Homework-3
11-664/763: Inference Algorithms for Language Modeling
Fall 2025

Your name (Andrew ID): Niket Jain (niketj)

Instructors: Graham Neubig, Amanda Bertsch
Teaching Assistants: Clara Na, Vashisth Tiwari, Xinran Zhao

Due: November 25, 2025

Instructions

Please refer to the collaboration and AI use policy as specified in the course syllabus. Additionally, note that **no off-the-shelf inference servers can be used (e.g. vLLM, sglang, etc)**. Assume NVIDIA GPU hardware with CUDA throughout this assignment.

Shared Tasks

Throughout the semester, you will be working with data from three shared tasks. We host the data for each shared task on Hugging Face; you can access them at **[this link]**. We will generally ask for results on the "dev-test" split, which consists of 100 examples for each task, using the evaluation scripts provided. The remainder of the examples can be used for validation, tuning hyperparameters, or any other experimentation you would like to perform. The final shared task at the end of the semester will be evaluated on a hidden test set.

Algorithmic The task that the language model will tackle is N-best Path Prediction (Top-P Shortest Paths). Given a directed graph $G = (V, E)$ with $|V| = N$ nodes labeled $0, \ldots, N-1$ and non-negative integer edge weights $w : E \to 1, \ldots, W$, the task is to find the top-P distinct simple paths from source $s = 0$ to target $t = N - 1$ minimizing the additive cost

$$c(\pi) = \sum_{(u,v)\in\pi} w(u, v). \tag{1}$$

The output is a pair

$$\texttt{paths} = [\pi_1, \ldots, \pi_P], \quad \texttt{weights} = [c(\pi_1), \ldots, c(\pi_P)], \tag{2}$$

sorted by non-decreasing cost. The language model will be expected to use tool calls[1] to specify its answer.

[1] https://platform.openai.com/docs/guides/function-calling

Evaluation compares predicted pairs $(\pi, c(\pi))$ against the reference set with the score

$$\text{score} = \frac{|(\pi, c(\pi))\text{pred} \cap (\pi, c(\pi))\text{gold}|}{P}. \tag{3}$$

MMLU medicine We will use the two medicine-themed splits of MMLU: college_medicine and professional_medicine. Evaluation is on exact match with the correct multiple-choice answer (e.g. "A").

Infobench Infobench provides open-ended queries with detailed evaluation rubrics. Evaluation **requires calling gpt-5-nano**; we expect that the total cost for evaluation for this homework will be substantially less than $5. See the paper for more information.

1 Optimization for available hardware [25 points]

GPUs and other accelerator hardware have been a significant factor in the (re)surge(nce) of interest in and progress in artificial intelligence and neural machine learning since the early 2010s. Relatively few people who regularly publish at top AI/ML/NLP venues would claim to have a deep understanding of the inner workings of a GPU, however. For the curious, these are two very nicely written blogs that go over the GPU math in detail and discuss the bottlenecks of inference.

- Transformer Inference Arithmetic by Kipply [1]
- Making Deep Learning Go Brrrr From First Principles by Horace He [2]

In general, there are enough well-maintained, open-source, high-level ML frameworks (e.g. PyTorch) and inference engines (e.g. vLLM) that an ML researcher or practitioner can usually get quite far in their career with just a handful of practical rules of thumb. Many of these practical rules of thumb revolve around GPU VRAM[2], and transfers between it and "regular" system RAM. We want to avoid both out-of-memory (OOM) errors and excessive swapping to system RAM.

1.1 Warm-up

During standard autoregressive generation with an LLM, the GPU's VRAM is typically loaded with model weights, KV cache, and a small amount of overhead for model code, intermediate calculations that are not part of the KV cache, and framework overhead.[3] Briefly describe the key differences between LLM generation and training in terms of what is typically loaded into VRAM. What needs to be known and tracked? Is there more or less uncertainty in one? Write 2-3 sentences total.

Solution:
During LLM training, the GPUs VRAM must store model parameters, gradients, and optimizer states (such as momentum or Adam moments), as the model weights are continually updated. In contrast, during inference, VRAM primarily holds the static model weights and the KV cache needed for efficient autoregressive decoding; gradients and optimizer states are not required. □

1.2 Some GPU math questions

1.2.1 Largest model that fits

What is the largest possible model size you could do a single forward pass[4], on a single 80GB GPU? Assume model weights are at half (16-bit floating point) precision, and report to the nearest billion parameters. State your assumptions and show your work.

Solution:

$$\text{Max parameters} = \frac{80 \times 1024^3 \times 8}{16} \approx 42.95 \times 10^9$$

Rounding to the nearest billion, this is 43 billion parameters.
Assumptions: All VRAM is used for weights; no overhead for activations or framework.

[2] the V stands for "video"; a GPU is after all a graphics processing unit
[3] Though machines with larger VRAM size exist, the most common GPUs used with or for LLMs as of 2025 are around 80GB or around half the size (e.g. 48GB for an A6000 or L40; there are also 40GB A100s).
[4] as we would do to evaluate a model-dataset combination for perplexity score – not for auto-regressive generation

□

1.2.2 Largest sequence that fits

What is the largest possible sequence length you could generate to on a single L40S GPU, with a batch size of 8 and `Qwen/Qwen3-14B`? Assume the empty string prompt, and again assume half precision for model weights. Please show your work, state your assumptions, and report your answer to the nearest 10 tokens. Feel free to ignore overhead needed for torch and launch operations (i.e., consider only the VRAM size, model weights, and KV cache).

Solution:
Given: L40S (48 GB), Qwen3-14B, $B = 8$, FP16
Architecture: $L = 40$ layers, $n_{\text{kv}} = 8$ KV heads, $d_{\text{head}} = 640$
Model weights:

$$M_{\text{weights}} = 14 \times 10^9 \times 2 = 28 \text{ GB}$$

Memory per token:

$$M_{\text{per token}} = 2 \times 40 \times 8 \times 640 \times 2 = 26{,}214{,}400 \text{ bytes}$$

Maximum sequence length:

$$S_{\text{max}} = \frac{(48 - 28) \times 1024^3}{26{,}214{,}400} = 819.2 \text{ tokens}$$

Answer: 819 tokens □

1.2.3 Estimating KV cache sizes

The size of the KV cache for a single sequence depends on model configuration, sequence length, and the number of bytes per parameter:

$$\text{KV Cache Size} = 2 \times S \times L \times n_{\text{kv}} \times d_{\text{head}} \times \text{bytes per parameter}$$

where S is sequence length, L is number of layers, n_{kv} is number of KV heads, and d_{model} is head dimension. Note: $d_{\text{model}} = n_{\text{kv}} \times d_{\text{head}}$ (or $n_{\text{q}} \times d_{\text{head}}$ for models with GQA/MQA).

Using the model configs for models in the Qwen3 family, calculate the size of the KV cache needed to generate a sequence length of 32k, for batch sizes of 1, 2, and 4. Assume a static KV cache, half precision model weights, and the empty string prompt.

Solution:

Model size (parameters)	Batch size 1	Batch size 2	Batch size 4
Qwen/Qwen3-0.6B	3.41GB	6.83 GB	13.67 GB
Qwen/Qwen3-1.7B	3.41GB	6.83GB	13.67 GB
Qwen/Qwen3-4B	10.98 GB	21.97 GB	43.94 GB
Qwen/Qwen3-8B	17.57 GB	35.15 GB	70.31 GB
Qwen/Qwen3-14B	24.42 GB	48.82 GB	97.65 GB
Qwen/Qwen3-32B	39.06 GB	78.12 GB	156.25 GB

□

Please also show your work for one of your KV cache size calculations.

