Time (%)  Total Time (ns)  Instances  Avg (ns)   Med (ns)   Min (ns)  Max (ns)  StdDev (ns)                                                   Name
--------  ---------------  ---------  ---------  ---------  --------  --------  -----------  ----------------------------------------------------------------------------------------------------
    38.3      18922330061     211680    89391.2    53024.0     15872    243583      64583.5  void kgemm_4bit_inference_naive<float, (int)128, (int)32>(int, int, int, T1 *, u
    19.0       9377585033     301378    31115.7     2176.0      1440   1520215     192633.4  void at::native::unrolled_elementwise_kernel<at::native::direct_copy_kernel_cuda
    13.1       6493046121       4896  1326194.1  1324666.5   1299832   1401691      12615.6  ampere_sgemm_128x128_tn
     8.3       4112334035        996  4128849.4  4128831.5   4119693   4137648       2968.9  std::enable_if<!T7, void>::type internal::gemvx::kernel<int, int, __half, __hal
     6.7       3311183235      11424   289844.5   139487.5     26272    576894     238738.1  void kDequantizeBlockwise<__half, (int)512, (int)64, (int)8, (int)2>(float *, un
     2.9       1417738493       3264   434356.2   434910.0    405598    459614       8968.7  ampere_sgemm_128x64_tn
     2.3       1119737747     223104     5018.9     3520.0      1951     11296       2671.7  void kDequantizeBlockwise<float, (int)512, (int)64, (int)8, (int)0>(float *, uns
     1.8        897653476     223328     4019.4     3072.0      1791      8928       1783.5  void at::native::elementwise_kernel<(int)128, (int)2, void at::native::gpu_kerne
```

Fig. 1. (a) Baseline CUDA Kernel Summary



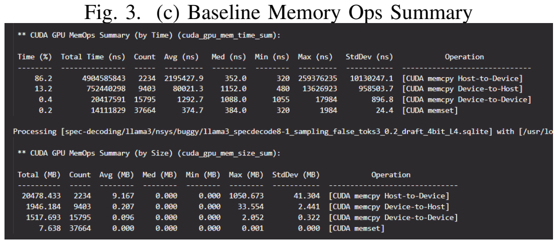
```
** CUDA GPU Kernel Summary (cuda_gpu_kern_sum):

Time (%)  Total Time (ns)  Instances  Avg (ns)   Med (ns)   Min (ns)  Max (ns)  StdDev (ns)                                                   Name
--------  ---------------  ---------  ---------  ---------  --------  --------  -----------  ----------------------------------------------------------------------------------------------------
    26.5      12200877493      24768   492606.5   492286.5    481790    506461       7364.9  ampere_bf16_s16816gemm_bf16_64x64_sliced1x2_ldg8_f2f_stages_64x6_tn
    20.5       9427767922      37216   253325.7   141023.0    139648    485182     158700.3  ampere_bf16_s16816gemm_bf16_64x64_ldg8_f2f_stages_64x5_tn
    11.8       5430129718     170688    31813.2     2752.0      2175    235743      72040.0  void at::native::unrolled_elementwise_kernel<at::native::direct_copy_kernel_cuda
     6.6       3037138639      49056    61911.7    28384.0      8160    131552      51118.4  void kDequantizeBlockwise<__half, (int)512, (int)64, (int)8, (int)2>(float *, uns
     5.1       2348210355       1086  2162256.3  2133335.0   2118935   2222520      39100.2  void gemv2T_kernel_val<int, int, __half, __half, __half, float, (int)128, (int)1
     4.1       1866724221       3200   583351.3   583198.0    567326    603101       6437.1  void cutlass::Kernel<cutlass_80_tensorop_bf16_s16816gemm_bf16_256x64_32x4_tn_ali
     4.0       1856427544        437  4248117.9  4245520.0   4228272   4288496      12003.6  ampere_bf16_s16816gemm_bf16_64x64_ldg8_f2f_stages_64x6_tn
     3.3       1527002155      72576    21040.0    11616.0      5056     44992      14768.3  void kgemm_4bit_inference_naive<__nv_bfloat16, (int)128, (int)16>(int, int, int,
     3.2       1468597178       4896   299958.6   164479.0    156832    585022     193925.0  ampere_bf16_s16816gemm_bf16_256x128_ldg8_f2f_stages_32x3_tn
     2.1        974520606      24768    39346.0    39168.0     37824     51551       1463.8  void cutlass::Kernel<cutlass_80_tensorop_s16816gemm_bf16_64x64_32x6_tn_align8>(T
     1.2        534557960      12384    43165.2    43104.0     39935     47712        943.7  void cutlass::Kernel<cutlass_80_wmma_tensorop_bf16_s161616gemm_bf16_16x16_128x1_
```

Fig. 2. (b) 4-bit Draft CUDA Kernel Summary



```
** CUDA GPU MemOps Summary (by Time) (cuda_gpu_mem_time_sum):

Time (%)  Total Time (ns)  Count  Avg (ns)    Med (ns)   Min (ns)  Max (ns)    StdDev (ns)            Operation
--------  ---------------  -----  ----------  ---------  --------  ----------  -----------  ------------------------------
    84.6      30175714551   1255  24044394.1  1611030.0       320  2912369052   94533111.5  [CUDA memcpy Host-to-Device]
    15.4       5476965098   3546   1544547.4     1152.0       864    47782318    7728291.0  [CUDA memcpy Device-to-Host]
     0.0          4977456   4424      1125.1     1088.0      1055        1664         91.3  [CUDA memcpy Device-to-Device]
     0.0          3771692   8432       447.3      448.0       320        1440         38.8  [CUDA memset]

Processing [spec-decoding/llama3/nsys/llama3_specdecode8-3_sampling_false_toks3_0.2_L4.sqlite] with [/usr/local/cuda-12.4/nsig

** CUDA GPU MemOps Summary (by Size) (cuda_gpu_mem_size_sum):

Total (MB)  Count  Avg (MB)  Med (MB)  Min (MB)  Max (MB)  StdDev (MB)            Operation
----------  -----  --------  --------  --------  --------  -----------  ------------------------------
 36444.787   1255    29.040     6.291     0.000  1050.673       61.912  [CUDA memcpy Host-to-Device]
 13958.648   3546     3.936     0.000     0.000   117.441       19.483  [CUDA memcpy Device-to-Host]
     2.543   8432     0.000     0.000     0.000     0.001        0.000  [CUDA memset]
     0.353   4424     0.000     0.000     0.000     0.001        0.000  [CUDA memcpy Device-to-Device]
```

Fig. 3. (c) Baseline Memory Ops Summary



```
** CUDA GPU MemOps Summary (by Time) (cuda_gpu_mem_time_sum):

Time (%)  Total Time (ns)  Count  Avg (ns)   Med (ns)  Min (ns)  Max (ns)   StdDev (ns)            Operation
--------  ---------------  -----  ---------  --------  --------  ---------  -----------  ------------------------------
    86.2       4904585843   2234  2195427.9     352.0       320  259376235   10130247.1  [CUDA memcpy Host-to-Device]