Solution:

Given Parameters:

- Sequence length: $S = 32{,}000$
- Model: Qwen3-14B
- Number of layers: $L = 40$
- KV heads (GQA): $n_{\text{kv}} = 8$
- Hidden dimension: $d_{\text{model}} = 5{,}120$
- Head dimension: $d_{\text{head}} = \frac{d_{\text{model}}}{n_{\text{kv}}} = \frac{5{,}120}{8} = 640$
- Precision: FP16 (2 bytes per parameter)
- Batch size: $B = 1$

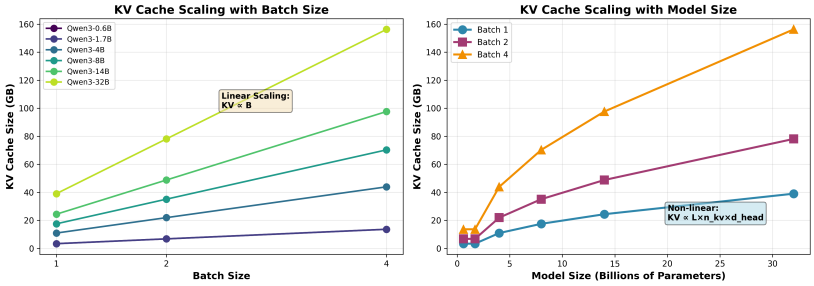
$$\begin{aligned}\text{KV Cache} &= 2 \times S \times L \times n_{\text{kv}} \times d_{\text{head}} \times 2 \times B \\ &= 2 \times 32{,}000 \times 40 \times 8 \times 640 \times 2 \times 1 \\ &= 26{,}214{,}400{,}000 \text{ bytes} \\ &= \frac{26{,}214{,}400{,}000}{1{,}073{,}741{,}824} \text{ GB} \\ &= 24.41 \text{ GB}\end{aligned}$$

The KV cache size is approximately 24.41 GB. □

How does the size of KV scale with batch size? How does it scale with total parameter size in this model family? Feel free to include a figure(s) as your answer.

Solution:

The KV cache scales linearly with increasing batch size. Whereas, with parameter size the KV cache size increases non-linearly because KV cache size depends on attention architecture (number of layers, KV heads, and hidden dimensions) rather than total parameter count. The following diagram describes the trend:



□

1.3 Basic benchmarking

In this question, you are asked to measure wall-clock time and token throughput. That is, input and output sequence lengths are random sequences intentionally constrained to specific values – e.g. `torch.randint(0, vocab_size), [1,64])` for `bs=1, input_len=64`, with enforced minimum = maximum output sequence lengths. Use 1 GPU and keep the type of GPU consistent for all parts of this question. An 80GB GPU is preferred, but 40GB+ is also acceptable if necessary. State which type of GPU hardware you are using.

Solution:
NVIDIA A100 80GB □

1.3.1 Varying input sequence lengths

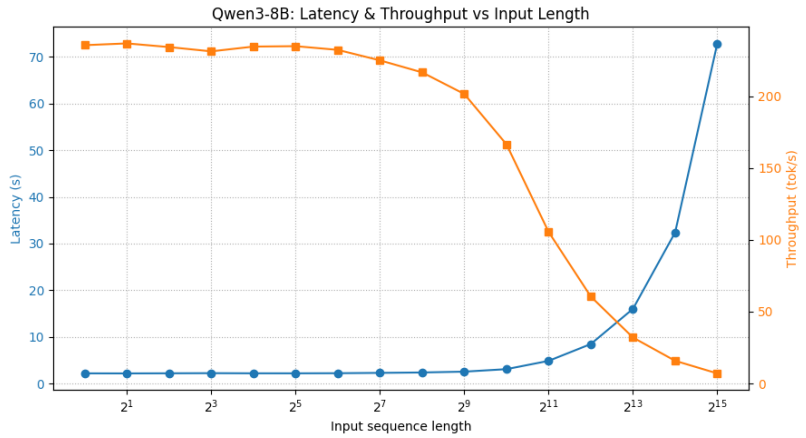
Using `Qwen/Qwen3-8B` at half precision (use bfloat16 unless you must use older hardware that does not support it), a batch size of 8, and an output sequence length of 64 (new) tokens, sweep over input sequence lengths of $\{2^n : 0 \leq n \leq 15\}$.

Be sure to perform 2-5 warm-up iterations (fewer needed with longer sequences), and don't forget to run `torch.cuda.synchronize()` after your model finishes running. Measure and report wall-clock time (just use `time.time()`)[5] and token throughput in tokens / second – each number should be an average over 10 batches. Fill out the table provided (only for subset of your sweep, but **also note configurations that led to OOM errors**), and include a figure(s), plotting each metric against each input sequence length. Figures may be either multiple plots or a single combined plot overlaying all three metrics on scales that make sense.

Solution:

[5]GPU execution time as measured with `torch.cuda.Event()` is often slightly less than the total CPU wall clock time, but in these particular settings, especially with single GPU inference with vanilla PyTorch/HF there is very little deviation expected

Input length (tokens)	Time (s)	Throughput (tokens/s)
1	2.173	235.570
4	2.186	234.239
16	2.183	234.585
64	2.204	232.289
256	2.363	216.623
1024	3.074	166.530
4096	8.457	60.538
16384	32.389	15.807
32768	72.908	7.022



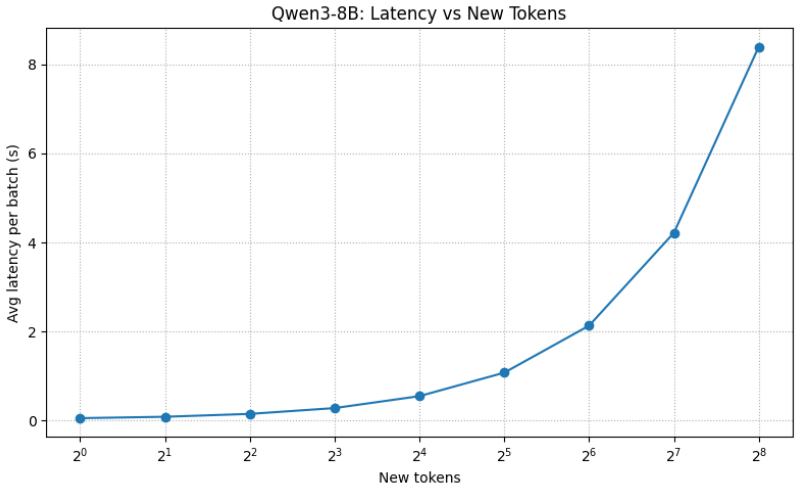
□

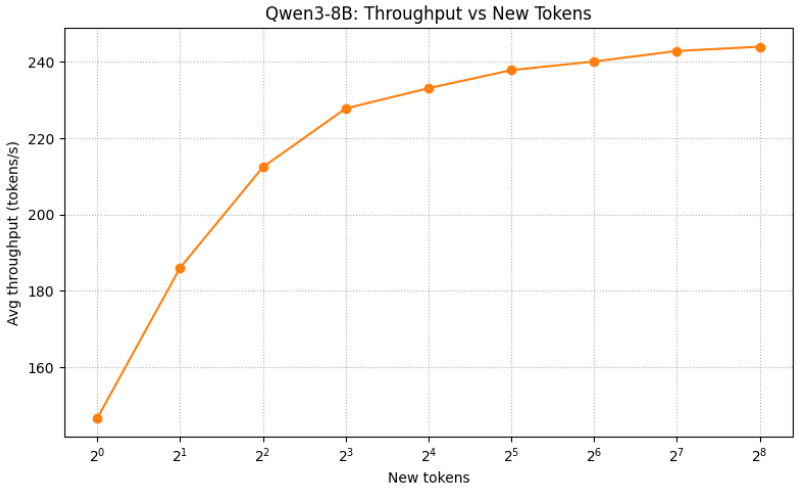
1.3.2 Varying output sequence lengths

Repeat the same as above, but with input size 64, and output sizes over $\{2^n : 0 \leq n \leq 8\}$.

Solution:

Output length (tokens)	Time (s)	Throughput (tokens/s
1	0.054	146.679
4	0.1506	212.441
16	0.549	233.105
64	2.132	240.104
256	8.395	243.973





□

1.3.3 Varying the model

Now fix input=64 tokens, output=64, and repeat for three models: `Qwen/Qwen3-1.7B`, `Qwen/Qwen3-8B`, and `allenai/OLMo-7B-0724-hf`. No need for a figure for this one, but report all values in the table, and write 1-2 sentences of reflection and/or analysis.

Solution:

Model	Time (s)	Throughput (tokens/s
`Qwen/Qwen3-1.7B`	1.709	299.725
`Qwen/Qwen3-8B`	2.150	238.078
`allenai/OLMo-7B-0724-hf`	1.269	403.172

OLMo-7B significantly outperforms both Qwen models in throughput despite having a similar parameter count to Qwen3-8B, suggesting architectural differences or implementation optimizations.

The Qwen3 models likely perform worse due to the `enable_thinking=True` default setting, which generates hidden reasoning tokens that increase computational overhead without contributing to the measured visible output tokens. □

2 Implementation of KV caching [25 points]

One essential component of generation with auto-regressive transformers is **KV caching**, or caching of computed key and value tensors from previous generation steps such that only a partial computation is needed at each new generation step. In this section, we ask you to 1) empirically observe inference with and without KV caching for a standard LLM, and 2) implement your own KV cache for a minimal transformer model.

2.1 Benchmarking with vs. without KV caching

In general, generating without a KV cache is quite slow and almost never done in practice, and longer sequences (both input and output) require more time. In this section, you will perform benchmarking across a variety of settings and gain intuitions about factors that can lead to meaningful differences in *scaling patterns* seen as output sequence length grows. You will compare a short prompt with a long prompt, a small model with a larger model, and an older model with a newer model... We provide a starter notebook, `benchmark-kv-cache.ipynb` and a separate file containing the longer prompt (which gets tokenized to approximately 32k tokens, in `long_prompt.txt`). You do *not* need to submit the notebook.

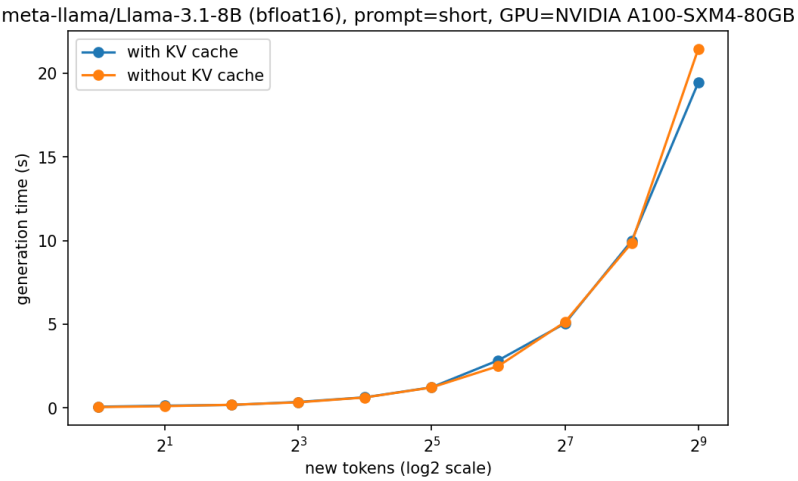
2.1.1 Baseline figure

The "baseline" benchmark uses 'meta-llama/Llama-3.1-8B' (at half precision) and a short prompt, "Once upon a time,". See the notebook for more complete instructions. Show your figure comparing generation time with vs without a KV cache, and report the GPU hardware you are using as well. **For complete instructions, see `benchmark-kv-cache.ipynb`.**

Solution:

GPU: A100-SXM4-80GB

Figure:



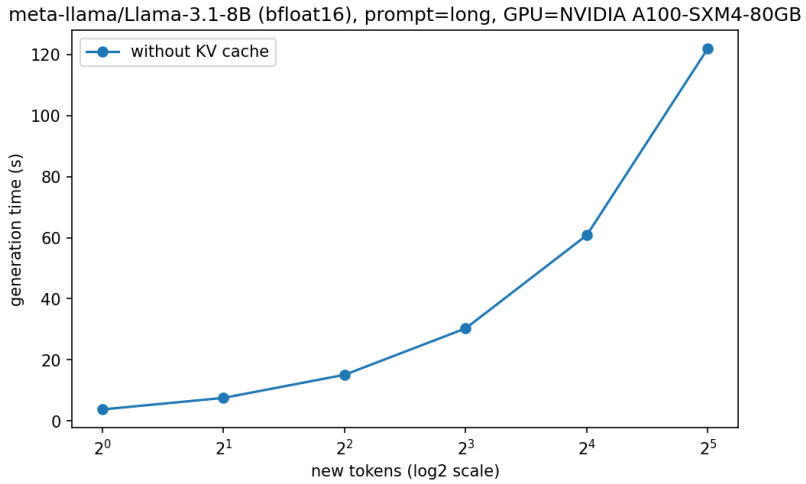
Description of trends: For smaller output lengths (2-128 tokens), both configurations show similar generation times because the overhead of recomputing attention for short sequences is minimal compared to cache management costs. However, as the output length increases beyond 128 tokens, the no-cache approach becomes significantly slower. This shows the decoding efficiency gained from KV caching, as recomputing attention keys and values for all previous tokens at each step becomes increasingly expensive without caching. □

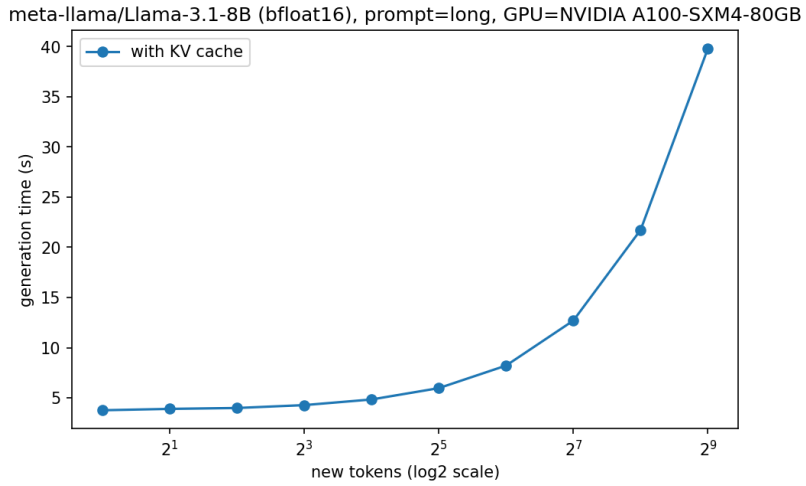
2.1.2 Longer prompt

Follow the instructions in the notebook to build a cache-vs-no-cache figure for the much longer prompt provided.