    13.2        752440298   9403    80021.3    1152.0       480   13626923     958503.7  [CUDA memcpy Device-to-Host]
     0.4         20417591  15795     1292.7    1088.0      1055      17984        896.8  [CUDA memcpy Device-to-Device]
     0.2         14111829  37664      374.7     384.0       320       1984         24.4  [CUDA memset]

Processing [spec-decoding/llama3/nsys/buggy/llama3_specdecode8-1_sampling_false_toks3_0.2_draft_4bit_L4.sqlite] with [/usr/lo

** CUDA GPU MemOps Summary (by Size) (cuda_gpu_mem_size_sum):

Total (MB)  Count  Avg (MB)  Med (MB)  Min (MB)  Max (MB)  StdDev (MB)            Operation
----------  -----  --------  --------  --------  --------  -----------  ------------------------------
 20478.433   2234     9.167     0.000     0.000  1050.673       41.304  [CUDA memcpy Host-to-Device]
  1946.184   9403     0.207     0.000     0.000    33.554        2.441  [CUDA memcpy Device-to-Host]
  1517.693  15795     0.096     0.000     0.000     2.052        0.322  [CUDA memcpy Device-to-Device]
     7.638  37664     0.000     0.000     0.000     0.001        0.000  [CUDA memset]
```

Fig. 4. (d) 4-bit Draft Memory Ops Summary

Fig. 5. Nsight Systems profiling comparisons between full-precision and 4-bit quantized speculative decoding runs.

inference. Total number of rollbacks gives an idea of how often the target model rejects the proposed tokens which is heavily tied to the acceptance rate which gives us information about how accurate the proposed tokens from the draft model are. In our project we tested two different draft models, Llama3-1B and LLama3-3B, so having metrics like total rollbacks and acceptance rate are useful to see how varying the draft model affects what percent of tokens proposed are being accepted by the target. When quantizing the model these metrics are also important as it shows us how much output quality we are sacrificing for potential speed up with quantization.

Pure Quantization Metrics:

We evaluated the impact of quantization using the following key metrics, each chosen to measure a different aspect of model performance:

- HumanEval Accuracy: To assess functional correctness in code generation, we used the HumanEval benchmark—a standard suite of coding tasks developed by OpenAI. We measured pass@1 accuracy across all 164 tasks, which captures the percentage of problems where the model's first generated solution is correct. This metric was chosen to measure whether quantization affects the

model's ability to generate valid, working code without requiring retries.
- Perplexity (WikiText-2): We used perplexity on the WikiText-2 dataset to evaluate how confidently the model predicts the true next token. This allowed us to quantify degradation in language modeling quality due to quantization, helping us understand whether compressed models maintain the ability to model natural language well.
- Throughput and Latency: We tracked token generation throughput (tokens/second) and per-token latency to measure inference speed. This metric was included to evaluate whether quantized models deliver the expected performance gains.
- GPU Utilization and Memory Efficiency: We monitored GPU utilization during generation tasks to assess hardware efficiency. These metrics helped us identify whether quantization led to better GPU resource usage and whether low utilization could explain unexpected latency or speed bottlenecks.

IV. Experimental Results

A. Experimental Setup

Hardware Setup

All experiments were conducted on a Google Cloud VM equipped with:

- GPU: 1 × NVIDIA L4 (24 GB VRAM)
- CPU: 16 vCPUs (Intel Cascade Lake platform)
- Memory: 64 GB RAM
- Disk: 400 GB NVMe balanced persistent disk
- OS Image: Debian 11 (Deep Learning VM image with CUDA 12.4)

Due to our inability to get consistent access to H100 GPUs, we decided to run all our experiments on an L4 GPU in VM instances with the above settings.

Speculative Decoding Experiments:

In order to see if speculative decoding improved for our target model (Llama3-8B), we ran experiments using two different draft models: Llama3-1B and Llama3-3B.

For each speculative decoding experiment, we recorded the following metrics: throughput, latency, rollback count, accept rate, speedup, and average GPU Utilization (%). Explained above in the evaluation metrics section.

For the experiments involving Llama3-1B, the parameters that were constant throughout all the experiments were the maximum prompt length and the number of tokens to be generated. The maximum prompt length we set to 128 which ensured that all inputs were exactly 128 tokens for consistency. The number of tokens to be generated was set to be 128 to simulate realistic text continuation lengths and this value ensured that speedups from speculative decoding would have room to show over multiple iterations. We also did our experiments on 500 samples from the preprocessed test set. The parameters that were varied in our experiments were the confidence threshold, number of assistant tokens, and whether sampling was true or false.

The confidence threshold checks whether the draft model is confident enough in its most recent token prediction. It gets the softmax probability of the most recently generated token and if that probability is less than the confidence threshold, the draft model stops generating more tokens in the current speculative decoding step. Otherwise, it will keep going up to the max assistant tokens that we set. We chose 2 different confidence threshold values to try: 0.0001 and 0.2. We chose 0.0001 as one of our confidence thresholds to test as it simulates standard speculative decoding where each speculative decoding step generates the full number of assistant tokens. With some brief hyperparameter tuning, we found that on tests with a small number of samples, 0.2 seemed to perform the best and chose that as our "optimal" confidence threshold to test on.