Solution:

Figure:





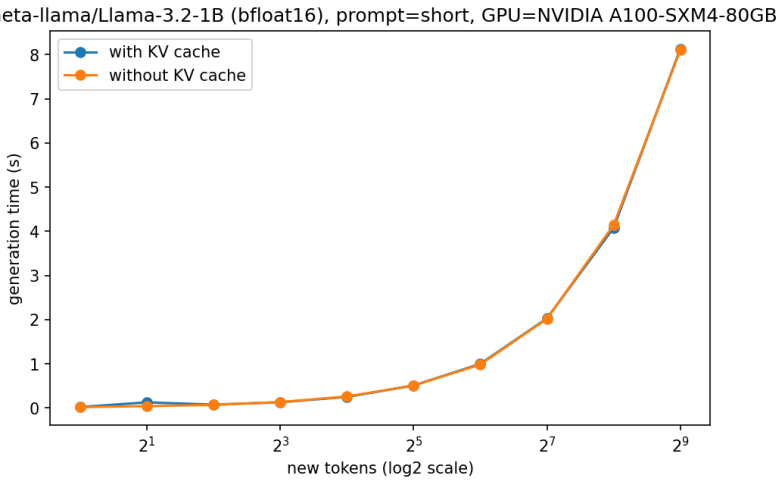
Comparison and explanation: With the longer prompt, the no-cache configuration shows dramatically higher generation times even for short output sequences. The KV cache configuration performs

inference much faster, demonstrating that KV caching benefits are amplified with longer input sequences. This is because without caching, attention must be recomputed over all input tokens plus previously generated tokens at each decoding step, creating a much larger computational burden than the baseline short prompt scenario. □

2.1.3 Smaller model

Follow the instructions in the notebook to build a cache-vs-no-cache figure for a smaller model, 'meta-llama/Llama-3.2-1B'.

Solution:
Figure:

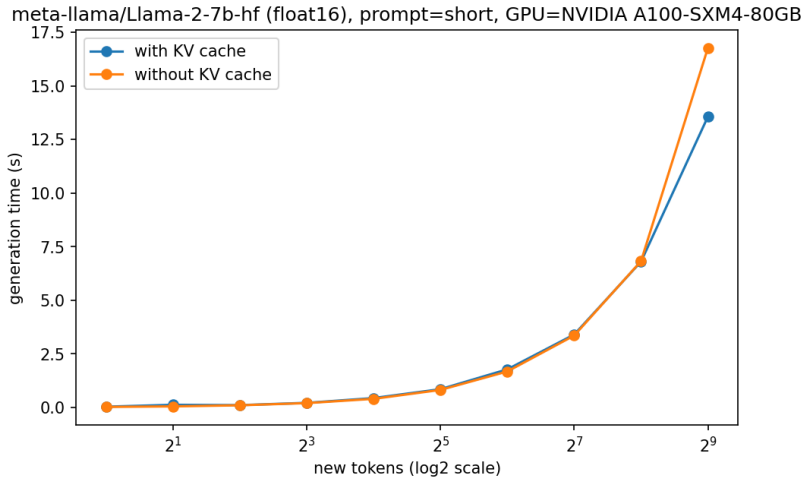


Comparison and explanation: The graph shows minimal difference between the two configurations for the 1B parameter model. This is mainly because the model does not take alot of time to recompute the attention values, compared to accessing the cached values. Hence, KV caching is more effective in bigger models compared to smaller ones. However, I believe that with longer output lengths, there would be a considerable increase in generation time without using KV cache. □

2.1.4 Older model

Follow the instructions in the notebook to build a cache-vs-no-cache figure for a smaller model, 'meta-llama/Llama-2-7b-hf'.

Solution:
Figure:



Comparison and explanation: For shorter output lengths, both configurations show similar performance. However, as output length increases, the cache configuration performs significantly faster than the no-cache setting. This difference is greater for the older model, suggesting that the older Llama-2-7b architecture may lack certain optimizations present in newer models (such as improved attention implementations or memory access patterns), thereby amplifying the relative benefits of KV caching. □

2.2 Implementing a KV cache

Begin with the starter code provided in `kv-cache-implementation.py` and submit your code along with your other deliverables. Be sure to include your name and andrewid at the top of your file in a comment. **Please leave the "TODO" lines intact in your code,** as this will help us with grading.

3 Batching of requests [20 points]

In §1, you were asked to consider the sizes of the models and data at inference time alongside the physical constraints of common GPU hardware. In this section, we focus on batching concerns specifically.

3.1 Conceptual questions

In LLM pre-training, documents are commonly packed together such that all sequences in each batch are a uniform maximum length. This allows for relatively token throughput and relatively high GPU utilization, but this is clearly not universally appropriate or practical at inference time when use cases and deployment settings vary.

3.1.1 Batch or no batch?

In general, inference with a batch size of one tends to lead to poor utilization in a modern GPU. However, it is often assumed or performed in practice. Describe *two* examples of settings where inference with a batch size of 1 is appropriate or even necessary. Prioritize describing specific traits and characteristics of these settings, as opposed to specific instances of these settings (e.g. instead of just naming a specific dataset or model, describe the characteristics that justify its inclusion). Write 2-3 sentences for each example, including an explanation.

Solution:
Batch size of one could be useful for real time interactive applications, such as conversational AI systems, voice assistants or real time coding tools because it is necessary for each user query to be processed immediately without waiting for multiple requests to accumulate. Hence, Time to First Token (TTFT) plays as a critical metric rather than the throughput, since batching requests would introduce unacceptable delays for user queries.
Another situation where batch size one would be beneficial is when there are tight constraints on the memory size. Multiple prompts would be of different sequence lengths, and batching them together would lead to the GPU memory being excessively used in storing padding tokens (used to pad the shorter prompts). This would lead to the GPU processing multiple useless tokens, leading to making batched inference much slower and may also exceed the GPU memory entirely. □

3.1.2 Static vs. dynamic vs. continuous batching

Naive batched inference pads sequences to the longest in each batch. In practice, these days it is almost always best to use an inference engine like vLLM or SGLang that does continuous batching, but not all inference settings are created equal, and not all inference optimizations are universally helpful. Describe *one* example of a setting where the benefits of continuous batching over naive batched inference might be relatively limited, and *one* example of a setting where continuous batching would be especially helpful in reducing some meaningful efficiency metric.

Solution:
Continuous batching would have limited effect in scenarios where the prompts have fixed lengths. In this case, all the prompts would be processed and sent out in batches at equal times, thereby, leaving no slot for another prompt to be processed while the other prompts get processed at the same time. However, when the prompts have variable lengths, continuous batching excels in serving scenarios with constant incoming requests that have highly variable generation lengths (like a chatbot handling both simple yes/no questions and complex explanation requests). In naive batching, short requests must wait for the longest request in their batch to complete before the next batch can begin, causing severe queuing delays and poor GPU utilization when the batch contains one 2000-token generation alongside several 50-token generations. Continuous batching eliminates this effect by immediately filling freed slots when short requests complete, dramatically improving both throughput (by maintaining high GPU utilization) and latency (by reducing queue wait times for incoming requests). □

3.2 Pseudo-code for continuous batching with disaggregated prefill and decode

Continuous batching handles not only variation in input sequence lengths (where requests might be bucketed into groups of similar input sequence lengths in order to reduce the amount of padding needed) but also variation in output lengths (where requests finish at different times and are "kicked out" of the GPU at different times). Although requests are sometimes sent with information about minimum or maximum generation length, dynamic serving settings are inherently associated with some uncertainty about output sequence length.

More recently, state of the art inference frameworks have begun supporting *disaggregated* prefill and decode, or disaggregated PD [3]. In this model, one spins up separate e.g. vLLM instances, each dedicated to just one of prefill or decode. KV caches calculated from prefill are saved and sent to the decode server. This has a number of advantages over traditional continuous batching: requests can have very similar input sizes

but very different generation lengths, or vice versa; and prefill is compute-bound and highly parallelizeable, while decode requires sequential processing and tends to be a blocking process despite using less compute in a given moment.

Your task: For this problem, you are asked to write pseudo-code as parts of a basic implementation of continuous batching with PD disaggregation. You will design two concurrent loops (one for prefill and one for decode) that together serve all requests. We provide a pseudo-code REQUEST class for you that you may modify if needed. Your pseudo-code should be at an abstract enough level that it should not matter whether or not the underlying memory is continuous.[6]

Assumptions:

- Reasonable relevant variable such as `kv_cache_allocation`, `current_prefill_requests`, `curent_decode_requests`, and `finished_requests` have been pre-initialized.
- You may make up new auxiliary variables as needed.
- Requests have been already been added to a queue, `requests_remaining`
- This is an offline serving setting such that we can assume no further requests will be added as the initial ones are being served
- Basically, no failure modes: feel free to assume that KV cache storage, communication, and loading will always happen at a reasonable speed, that we never run out of disk space, and that we will never have occasion to requeue a request.

We are not looking for any one specific answer. Instead, credit will be awarded for well-thought algorithms with justification. That being said, be sure to address:

1. Conditions for starting a queued request
2. Conditions for stopping generation for a request
3. State tracking and updating of requests
4. Batching requests appropriately for prefill, grouping by similar input lengths
5. Dynamic batch management in decode. You may optionally write an additional BATCH class to help clarify your implementation
6. KV cache management (high-level: update, store, load, check sizes)

Magical function calls you may imagine you have at your disposal (unless, of course, you choose to implement them as your additional feature):

1. `does_fit()`, which takes in a request, a collection of requests already in memory, and a total KV cache size and checks (at negligible cost) whether the single request can fit in the GPU. Intended to work for prefill or decode, with the caveat that it will default to the maximum sequence length for the model if no output sequence length is set (and so `does_fit()` should be treated as a conservative bound).

[6] In practice, paged attention [4] is typically used with continuous batching in order to alleviate the significant memory fragmentation and associated memory inefficiency that can occur with continuous batching.

2. `constrained_get()`, a PriorityQueue class method which takes a condition (described in pseudo-code or natural language :)) and returns the next request that fulfills the condition. Intended to work for either prefill and decode

For up to 3 bonus points on this assignment, incorporate *one* of the following additional features (as pseudo-code) into your pseudo-code:

1. A PriorityQueue class implementation that implements length bucketing logic
2. Explicit management of KV cache storage and communication/movement between servers
3. Want to do something else? Feel free to make a Piazza post and get pre-approval

Feel free to write your pseudo-code in a separate file and ask an LLM for help with formatting it into LaTeX :) **Please write your new lines of code and any modifications of provided code in this color**.

Solution:

Algorithm 1 Class Request

```
class Request:
    Input: request_id, input_tokens, output_length (optional)
    self.id ← request_id
    self.input_tokens ← input_tokens
    self.output_length ← output_length
    self.kv_cache ← None # Hint: update during prefill, and check + update during decode
    self.state ← "queued" # Hint: update this to "prefill" → "decode" → "done"
    self.generated_toks ← []
    # You may add additional initializations here
    self.input_length ← len(input_tokens)
    self.kv_cache_path ← None # For storage between prefill/decode
    self.max_length ← MAX_SEQ_LEN if output_length is None else output_length
    self.current_length ← 0 # Track decode progress

    Function has_prefill_requests(self):
        Return True if not any requests have not yet reached "decode" or "done"

    Function has_decode_requests(self):
        Return True if not any requests have not yet reached "done"

    Function add_token(self, token):
        Append token to self.generated_toks

    # You may add additional functions here
    Function is_finished(self):
        if self.output_length is not None then
            Return len(self.generated_toks) ≥ self.output_length
        Return EOS_TOKEN in self.generated_toks or len(self.generated_toks) ≥ self.max_length
```

Algorithm 2 Prefill loop (runs simultaneously with decode loop)

```
while requests_remaining.has_prefill_requests() do

    prefill_batch ← []

    while len(prefill_batch) < MAX_PREFILL_BATCH_SIZE do
        if requests_remaining.empty() then
            break

        if len(prefill_batch) == 0 then
            req ← requests_remaining.get()
        else
            target_len ← prefill_batch[0].input_length
            req ← requests_remaining.constrained_get(
              condition: pick only those prompts that have to generate around the same amount of new input tokens)
            if req is None then
                break

        if does_fit(req, prefill_batch, kv_cache_allocation) then
            prefill_batch.append(req)
            req.state ← "prefill"
        else
            requests_remaining.put_back(req)
            break

    if len(prefill_batch) == 0 then
        continue

    for req in prefill_batch do
        req.kv_cache ← model.prefill(req.input_tokens)
        req.kv_cache_path ← save_kv_cache_to_disk(req.id, req.kv_cache)
        send_to_decode_server(req)
        req.state ← "decode"
        current_decode_requests.add(req)
        req.kv_cache ← None
```

Algorithm 3 Decode loop (runs simultaneously with prefill loop)

```
while requests_remaining.has_decode_requests() do
    # TODO: your pseudo-code implementation here
    for req in current_decode_requests do
        if req.is_finished() then
            current_decode_requests.remove(req)
            req.state ← "done"
            finished_requests.add(req)
            delete_kv_cache(req.kv_cache_path)
            req.kv_cache ← None

    while len(current_decode_requests) < MAX_DECODE_BATCH_SIZE do
        new_req ← get_from_decode_queue()
        if new_req is None then
            break
        new_req.kv_cache ← load_kv_cache(new_req.kv_cache_path)
        if does_fit(new_req, current_decode_requests, kv_cache_allocation) then
            current_decode_requests.add(new_req)
        else
            put_back_in_decode_queue(new_req)
            break

    if len(current_decode_requests) == 0 then
        continue

    current_decode_requests.step() # generates one token per sequence currently in the batch
    for req in current_decode_requests do
        last_tok ← req.generated_toks[-1] if req.generated_toks else req.input_tokens[-1]
        new_token, updated_kv ← model.decode_one_token(last_tok, req.kv_cache)
        req.add_token(new_token)
        req.kv_cache ← updated_kv
        req.current_length ← req.current_length + 1
```

□

Use the space below to describe (in regular English) any assumptions you have made, along with a description of your algorithm and what it does. List and describe also any optional feature(s) you have included

Solution:

The algorithm assumes a system with separate prefill and decode servers that run simultaneously and independently. Both servers have access to shared storage (disk or network storage) where KV caches can be efficiently saved and loaded to enable communication between servers. The `does_fit()` function estimates whether adding a new request to the current batch would exceed available GPU memory by considering both the KV cache size and the maximum possible generation length. Communication between servers is handled through functions like `send_to_decode_server()`, `get_from_decode_queue()`, and `put_back_in_decode_queue()` which transmit metadata about re-

quest readiness and KV cache locations. Also, it is assumed that GPUs are communicating with eachother using NVLink.