When we set the threshold to 0.0001, since the threshold is very low the draft model will almost always generate the full number of assistant tokens. However, this might lead to more rollbacks since we will almost always generate the full number of tokens without checking whether or not the draft model is confident enough in its most recent token prediction.

When we set the threshold to 0.2, the draft model stops proposing tokens sooner because the threshold is now significantly higher meaning that not the full amount of tokens is being generated during each speculative decoding step. This should reduce rollbacks since only tokens above this threshold are proposed but it may limit speedup since fewer tokens are verified by the target model per forward pass.

It was important to vary the number of assistant tokens as that represents the number of tokens the draft model proposes to the target model in each iteration of speculative decoding. Following the speculative decoding paper, we decided to test on 3, 5, and 7 for our number of assistant tokens tracking the key evaluation metrics for each.

When sampling is False, the draft model uses greedy decoding, it selects the most probable token at each step. The target model then simply checks whether it would have generated the same tokens. If so, the tokens are accepted; if not, the process rolls back to the last accepted token. This is a faster and simpler verification method because it only requires comparing token IDs without additional sampling steps.

When sampling is True, the draft model samples from its probability distribution at each step. The target model then performs speculative sampling, following the algorithm described in the speculative decoding paper. Specifically, it:

- Calculates the acceptance probability for each proposed token based on the ratio between the target model's and draft model's probabilities.
- Accepts or rejects tokens probabilistically, proceeding until the first rejection.
- Samples a new token from a corrected distribution based on the difference between the target and draft model logits, ensuring the final output reflects the target model's distribution.

This process allows for more diverse generation when sampling is enabled, but it introduces additional verification steps that can affect speed and acceptance rates.

For speculative decoding with the 3B model as the draft model the experimental setup is the exact same for 1B except for the obvious change of draft model.

For our speculative decoding with quantization experiments, we applied weight-only quantization using the `BitsAndBytesConfig` module from the `transformers` library. For 8-bit quantization, we enabled `load_in_8bit=True`, which compresses model weights to 8-bit integers while keeping activations in full precision. For 4-bit quantization, we used `load_in_4bit=True` with the `nf4` quantization scheme, which is optimized for preserving dynamic range. We set the compute dtype to `bfloat16` for matrix multiplications and enabled `bnb_4bit_use_double_quant=True` to reduce memory usage further via two-stage quantization (8-bit followed by 4-bit). This setup allowed us to evaluate performance trade-offs under aggressive model compression.

We also tried to combine speculative decoding with quantization to explore potential further speed ups through quantization. By quantizing the draft model, it reduces the latency of draft token generation which is performed frequently. In each speculative decoding step, the draft model is autoregressive meaning that it will need to do numerous forward passes to generate each token to be proposed to the target model. Reducing the latency for this draft token generation should yield significant speedup as long as the reduced model output quality as a result of the lowered precision doesn't increase the number of rollbacks to the point of outweighing the speedup gained from lower precision computation. By quantizing the target model, it reduces the cost of verification of the proposed tokens which is also a potential area for speedup as in each speculative decoding step the proposed tokens from the draft necessitate a full forward pass from the target model. For both choices of draft model, we did 5 experiments each to see if introducing quantization yielded overall speedup:

- 8 bit quantized draft model
- 4 bit quantized draft model
- 8 bit quantized target model
- 8 bit quantized target with 8 bit quantized draft
- 8 bit quantized target with 4 bit quantized draft

For each of these experiments, we took the best parameter settings from our speculative decoding experiments without quantization which had no sampling, assistant tokens equal to 3 and confidence threshold of 0.2.

Pure Quantization Experiments:

In this setup, 8-bit quantization was enabled using `load_in_8bit=True`, which stores model weights in 8-bit precision to reduce memory usage. For 4-bit quantization, we used `load_in_4bit=True` with the `nf4` quantization type and `float16` as the compute data type.

To analyze the trade-off between quantization, speed, and accuracy, we ran a series of controlled experiments using the HumanEval benchmark.

We tested LLaMA 3 8B models under different quantization settings: 16-bit (FP16), 8-bit, and 4-bit, using both sampling-based and greedy (non-sampled) generation strategies. In the sampled setup, we used temperature=0.7 and top_p=0.95 to introduce controlled randomness and simulate more diverse output generation.

For each configuration, we computed pass@1 accuracy across all 164 of HumanEval tasks), measuring the percentage of completions where the first generated output passed all unit tests. This allowed us to quantify the functional correctness degradation introduced by quantization.

To evaluate how quantization affects both language modeling quality and inference efficiency, we conducted experiments on the WikiText-2 dataset, measuring both perplexity and generation throughput/latency.

For the generation benchmark, we used max_prompt_tokens=128 and max_new_tokens=128, and ran the model on 200 test samples with greedy decoding. This setup allowed us to accurately capture how quantization impacts token generation speed and latency per token under realistic prompt sizes.

For the perplexity evaluation, we used a sliding window setup with stride=512 and max_length=2048 to compute negative log-likelihood across the full test set. This approach ensures that long sequences are processed in overlapping chunks, giving a stable and reliable measure of the model's confidence in predicting the true next token across a natural language corpus.By comparing perplexity, latency, and throughput across different quantization levels (16-bit, 8-bit, and 4-bit), we were able to assess the trade-offs between model quality, efficiency, and hardware utilization. GPU utilization was also tracked throughout to better understand the relationship between quantization and performance bottlenecks.