The prefill loop is responsible for processing the input tokens of each request and creating their initial KV caches. It first builds batches of requests with similar input lengths to minimize padding overhead. For each batch, the algorithm first selects an initial request arbitrarily. Then, it attempts to add more requests to the batch using `constrained_get()`, which retrieves only those requests whose input lengths are within a window (defined by `LENGTH_BUCKET_SIZE`) of the first request's input length. This length-bucketing strategy ensures that requests in the same batch have similar input lengths, reducing wasted computation on padding tokens.

The algorithm checks memory constraints using `does_fit()` before adding each request to ensure the batch doesn't exceed GPU memory capacity. Once a batch is filled (up to `MAX_PREFILL_BATCH_SIZE`) or no more compatible requests are available, the prefill computation proceeds. For each request in the batch, the model processes all input tokens in parallel via `model.prefill()`, generating the KV cache. The KV cache is then saved to shared storage using `save_kv_cache_to_disk()`, and the request metadata is sent to the decode server. The request's state is updated to "decode," and the KV cache is freed from the prefill server's GPU memory to make room for the next batch.

The decode loop manages the batch of requests that are currently generating output tokens. It operates in three main phases per iteration.

First, in the cleanup phase, the loop checks all requests in the current decode batch and removes any that have finished generating, as determined by `is_finished()`, which checks if the output length is reached, an EOS token is generated, or the maximum sequence length is exceeded. For finished requests, the KV caches are deleted from shared storage via `delete_kv_cache()` to free resources.

Second, in the batch addition phase, the loop attempts to add new requests that have completed prefill to the current decode batch, up to `MAX_DECODE_BATCH_SIZE`. For each new request, it loads the KV cache from shared storage using `load_kv_cache()` and checks if the request fits in GPU memory. If a request doesn't fit, it's placed back in the queue to be tried again in a future iteration when memory becomes available.

Third, in the token generation phase, for each request in the current batch, the algorithm generates exactly one token by calling `model.decode_one_token()` with the last generated token (or the last input token if generation just started) and the request's KV cache. The new token is appended to the request's generated tokens, and the KV cache is updated to include the new token's key-value pairs. The `current_length` counter is incremented to track decode progress. This continuous batching approach allows the decode server to dynamically adjust its batch size as requests finish at different rates, maximizing GPU utilization while avoiding head-of-line blocking.

There are many optional features that I have included in the algorithm. The prefill loop implements intelligent batching by grouping requests with similar input lengths. The `constrained_get()` function retrieves requests where $|\text{req.input\_length} - \text{target\_len}| < \text{LENGTH\_BUCKET\_SIZE}$, ensuring that all requests in a batch have input lengths within a specified window of each other. This minimizes padding overhead during parallel prefill computation, as shorter sequences don't need to be padded excessively to match much longer sequences. The tradeoff is that stricter length bucketing may result in smaller batch sizes when compatible requests aren't available, but the reduction in wasted computation on padding tokens typically outweighs this cost.

The algorithm implements a complete lifecycle for KV cache management between the disaggregated prefill and decode servers. After prefill, each request's KV cache is saved to shared storage via

`save_kv_cache_to_disk(req.id, req.kv_cache)`, which stores the cache and returns a file path identifier. The `send_to_decode_server(req)` function transmits request metadata (including the KV cache path) to the decode server, allowing it to know which requests are ready for token generation. When the decode server adds a new request to its batch, it loads the KV cache from storage via `load_kv_cache(new_req.kv_cache_path)`. After saving the KV cache, the prefill server immediately sets `req.kv_cache` $\leftarrow$ `None` to free GPU memory. Similarly, when a request finishes generation, the decode server deletes the stored cache via `delete_kv_cache(req.kv_cache_path)`. This storage mechanism enables true disaggregation, allowing prefill and decode servers to scale independently and potentially run on different hardware optimized for their respective workloads.

□

4 Speculative Decoding [30 Points]

4.1 Background

Large language models (LLMs) typically perform *autoregressive decoding*, generating one token at a time. This process is often *memory-bound*, meaning throughput is limited by the transfer of model weights rather than compute.

Speculative decoding accelerates generation by using a small, fast *draft model* to propose multiple tokens at once, which are then verified in parallel by a larger *target model*.

The method was independently introduced in *Fast Inference from Transformers via Speculative Decoding* [5] and *Accelerating Large Language Model Decoding with Speculative Sampling* [6].

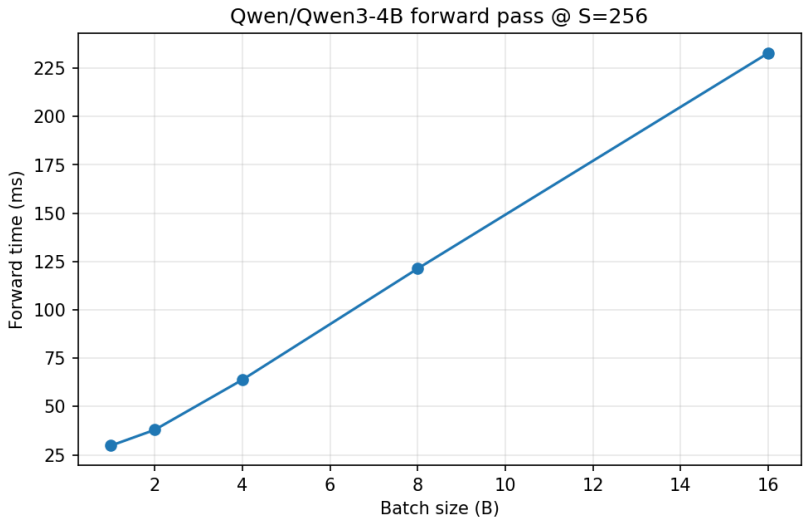
We will use **Algorithm 2** from the second paper as our reference for the implementation. Reading Theorem 1 from paper 2 is optional but helpful for intuition about why rejection sampling preserves the target distribution.

4.2 Benchmarking Forward Pass on a Single GPU

Using model `Qwen/Qwen3-4B`:

- Measure the forward-pass time for batch sizes $B \in \{1, 2, 4, 8, 16\}$ at a fixed sequence length $S = 256$.
- Run each configuration for 5 trials after appropriate warm-up iterations to obtain the average time. Use `torch.cuda.Event`, `torch.cuda.time_record` for accurate GPU timing.
- **Plot results:** Batch Size vs. Wall-Clock Time (ms).
- **Briefly discuss:** How does the forward cost scale with B, and why is this relevant for speculative decoding?

Solution:



The forward cost scales linearly with batch size due to the increased computational requirements for processing more tokens in parallel. However, this demonstrates efficient GPU utilization. The relevance to speculative decoding is that the verification step can process multiple draft tokens in a single forward pass at roughly the same cost as processing a single token. This parallel verification makes speculative decoding effective for batch size of 1, as the target model can verify multiple draft tokens simultaneously rather than generating tokens one at a time, thereby spreading out the memory-bound costs of autoregressive generation. □

4.3 Implementation

Implement speculative decoding in the provided scaffold `specdec.py`. You only need to modify the `decode()` function; model loading and tokenization are handled for you.

You are given 20 test prompts (`prompts.jsonl`) and a benchmarking script (`benchmark.py`) that runs inference over them. You may edit it for finer timing analysis.

Solution:
I have filled out the specdec.py file. □

4.4 Evaluation

In practice, speedups depend on both software and hardware factors. In this section, you will analyze how different target–draft pairs and lookahead values affect throughput under heterogeneous request lengths.

All experiments should be run on a single GPU. Each configuration should fit comfortably on GPUs with $\geq$48 GB VRAM.

Run your implementation with the following configurations:

- Target = Qwen-3-8B; Drafts = {Qwen-3-1.7B, Qwen-3-0.6B}
- Target = Llama-3.1-8B; Draft = Llama-3.2-1B

For each configuration, evaluate with speculative lookahead $\gamma \in \{2, 3, 5, 7\}$ and record:

- Empirical speedup = (Wall-clock time of AR baseline) / (Wall-clock time of speculative decoding)
- Empirical acceptance rate α

Deliverables

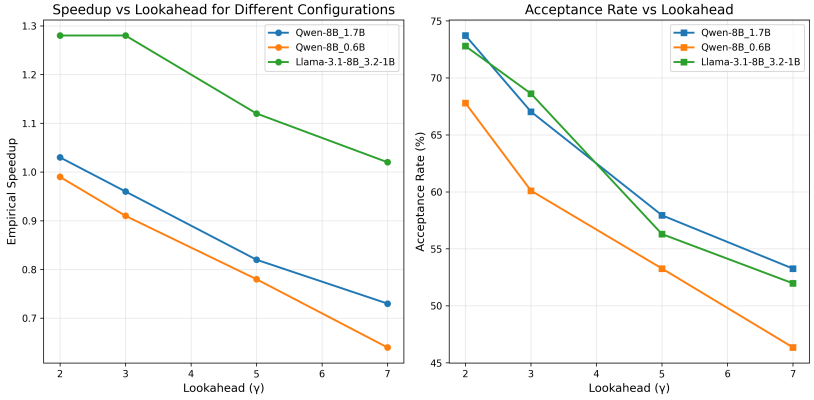
- Present results in a **table** and **plot the speedups vs. γ for each target–draft pair**.
- For your highest overall speedup configuration also comment on how the speedups vary per data source and why.

Include the GPU name and memory capacity in your table. Use `torch.manual_seed(42)` to ensure reproducibility.

Solution:

config_name	target	draft	lookahead	speedup	acceptance_rate	gpu_name (gb)
Qwen-8B_1.7B	Qwen/Qwen3-8B	Qwen/Qwen3-1.7B	2	1.03	73.71	A100-SXM4-80GB (79.25)
Qwen-8B_1.7B	Qwen/Qwen3-8B	Qwen/Qwen3-1.7B	3	0.96	67.03	A100-SXM4-80GB (79.25)
Qwen-8B_1.7B	Qwen/Qwen3-8B	Qwen/Qwen3-1.7B	5	0.82	57.94	A100-SXM4-80GB (79.25)
Qwen-8B_1.7B	Qwen/Qwen3-8B	Qwen/Qwen3-1.7B	7	0.73	53.26	A100-SXM4-80GB (79.25)
Qwen-8B_0.6B	Qwen/Qwen3-8B	Qwen/Qwen3-0.6B	2	0.99	67.78	A100-SXM4-80GB (79.25)
Qwen-8B_0.6B	Qwen/Qwen3-8B	Qwen/Qwen3-0.6B	3	0.91	60.10	A100-SXM4-80GB (79.25)
Qwen-8B_0.6B	Qwen/Qwen3-8B	Qwen/Qwen3-0.6B	5	0.78	53.27	A100-SXM4-80GB (79.25)
Qwen-8B_0.6B	Qwen/Qwen3-8B	Qwen/Qwen3-0.6B	7	0.64	46.35	A100-SXM4-80GB (79.25)
Llama-3.1-8B_3.2-1B	meta-llama/Llama-3.1-8B	meta-llama/Llama-3.2-1B	2	1.28	72.78	A100-SXM4-80GB (79.25)
Llama-3.1-8B_3.2-1B	meta-llama/Llama-3.1-8B	meta-llama/Llama-3.2-1B	3	1.28	68.61	A100-SXM4-80GB (79.25)
Llama-3.1-8B_3.2-1B	meta-llama/Llama-3.1-8B	meta-llama/Llama-3.2-1B	5	1.12	56.29	A100-SXM4-80GB (79.25)
Llama-3.1-8B_3.2-1B	meta-llama/Llama-3.1-8B	meta-llama/Llama-3.2-1B	7	1.02	51.97	A100-SXM4-80GB (79.25)

Table 1: Acceptance rate and speedup for different target draft model pairs and lookahead values.



□

4.5 Hardware Analysis

Speculative decoding assumes inference is primarily memory-bound rather than compute-bound. This balance can shift across GPU architectures.

Select the best- and worst-performing configurations (in terms of speedup) and rerun them on a different

GPU architecture. A *configuration* refers to a (Target, Draft, γ) combination.

1. Create a table including: VRAM capacity, memory bandwidth (GB/s), compute capability, and tensor core specifications for both GPUs (for bf16).

Solution:

GPU Model	Architecture	VRAM (GB)	Bandwidth (GB/s)	Compute Cap.	Tensor Cores	Tensor Core Gen	BF16 TFLOPS
NVIDIA A100-SXM4-80GB	Ampere	80.00	2039.04	8.0	432	3rd Gen (Ampere)	312
NVIDIA H100 80GB HBM3	Hopper	79.65	3352.32	9.0	456	4th Gen (Hopper)	989

Table 2: Comparison of NVIDIA A100 and H100 GPU architectures.

□

2. Report speedup on both GPUs and compare results.

Solution:

GPU	Config	Target Model	Draft Model	γ	AR Time (s)	SD Time (s)	Acceptance Rate	Speedup
A100-80GB	best_config	Llama-3.1-8B	Llama-3.2-1B	2	2.5085	1.9688	72.10%	1.27x
A100-80GB	worst_config	Qwen3-8B	Qwen3-0.6B	7	3.3407	5.0585	45.80%	0.66x
H100	best_config	Llama-3.1-8B	Llama-3.2-1B	2	1.6927	1.4166	71.82%	1.19x
H100	worst_config	Qwen3-8B	Qwen3-0.6B	7	2.3670	3.9663	45.58%	0.60x

Table 3: Best and worst configuration comparisons across A100 and H100 GPUs.

□

3. Briefly explain the observed differences in speedup, referencing hardware factors such as memory bandwidth and TFLOPS.

Solution:

The H100 shows slightly lower speedups than the A100, which is counterintuitive given its superior hardware specifications. The main reason is that batch size of 1 is being run, which significantly underutilizes both GPUs, but especially the H100. The H100's compute capability increased far more than its memory bandwidth, making autoregressive decoding even more memory-bound on this architecture. When the baseline decoding is already heavily constrained by memory bandwidth, speculative decoding provides smaller relative gains because the additional overhead from draft generation and verification becomes more expensive compared to a baseline that is already quite fast.