B. *Performance Comparison (Before and After Optimizations)*

TABLE I
LLAMA-3 8B/1B – SPECULATIVE DECODING ($\tau = 0.0001$, NO SAMPLING)

Tok	Thr. (t/s)	Lat. (ms/t)	RB	Acc.	Speedup	GPU %
3	19.995	50.013	2502	0.547	1.301	89.08
5	18.653	53.611	2700	0.440	1.214	86.79
7	16.696	59.896	2796	0.363	1.086	72.30

TABLE II
LLAMA-3 8B/1B – SPECULATIVE DECODING ($\tau = 0.0001$, WITH SAMPLING)

Tok	Thr.	Lat.	RB	Acc.	Speedup	GPU %
3	19.833	50.420	2491	0.549	1.290	89.88
5	18.226	54.866	2785	0.431	1.186	76.08
7	16.510	60.570	2863	0.361	1.074	73.84

TABLE III
LLAMA-3 8B/1B – SPECULATIVE DECODING ($\tau = 0.2$, NO SAMPLING)

Tok	Thr.	Lat.	RB	Acc.	Speedup	GPU %
3	20.769	48.150	2428	0.609	1.351	81.27
5	20.535	48.697	2606	0.544	1.336	76.91
7	20.334	49.178	2653	0.500	1.323	77.06

TABLE IV
LLAMA-3 8B/1B – SPECULATIVE DECODING ($\tau = 0.2$, WITH SAMPLING)

Tok	Thr.	Lat.	RB	Acc.	Speedup	GPU %
3	20.425	48.960	2353	0.605	1.329	79.97
5	19.385	51.585	2550	0.490	1.261	76.61
7	19.152	52.215	2657	0.480	1.246	75.21

TABLE V
LLAMA-3 8B/1B – QUANTIZED DRAFT/TARGET (TOK = 3, $\tau = 0.2$)

Config	Thr.	Lat.	RB	Acc.	Speedup	GPU %
8-bit Draft	12.818	78.018	2491	0.602	0.834	50.26
4-bit Draft	14.423	69.335	2883	0.539	0.938	67.08
8-bit Target	14.661	68.207	2491	0.601	0.954	46.25
8-bit T, 4-bit D	11.025	90.702	2900	0.537	0.717	41.54

TABLE VI
LLAMA-3 8B/3B – SPECULATIVE DECODING ($\tau = 0.2$, NO SAMPLING)

Tok	Thr.	Lat.	RB	Acc.	Speedup	GPU %
3	17.099	58.481	1967	0.673	1.112	88.14
5	16.580	60.313	2116	0.610	1.079	92.56
7	15.928	62.782	2194	0.559	1.036	92.61

TABLE VII
LLAMA-3 8B/3B – SPECULATIVE DECODING ($\tau = 0.2$, WITH SAMPLING)

Tok	Thr.	Lat.	RB	Acc.	Speedup	GPU %
3	16.794	59.546	1985	0.660	1.093	90.80
5	16.324	61.261	2055	0.605	1.062	91.41
7	15.475	64.621	2170	0.556	1.010	90.45

TABLE VIII
LLAMA-3 8B/3B – QUANTIZED DRAFT/TARGET (TOK = 3, $\tau = 0.2$)

Config	Thr.	Lat.	RB	Acc.	Speedup	GPU %
8-bit Draft	9.358	106.863	2047	0.663	0.609	42.91
4-bit Draft	10.627	94.103	2264	0.637	0.691	74.52
8-bit Target	12.804	78.101	2006	0.661	0.833	60.49
8-bit T, 4-bit D	8.607	116.188	2250	0.634	0.560	55.15

TABLE IX
HUMAN-EVAL ACCURACY – SAMPLED DECODING (LLAMA-3 8B)

Quant.	Success Rate
4-bit	0.317
8-bit	0.329
16-bit	0.329

TABLE X
HUMAN-EVAL ACCURACY – NON-SAMPLED (LLAMA-3 8B)

Quant.	Success Rate
4-bit	0.292
8-bit	0.323
16-bit	0.336

TABLE XI
WIKITEXT-2 GENERATION – THROUGHPUT, PERPLEXITY, UTILIZATION

Quant.	Thr.	Lat.	PPL	GPU %
16-bit	15.499	64.519	5.563	97.97
8-bit	9.028	110.765	5.640	48.93
4-bit	21.036	47.538	6.242	70.37

C. Analysis of Results

Analysis of Speculative Decoding without Quantization: We evaluated speculative decoding across two draft models: LLaMA3-1B and LLaMA3-3B, while varying confidence threshold, assistant tokens, and sampling mode. The performance trends were consistent across both model configurations.

1. Confidence Threshold

Using a threshold of τ = 0.0001, the draft model almost always generated the full number of assistant tokens. However, this led to lower acceptance rates and more rollbacks. For example, in the 1B greedy decoding setup, the acceptance rate decreased from 0.609 at τ = 0.2 (Table III) to 0.547 at τ = 0.0001 (Table I), and rollbacks increased from 2428 to 2502.
This demonstrates that a low threshold results in longer proposals but lower-quality tokens near the end of the sequence, increasing the likelihood of rejection and wasted verification. Conversely, a higher threshold (τ = 0.2) causes the draft model to stop earlier and produce shorter, more confident sequences, which improves acceptance and reduces overhead. This led to improved speedup — from 1.301× (τ = 0.0001) to 1.351× (τ = 0.2) at k=3 (Tables I and III).
my own: One of the parameters we varied as stated in the experimental setup was the confidence threshold. With a confidence threshold of 0.0001, the draft model nearly always generates the full assistant token count, since very few tokens fall below such a low probability cutoff. In contrast, a threshold of 0.2 causes the draft model to stop early when it generates tokens with low confidence, producing shorter but higher-quality speculative proposals. The reason why we saw a speedup when setting the confidence threshold to 0.2 is because if the draft model produced a token with softmax probability below that threshold, it was likely to be rejected by the target model anyways. As a result, instead of continuing to generate further tokens which will anyways be discarded after the first rejected token during the target model's verification, the proposed sequence is shorter and of higher confidence which means it will have a higher acceptance rate which is

corroborated by the data. By doing this pre-filtering on the proposed sequence, it overall minimizes wasted compute and allows for speedup compared to the 0.0001 naive threshold.

2. Assistant Tokens

As we increased the number of assistant tokens (k) from 3 to 5 to 7, we observed a consistent drop in acceptance rates and speedup. For example, in Table I (τ = 0.0001, no sampling), increasing from k=3 to k=7 caused:

- Acceptance rate drop: 0.547 $\rightarrow$ 0.363
- Rollbacks increase: 2502 $\rightarrow$ 2796
- Speedup drop: 1.301× $\rightarrow$ 1.086×

This is expected since as overall proposal length increases, there is an increased chance of rejection leading to more total rollbacks. Since each rollback requires an entirely new speculative decoding step which consists of draft token generation and a full forward pass for verifying those proposed tokens, each rollback increases overhead and more rollbacks means less overall speedup. However, this degradation is more moderate when τ = 0.2, where speedup only drops from 1.351× to 1.323× as k increases from 3 to 7 (Table III) because fewer assistant tokens are actually generated in each speculative step due to the confidence threshold reducing the frequency of long (and potentially low-confidence) proposals, which helps mitigate rollback impact.

3. Sampling vs. Greedy

Across all experiments, greedy decoding (no sampling) consistently outperformed speculative sampling. For example, with 1B at k=3, τ = 0.2:

- Greedy speedup: 1.351× (Table III)
- Sampling speedup: 1.329× (Table IV)

Sampling adds stochasticity to both the generation of the draft tokens as well as the speculative sampling process. This added randomness in the draft generation means that the proposed tokens are less likely to be accepted by the draft model than in the deterministic scheme given by the greedy method. Speculative sampling also introduces randomness into the verification process, meaning the draft and target models are less likely to align. Overall, this leads to an increase in rollbacks and therefore leads to worse speedup.

4. 1B vs. 3B Draft Model

Although both draft sizes showed similar trends, the 1B model consistently outperformed 3B in terms of speedup. At τ = 0.2, k=3, no sampling:

- 1B speedup: 1.351× (Table III)
- 3B speedup: 1.112× (Table VI)

Comparing the acceptance rates on the runs, the 3B results had a 67% acceptance rate compared to the lower 61% for the 1B. This is expected since the 3B model has triple the parameters and therefore should have better overall model output quality. However, this increased model size also comes with a downside of being more computationally heavy meaning that each forward pass takes significantly longer than in the 1B model. As a result, although the 3B model is producing higher quality proposal sequences that get accepted more often, the extra time taken to generate these tokens led to overall speedup being much worse. This reinforces the importance of selecting a draft model that is significantly cheaper than the target to realize the full benefits of speculative decoding.

Analysis of Speculative Decoding with Quantization: Combining speculative decoding with quantization yielded limited performance benefits and, in fact, led to worse speedup in our experiments. Specifically, quantizing the draft model—particularly to 4-bit precision—resulted in a significant degradation in output quality. This was evident from the increase in the number of rollbacks, rising from 2428 in the non-quantized 1B setup (Table **??**) to 2883 in the 4-bit quantized run (Table **??**). While quantization theoretically reduces memory usage and can accelerate inference by enabling faster matrix operations, these benefits are offset when the draft model's token proposals become unreliable. A higher number of rejected tokens necessitates more frequent verification by the target model, negating any latency improvements gained from quantization. From our Nsight Systems profiling, we observed that quantized setups incurred additional kernel overheads, including costly dequantization routines and fragmented compute execution. The number of `cudaLaunchKernel` calls increased markedly, and several element-wise operations—such as `kDequantizeBlockwise` and `kgemm_4bit_inference_naive`—appeared frequently in the 4-bit kernel traces, consuming a significant fraction of total execution time. This aligns with our expectations: low-bit quantized inference often introduces auxiliary operations that offset theoretical speedups, especially when executed on hardware without native INT4 acceleration. GPU utilization metrics further support this inefficiency. In the full-precision speculative decoding runs, utilization remained high—peaking at around 89–90% with 3 assistant tokens, and maintaining 77–81% even at higher thresholds (e.g., $\tau = 0.2$) due to shorter speculative bursts. However, when using quantized draft models, utilization dropped sharply: 8-bit setups averaged around 50%, and the mixed 8-bit target with 4-bit draft configuration fell as low as 41%. This suggests that the performance bottleneck shifted from compute-bound operations to memory-bound and synchronization overheads introduced by quantization. Although our results do not show net performance gains when combining quantization with speculative decoding on L4 hardware, we believe the approach remains promising on modern GPUs. Future experiments using larger model pairings—such as a `LLaMA3-70B` target and `LLaMA3-8B` draft—may yield higher-quality draft proposals. If the draft model performs well in full precision, quantizing it to 4- or 8-bit could offer considerable memory savings without significantly increasing rollback rates. This setup could better exploit the throughput and memory bandwidth advantages of newer architectures while preserving speculative decoding efficiency.

Analysis of Pure Quantization Results:

Accuracy Trends on HumanEval.
In the sampled setting, 8-bit quantization matched 16-bit in accuracy (pass@1 = 0.329), showing no degradation. In contrast, 4-bit exhibited a modest drop to 0.317, a 3.65% relative decrease. However, in the greedy (non-sampled) setting, the accuracy gap widened: 8-bit dropped to 0.323 (−3.9% from 16-bit), and 4-bit fell more sharply to 0.292 (−13% from 16-bit). Sampling improved performance at lower bit-widths (4-bit, 8-bit), though 16-bit saw a slight performance drop (−2%). This suggests that sampling helps mitigate degradation caused by aggressive quantization, especially at lower bit-widths.

Speed and Latency on WikiText-2.
For non-sampled text generation on 200 prompts, 4-bit quantization yielded the best throughput (21.04 tokens/sec) and lowest latency (47.5 ms/token), a ∼35% speedup over 16-bit (15.5 tokens/sec, 64.5 ms/token). In contrast, 8-bit quantization performed worst, achieving only 9.0 tokens/sec with 110.8 ms/token latency. This underperformance is likely due to suboptimal kernel/hardware optimizations for 8-bit inference on L4 GPUs, leading to inefficient memory access and lower GPU utilization (48.9%). 16-bit inference utilized the GPU effectively (∼98% utilization), while 8-bit showed a significant drop (∼49%), reinforcing the claim that hardware support is critical for mid-precision quantization. The 4-bit model performed better than 8-bit in both throughput and utilization (∼70%).

As expected, perplexity increased with more aggressive quantization. The 16-bit model achieved the lowest perplexity (5.563), while 8-bit showed only a slight degradation (5.640, +1.4%). The 4-bit model showed a larger shift to 6.242 (+12.3%), reflecting reduced confidence in the correct next token.