Architectural differences also contribute to this behavior. The H100's Hopper architecture, with its 4th-generation tensor cores, is optimized for high-throughput workloads that rely on larger batch sizes. In contrast, batch-size-1 speculative decoding setup is a sequential workload. The A100's Ampere architecture appears to be better balanced for this style of computation, as its compute-to-bandwidth ratio and overall design align more closely with the small-batch, sequential nature of autoregressive generation. Thus, even though the H100 is objectively more powerful, its strengths do not align well with the specific demands of single-sequence speculative decoding.

□

4.6 Analysis

4.6.1 Background

Let T_T and T_D denote the time for one forward pass of the target and draft models respectively, and T_V the verification time for γ tokens by the target.

The total speculative decoding time is:

$$T_{\text{Total}}^{SD} = \gamma \cdot T_D(B, S) + T_V(B, S, \gamma).$$

Given draft token acceptance rate $\alpha \in [0, 1]$ and lookahead γ, the expected number of tokens generated in one verification step is [5]:

$$\Omega(\gamma, \alpha) = \frac{1 - \alpha^{\gamma+1}}{1 - \alpha}. \tag{4}$$

Thus, the expected average latency per token is:

$$T_{\text{Avg}}^{SD} = \frac{T_{\text{Total}}^{SD}}{\Omega(\gamma, \alpha)},$$

and the relative latency (normalized by the target model's cost) is [7]:

$$\frac{T_{\text{Avg}}^{SD}}{T_T} = \frac{1}{\Omega(\gamma, \alpha)} \left(\gamma \cdot \frac{T_D}{T_T} + \frac{T_V(\gamma)}{T_T} \right). \tag{5}$$

From Eq. 5, the speedup depends on three key factors: (i) acceptance rate and expected generated tokens, (ii) the draft-to-target cost ratio, and (iii) verification overhead.

4.6.2 Questions

Combine your empirical results in the light of the the factors discussed above (and other relevant factors) to answer the following questions.

1. How do speedup and acceptance rate vary with γ? How do these trends differ across draft sizes and model families and why?

   Solution:
   Both speedup and acceptance rate decline consistently as gamma increases across all configurations. Errors compound as the speculation window grows, leading to more rejections during verification. The speedup degradation occurs because the cost of generating and verifying draft tokens grows while acceptance rates fall, meaning most drafting work gets wasted. The Llama configuration dramatically outperforms both Qwen configurations in terms of speedup, despite having comparable acceptance rates. The key reason is that Qwen3-8B runs with `enable_thinking=True` by default, generating hidden reasoning tokens before producing visible output. The speculative decoding algorithm treats all tokens uniformly and doesn't distinguish between reasoning and output tokens. This means the draft models must predict both the complex chain-of-thought reasoning and the final answer, but these smaller drafts lack the reasoning capabilities needed to replicate the target's thought process. Even when drafts occasionally match the visible output, mismatches in the hidden reasoning tokens lead to rejections.

Llama-3.1-8B is a standard instruction-tuned model without reasoning mode that only generates visible output tokens. The Llama draft just needs to predict straightforward responses without replicating hidden reasoning, making draft-target alignment much simpler and resulting in substantially higher speedups despite similar acceptance rates. The overhead of processing reasoning tokens in Qwen likely contributes to worse speedups even when acceptance occurs. Hence, speculative decoding works best when the draft model closely matches how the target generates tokens. For reasoning models, standard speculative decoding struggles because draft models can't replicate structured reasoning chains, creating a distribution mismatch that reduces acceptance rates. □

2. **Optimal γ:** As you saw from your experiments, the speedup is a function of the γ value.

   How would you determine the optimal γ for your given system? Which factors should be considered?

   Solution:
   To determine the optimal gamma for a given system, I would measure wall-clock speedup across a range of gamma values and select the configuration that maximizes actual speedup. From the experimental results, the optimal gamma is consistently two or three across all model pairs tested.
   The key factors to consider when determining optimal gamma are: empirical speedup (the most important metric), acceptance rate (higher is better but not the only factor), draft-to-target cost ratio (faster drafts allow higher gamma), verification overhead (grows with gamma), hardware characteristics (memory bandwidth and compute capability), and task characteristics (reasoning tasks may favor lower gamma due to token unpredictability).
   The optimal gamma represents a tradeoff point. At low gamma, we underutilize parallel verification capabilities. At high gamma, drafting and verification overhead dominates any gains from occasional multi-token acceptance. The empirical data shows gamma equals two or three hits this sweet spot, where acceptance rates remain high enough and overhead stays manageable enough to achieve maximum speedup. For reasoning models like Qwen3-8B, even lower gamma values may be optimal since predicting thinking tokens becomes increasingly difficult as the speculation window extends. □

3. **Batched Inference:** What challenges arise for speculative decoding with batch size > 1? Sketch pseudo-code for a batched version and discuss what makes the implementation challenging.

   You might want to think about this in detail, as it can be very useful for the final project.

   Solution:
   When the batch size is greater than 1, different prompts will have different acceptance rates during speculative decoding, which means that each sequence generates a different number of tokens at every iteration. Some sequences may accept all draft tokens, while others accept only a subset. This creates difficulties for efficient batched computation, because transformers rely on rectangular matrix multiplications, and sequences of unequal length require either padding or complex masking logic. Padding ensures that all sequences share a common length, but it introduces computational overhead: many padded positions do not correspond to real tokens,

yet still participate in attention and MLP operations. Additionally, each sequence maintains its own KV cache that grows at a different rate. When a sequence rejects early while others continue verification, the early-terminated sequence effectively idles, reducing GPU utilization. These variable-length KV caches also lead to non-contiguous memory layouts, making batched operations harder to optimize. Furthermore, whenever draft tokens are rejected, the KV cache needs to be truncated to reflect the true sequence length, which requires careful indexing and can be an expensive memory operation. Because sequences progress non-uniformly, some finish earlier than others, causing the effective batch size to shrink over time. This reduces parallelism and degrades throughput. Finally, although verification of γ draft tokens is parallel within each sequence, sequences accept different numbers of tokens, which forces synchronization before the next speculation round and complicates the orchestration of the batched decode loop.
The following pseudocode shows batched speculative decoding:

```
Function Speculative_Decoding:
1.  Make a list of all the prompts in the batch.
2.  Add padding tokens and construct an attention mask so that padded tokens
are ignored, ensuring equal sequence lengths.
3.  Run the draft model to generate draft tokens and draft probabilities for
all sequences.
4.  Run the target model on each prompt plus its draft tokens to obtain target
probabilities.
5.  Apply rejection sampling independently for each sequence and append the
accepted (and bonus) tokens to the corresponding prompts.
6.  Adjust the KV cache of each sequence according to the number of accepted
tokens.
7.  Repeat from step 1 until the maximum new token limit is reached.
```

□

References

[1] Kipply. Transformer inference arithmetic. https://kipp.ly/transformer-inference-arithmetic/, 2023. Accessed: 2025-11-02.

[2] Horace He. Making deep learning go brrrr: From first principles. https://horace.io/brrr_intro.html, 2020. Accessed: 2025-11-02.

[3] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving, 2024.

[4] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Haotong Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023.

[5] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding, 2023.

[6] Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling, 2023.

[7] Ranajoy Sadhukhan, Jian Chen, Zhuoming Chen, Vashisth Tiwari, Ruihang Lai, Jinyuan Shi, Ian En-Hsu Yen, Avner May, Tianqi Chen, and Beidi Chen. Magicdec: Breaking the latency-throughput tradeoff for long context generation with speculative decoding, 2025.