Overall, 4-bit quantization demonstrated strong performance, with only a modest drop in accuracy on HumanEval and a slight increase in perplexity on WikiText-2. This makes it a compelling option when memory constraints are tight and small reductions in accuracy are acceptable. In contrast, 8-bit quantization preserved accuracy very well, closely matching 16-bit performance. However, its low throughput on L4 GPUs—likely due to limited kernel-level optimization—makes it a less attractive choice for deployment on such hardware, unless throughput is not a concern and memory savings are still a priority. 16-bit, as expected, performed best across the board, but requires significantly more memory and compute resources.

V. Discussion

A. Interpretation of Results

Overall, the results showed a meaningful speedup of LLaMA3-8B using speculative decoding, demonstrating that it is a viable and practical method for optimizing LLM inference. In particular, using a LLaMA3-1B draft model, we achieved up to 1.351× speedup under optimal settings (confidence threshold τ = 0.2, 3 assistant tokens, greedy decoding), as shown in Table III. When comparing the 1B vs. 3B draft models, the larger 3B model produced more accurate tokens (higher acceptance rate), but its higher compute cost per step negated those gains, resulting in lower overall speedup (1.112× vs. 1.351×). This confirms that the draft model must be significantly faster than the target to fully exploit speculative decoding. Finally, quantization introduced additional complexity. While it reduces memory and compute, quantizing the draft model led to lower token quality and a substantial increase in rollbacks. As a result, the overall speedup was worse than the non-quantized version. These results highlight the need for careful tradeoff analysis when combining quantization with speculative decoding. Finally, in our standalone quantization experiments (Tables IX–XI), we observed that quantization alone can reduce latency significantly, especially at 4-bit precision. However, we also noticed that quantization must be carefully tailored to the target hardware and usage context as seen with our 8-bit quantization getting significantly worse results likely due to the fact that the L4 has insufficient native support for INT8.

B. Comparison with Previous Studies

Our findings align closely with prior work on speculative decoding. In particular, we observed a change in model accuracy as we varied the number of assisted tokens, with the best performance achieved at 3 tokens. This is consistent with the original speculative decoding paper [2], which also identifies a trade-off between the number of assisted tokens and inference efficiency.

Our experiments further validate the expected performance degradation when enabling sampling. Greedy decoding consistently resulted in faster inference compared to stochastic sampling, in line with observations made in earlier works.

A significant impact on inference speed was observed when tuning the acceptance threshold for the draft model. Setting the confidence threshold to a very low value (e.g., 0.0001) led to a large number of generated tokens being rejected by the target model, ultimately slowing down generation. By increasing the threshold to 0.2, we achieved a substantial improvement in throughput. This behavior is consistent with the findings of Mamou et al. [3], who employ a classification head to adaptively tune this threshold. In contrast, our method relies on fixing the threshold as a static hyperparameter in an unsupervised manner.

However, one area where our results diverge from previous literature is in the use of quantized models with speculative decoding. While earlier studies suggest quantization can accelerate inference, we observed a slowdown. We hypothesize that quantization reduces the confidence of token predictions from the draft model, thereby increasing the number of rollbacks and lowering the acceptance rate—ultimately hurting performance despite theoretical gains in compute efficiency.

C. Challenges and Limitations

One of the primary challenges we faced was limited access to high-performance compute resources. Unlike prior works that benefited from consistent availability of powerful GPUs such as the A100 or H100, our experiments were conducted entirely on NVIDIA L4 GPUs. This constrained our ability to explore larger-scale configurations or run extensive hyperparameter sweeps.

Another major limitation was the lack of documentation and modularity within the speculative decoding library code. Modifying and extending the codebase to suit our experimental goals required significant effort, as the implementation contained nuanced logic that was not clearly exposed.

We also had to build infrastructure from scratch for profiling and evaluation. This included developing metric tracking suites and custom pipelines to ensure reproducible experiments across different model sizes, token settings, and decoding configurations.

Memory constraints posed persistent obstacles throughout the project. We encountered frequent out-of-memory (OOM) errors, especially when working with larger models, which forced us to iteratively pivot the project direction and downscale experiments to fit within hardware limits.

Although we initially explored multiple model architectures—including LLaMA-2 and Mistral—we ultimately focused on LLaMA-3 due to a lack of meaningful performance differentiation across drafts of similar size within the earlier families. This helped us streamline the benchmark space without sacrificing key experimental insights.

D. Future Directions

A promising direction for future work is to explore larger-scale speculative decoding with quantization using higher-VRAM GPUs such as A100 or H100. With more powerful hardware, we could deploy a significantly larger target model like LLaMA3-70B and pair it with a quantized LLaMA3-8B draft model. Our hypothesis is that the larger draft model would produce higher-quality speculative proposals, reducing rollbacks even under 4-bit quantization. This setup could better balance speed and accuracy than the smaller draft models used in our study. Additionally, future experiments could explore more advanced quantization methods (e.g., group-wise or outlier-aware quantization) and enhanced speculative decoding algorithms, such as dynamic token proposal lengths or rejection-aware speculation. These improvements could yield better trade-offs in both throughput and quality, especially when scaling speculative decoding to industrial-grade LLM deployments.

VI. Conclusion

A. Summary of Findings

This project explored speculative decoding and quantization as complementary strategies for improving the inference performance of large language models. Our experiments confirmed that speculative decoding can offer meaningful speedup, over 35% improvement in optimal configurations, while maintaining output quality. We found that achieving this speedup depends critically on tuning parameters such as the confidence threshold, the number of assistant tokens, and the draft model size. We also evaluated quantization both independently and in combination with speculative decoding. While 4-bit quantization provided the best throughput in standalone settings, its effectiveness diminished when paired with speculative decoding due to increased rollbacks caused by degraded draft quality. Additionally, we observed that hardware limitations, such as limited support for INT8 inference on L4 GPUs, can significantly impact quantized performance. Taken together, our results show that speculative decoding is a viable method for accelerating LLMs, but that careful tuning and hardware-aware quantization strategies are necessary to fully realize its benefits.

B. Contributions

Our team held meetings 1-2x a week and most of the work was done together with a large chunk of time brainstorming and deliberating over major project decisions. Overall, every team member was involved in all aspects of the project but for more specific contributions here are some highlights for each of the team members:

- Ryan: Spec decoding optimization for key evaluation metrics like rollback and accepted token percentage, hyperparameter tuning
- Michael: HumanEval, quantization evaluation, and spec decoding optimization for confidence threshold
- Niranjan: Baselines for spec decoding and quantization, hardware and system configurations across hardwares, profiling

References

[1] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," 2020. [Online]. Available: https://arxiv.org/abs/2001.08361

[2] C. Chen, S. Borgeaud, G. Irving, J.-B. Lespiau, L. Sifre, and J. Jumper, "Accelerating large language model decoding with speculative sampling," 2023. [Online]. Available: https://arxiv.org/abs/2302.01318

[3] J. Mamou, O. Pereg, D. Korat, M. Berchansky, N. Timor, M. Wasserblat, and R. Schwartz, "Dynamic speculation lookahead accelerates speculative decoding of large language models," *arXiv preprint arXiv:2405.04304*, 2024. [Online]. Available: https://arxiv.org/abs/2405.04304

[4] S. Yang, S. Huang, X. Dai, and J. Chen, "Multi-candidate speculative decoding," 2024. [Online]. Available: https://arxiv.org/abs/2401.06706

[5] X. Lu, Y. Zeng, F. Ma, Z. Yu, and M. Levorato, "Improving multi-candidate speculative decoding," 2024. [Online]. Available: https://arxiv.org/abs/2409.10644

[6] HuggingFace, "Bitsandbytes quantization — huggingface transformers," 2024, https://huggingface.co/docs/transformers/en/quantization/bitsandbytes.

[7] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[8] S. Merity, C. Xiong, J. Bradbury, and R. Socher, "Pointer sentinel mixture models," 2016. [Online]. Available: https://arxiv.org/abs/1609.07843

[9] A. Grattafiori, A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Vaughan, A. Yang, A. Fan, A. Goyal, A. Hartshorn, A. Yang, A. Mitra, A. Sravankumar, A. Korenev, A. Hinsvark, A. Rao, A. Zhang, A. Rodriguez, A. Gregerson, A. Spataru, B. Roziere, B. Biron, B. Tang, B. Chern, C. Caucheteux, C. Nayak, C. Bi, C. Marra, C. McConnell, C. Keller, C. Touret, C. Wu, C. Wong, C. C. Ferrer, C. Nikolaidis, D. Allonsius, D. Song, D. Pintz, D. Livshits, D. Wyatt, D. Esiobu, D. Choudhary, D. Mahajan, D. Garcia-Olano, D. Perino, D. Hupkes, E. Lakomkin, E. AlBadawy, E. Lobanova, E. Dinan, E. M. Smith, F. Radenovic, F. Guzmán, F. Zhang, G. Synnaeve, G. Lee, G. L. Anderson, G. Thattai, G. Nail, G. Mialon, G. Pang, G. Cucurell, H. Nguyen, H. Korevaar, H. Xu, H. Touvron, I. Zarov, I. A. Ibarra, I. Kloumann, I. Misra, I. Evtimov, J. Zhang, J. Copet, J. Lee, J. Geffert, J. Vranes, J. Park, J. Mahadeokar, J. Shah, J. van der Linde, J. Billock, J. Hong, J. Lee, J. Fu, J. Chi, J. Huang, J. Liu, J. Wang, J. Yu, J. Bitton, J. Spisak, J. Park, J. Rocca, J. Johnstun, J. Saxe, J. Jia, K. V. Alwala, K. Prasad, K. Upasani, K. Plawiak, K. Li, K. Heafield, K. Stone, K. El-Arini, K. Iyer, K. Malik, K. Chiu, K. Bhalla, K. Lakhotia, L. Rantala-Yeary, L. van der Maaten, L. Chen, L. Tan, L. Jenkins, L. Martin, L. Madaan, L. Malo, L. Blecher, L. Landzaat, L. de Oliveira, M. Muzzi, M. Pasupuleti, M. Singh, M. Paluri, M. Kardas, M. Tsimpoukelli, M. Oldham, M. Rita, M. Pavlova, M. Kambadur, M. Lewis, M. Si, M. K. Singh, M. Hassan, N. Goyal, N. Torabi, N. Bashlykov, N. Bogoychev, N. Chatterji, N. Zhang, O. Duchenne, O. Çelebi, P. Alrassy, P. Zhang, P. Li, P. Vasic, P. Weng, P. Bhargava, P. Dubal, P. Krishnan, P. S. Koura, P. Xu, Q. He, Q. Dong, R. Srinivasan, R. Ganapathy, R. Calderer, R. S. Cabral, R. Stojnic, R. Raileanu, R. Maheswari, R. Girdhar, R. Patel, R. Sauvestre, R. Polidoro, R. Sumbaly, R. Taylor, R. Silva, R. Hou, R. Wang, S. Hosseini, S. Chennabasappa, S. Singh, S. Bell, S. S. Kim, S. Edunov, S. Nie, S. Narang, S. Raparthy, S. Shen, S. Wan, S. Bhosale, S. Zhang, S. Vandenhende, S. Batra, S. Whitman, S. Sootla, S. Collot, S. Gururangan, S. Borodinsky, T. Herman, T. Fowler, T. Sheasha, T. Georgiou, T. Scialom, T. Speckbacher, T. Mihaylov, T. Xiao, U. Karn, V. Goswami, V. Gupta, V. Ramanathan, V. Kerkez, V. Gonguet, V. Do, V. Vogeti, V. Albiero, V. Petrovic, W. Chu, W. Xiong, W. Fu, W. Meers, X. Martinet, X. Wang, X. Wang, X. E. Tan, X. Xia, X. Xie, X. Jia, X. Wang, Y. Goldschlag, Y. Gaur, Y. Babaei, Y. Wen, Y. Song, Y. Zhang, Y. Li, Y. Mao, Z. D. Coudert, Z. Yan, Z. Chen, Z. Papakipos, A. Singh, A. Srivastava, A. Jain, A. Kelsey, A. Shajnfeld, A. Gangidi, A. Victoria, A. Goldstand, A. Menon, A. Sharma, A. Boesenberg, A. Baevski, A. Feinstein, A. Kallet, A. Sangani, A. Teo, A. Yunus, A. Lupu, A. Alvarado, A. Caples, A. Gu, A. Ho, A. Poulton, A. Ryan, A. Ramchandani, A. Dong, A. Franco, A. Goyal, A. Saraf, A. Chowdhury, A. Gabriel, A. Bharambe, A. Eisenman, A. Yazdan, B. James, B. Maurer, B. Leonhardi, B. Huang, B. Loyd, B. D. Paola, B. Paranjape, B. Liu, B. Wu, B. Ni, B. Hancock, B. Wasti, B. Spence, B. Stojkovic, B. Gamido, B. Montalvo, C. Parker, C. Burton, C. Mejia, C. Liu, C. Wang, C. Kim, C. Zhou, C. Hu, C.-H. Chu, C. Cai, C. Tindal, C. Feichtenhofer, C. Gao, D. Civin, D. Beaty, D. Kreymer, D. Li, D. Adkins, D. Xu, D. Testuggine, D. David, D. Parikh, D. Liskovich, D. Foss, D. Wang, D. Le, D. Holland, E. Dowling, E. Jamil, E. Montgomery, E. Presani, E. Hahn, E. Wood, E.-T. Le, E. Brinkman, E. Arcaute, E. Dunbar, E. Smothers, F. Sun, F. Kreuk, F. Tian, F. Kokkinos, F. Ozgenel, F. Caggioni, F. Kanayet, F. Seide, G. M. Florez, G. Schwarz, G. Badeer, G. Swee, G. Halpern, G. Herman, G. Sizov, Guangyi, Zhang, G. Lakshminarayanan, H. Inan, H. Shojanazeri, H. Zou, H. Wang, H. Zha, H. Habeeb, H. Rudolph, H. Suk, H. Aspegren, H. Goldman, H. Zhan, I. Damlaj, I. Molybog, I. Tufanov, I. Leontiadis, I.-E. Veliche, I. Gat, J. Weissman, J. Geboski, J. Kohli, J. Lam, J. Asher, J.-B. Gaya, J. Marcus, J. Tang, J. Chan, J. Zhen, J. Reizenstein, J. Teboul, J. Zhong, J. Jin, J. Yang, J. Cummings, J. Carvill, J. Shepard, J. McPhie, J. Torres, J. Ginsburg, J. Wang, K. Wu, K. H. U, K. Saxena, K. Khandelwal, K. Zand, K. Matosich, K. Veeraraghavan, K. Michelena, K. Li, K. Jagadeesh, K. Huang, K. Chawla, K. Huang, L. Chen, L. Garg, L. A, L. Silva, L. Bell, L. Zhang, L. Guo, L. Yu, L. Moshkovich, L. Wehrstedt, M. Khabsa, M. Avalani, M. Bhatt, M. Mankus, M. Hasson, M. Lennie, M. Reso, M. Groshev, M. Naumov, M. Lathi, M. Keneally, M. Liu, M. L. Seltzer, M. Valko, M. Restrepo, M. Patel, M. Vyatskov, M. Samvelyan, M. Clark, M. Macey, M. Wang, M. J. Hermoso, M. Metanat, M. Rastegari, M. Bansal, N. Santhanam, N. Parks, N. White, N. Bawa, N. Singhal, N. Egebo, N. Usunier, N. Mehta, N. P. Laptev, N. Dong, N. Cheng, O. Chernoguz, O. Hart, O. Salpekar, O. Kalinli, P. Kent, P. Parekh, P. Saab, P. Balaji, P. Rittner, P. Bontrager, P. Roux, P. Dollar, P. Zvyagina, P. Ratanchandani, P. Yuvraj, Q. Liang, R. Alao, R. Rodriguez, R. Ayub, R. Murthy, R. Nayani, R. Mitra, R. Parthasarathy, R. Li, R. Hogan, R. Battey, R. Wang, R. Howes, R. Rinott, S. Mehta, S. Siby, S. J. Bondu, S. Datta, S. Chugh, S. Hunt, S. Dhillon, S. Sidorov, S. Pan, S. Mahajan, S. Verma, S. Yamamoto, S. Ramaswamy, S. Lindsay, S. Lindsay, S. Feng, S. Lin, S. C. Zha, S. Patil, S. Shankar, S. Zhang, S. Zhang, S. Wang, S. Agarwal, S. Sajuyigbe, S. Chintala, S. Max, S. Chen, S. Kehoe, S. Satterfield, S. Govindaprasad, S. Gupta, S. Deng, S. Cho, S. Virk, S. Subramanian, S. Choudhury, S. Goldman, T. Remez, T. Glaser, T. Best, T. Koehler, T. Robinson, T. Li, T. Zhang, T. Matthews, T. Chou, T. Shaked, V. Vontimitta, V. Ajayi, V. Montanez, V. Mohan, V. S. Kumar, V. Mangla, V. Ionescu, V. Poenaru, V. T. Mihailescu, V. Ivanov, W. Li, W. Wang, W. Jiang, W. Bouaziz, W. Constable, X. Tang, X. Wu, X. Wang, X. Wu, X. Gao, Y. Kleinman, Y. Chen, Y. Hu, Y. Jia, Y. Qi, Y. Li, Y. Zhang, Y. Zhang, Y. Adi, Y. Nam, Yu, Wang, Y. Zhao, Y. Hao, Y. Qian, Y. Li, Y. He, Z. Rait, Z. DeVito, Z. Rosnbrick, Z. Wen, Z. Yang, Z. Zhao, and Z. Ma, "The llama 3 herd of models," 2024. [Online]. Available: https://arxiv.org/abs/2407.21783

[10] L. Biewald, "Experiment tracking with weights and biases," 2020, software available from wandb.com. [Online]. Available: https://www.wandb.com/

[11] NVIDIA Corporation, "Nvidia nsight systems," 2025, https://developer.nvidia.com/nsight-systems.