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

Your name (Andrew ID): Manav Nitin Kapadnis (mkapadni)

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 training, VRAM must hold gradients for all parameters, optimizer states (typically 2-3x the model size for optimizers like Adam), and intermediate activations from the forward pass needed for backpropagation, making it significantly more memory-intensive than inference. Generation requires only the static model weights and the dynamically growing KV cache, with minimal overhead for immediate computations. Training has more predictable memory requirements with fixed batch sizes, while generation's memory footprint varies with sequence length, though the overall memory usage remains substantially lower and more manageable than training. □

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:
Assumptions:

[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

- Batch size of 1 with minimal sequence length (e.g., single token or small prompt)
- FP16 precision: 2 bytes per parameter for model weights
- Activation memory and framework overhead: approximately 20% of model weight memory
- No KV cache needed (single forward pass, not autoregressive generation)

Calculation:
Let N be the number of parameters in billions.
Total memory required:

- Model weights: $N \times 10^9 \times 2$ bytes
- Activations + overhead: $N \times 10^9 \times 2 \times 0.2$ bytes
- Total: $N \times 10^9 \times 2 \times 1.2 = N \times 2.4 \times 10^9$ bytes

Setting equal to available memory:

$$N \times 2.4 \times 10^9 = 80 \times 10^9 \text{ bytes}$$

$$N = \frac{80}{2.4} \approx 33.3 \text{ billion parameters}$$

Answer: Approximately 33 billion parameters
Note: With more aggressive optimization (reducing overhead to ∼10%), up to 36B parameters might fit, but 33B is a conservative estimate that accounts for realistic activation storage during the forward pass. □

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:
Model Specifications:

- Qwen3-14B: 14.8B parameters
- 40 layers
- Grouped Query Attention: 40 query heads, 8 KV heads
- Head dimension: 128 (explicit in config)
- Hidden size: 5120

Hardware:

- L40S GPU: 48GB VRAM

Assumptions:

- FP16 precision: 2 bytes per parameter and per KV cache element
- Batch size: 8
- Empty prompt (all tokens are generated)

- Ignoring torch/launch overhead (only model weights + KV cache)

Calculation:

Model weights:

$$14.8 \times 10^9 \text{ parameters} \times 2 \text{ bytes} = 29.6 \text{ GB}$$

KV cache memory formula (GQA):

$$\text{KV Memory} = 2 \times b \times s \times L \times h_{kv} \times d_h \times \text{bytes}$$

where b = batch size, s = sequence length, L = number of layers, h_{kv} = KV heads, d_h = head dimension.

$$\text{KV Memory per token} = 2 \times 8 \times 40 \times 8 \times 128 \times 2 = 1{,}310{,}720 \text{ bytes}$$

Available memory for KV cache:

$$48 \text{ GB} - 29.6 \text{ GB} = 18.4 \text{ GB} = 19{,}756{,}130{,}304 \text{ bytes}$$

Maximum sequence length:

$$s = \frac{19{,}756{,}130{,}304}{1{,}310{,}720} \approx 15{,}069 \text{ tokens}$$

Answer: 15,070 tokens (to the nearest 10 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	3.50 GB	7.00 GB	14.00 GB
Qwen/Qwen3-1.7B	3.50 GB	7.00 GB	14.00 GB
Qwen/Qwen3-4B	4.50 GB	9.00 GB	18.00 GB
Qwen/Qwen3-8B	4.50 GB	9.00 GB	18.00 GB
Qwen/Qwen3-14B	5.00 GB	10.00 GB	20.00 GB
Qwen/Qwen3-32B	8.00 GB	16.00 GB	32.00 GB

□

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

Solution:

Model Configuration (from Qwen3 Technical Report):

- Layers: $L = 40$
- KV heads: $n_{\text{kv}} = 8$ (Grouped Query Attention)
- Head dimension: $d_{\text{head}} = 128$
- Sequence length: $S = 32{,}768$ tokens
- Precision: FP16 (2 bytes per parameter)

Calculation for Batch Size = 1:

Using the formula:

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

Substituting values:

$$\begin{aligned}\text{KV Cache Size} &= 2 \times 1 \times 32{,}768 \times 40 \times 8 \times 128 \times 2 \\ &= 5{,}368{,}709{,}120 \text{ bytes} \\ &= \frac{5{,}368{,}709{,}120}{1{,}073{,}741{,}824} \text{ GB} \\ &= 5.00 \text{ GB}\end{aligned}$$

For other batch sizes:

- Batch size 2: $5.00 \times 2 = 10.00$ GB
- Batch size 4: $5.00 \times 4 = 20.00$ GB

Key observations:

- All Qwen3 models use 8 KV heads (Grouped Query Attention), which significantly reduces KV cache size compared to Multi-Head Attention
- KV cache scales linearly with batch size
- Despite different model sizes, Qwen3-0.6B and Qwen3-1.7B have the same KV cache size (same layers, same KV heads), as do Qwen3-4B and Qwen3-8B

□

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:

KV cache scales linearly with batch size B for fixed sequence length S, depth L, KV heads n_{kv}, head dimension d_{head}, and dtype bytes, i.e., KV $\propto B$ by KV $= 2 \times B \times S \times L \times n_{\text{kv}} \times d_{\text{head}} \times$ bytes. Across the Qwen3 family, KV size does not directly scale with total parameter count; instead, it scales with architectural choices, primarily the number of layers and the KV attention shape while GQA keeps n_{kv} fixed at 8 and d_{head} roughly constant, so increases in KV size track increases in layers rather than total parameters.

Concretely, Qwen3-0.6B and Qwen3-1.7B both have 28 layers and therefore the same KV size at a fixed

S, while Qwen3-4B and Qwen3-8B share 36 layers and thus share a larger (but equal) KV footprint; Qwen3-14B (40 layers) and Qwen3-32B (64 layers) require progressively more KV, illustrating that KV growth is stepwise with depth and largely insensitive to width-driven parameter increases under GQA. For example, using Qwen3-14Bś configuration (L=40, n_{kv}=8, $d_{\text{head}}\approx$128) at FP16 and S=32,768, the KV cache is KV $= 2 \times B \times 32{,}768 \times 40 \times 8 \times 128 \times 2$ bytes $\approx$ 5.00 GB at B=1, 10.00 GB at B=2, and 20.00 GB at B=4.

□

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:
GPU used: NVIDIA A100-SXM4-80GB □

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:

Input length (tokens)	Time (s)	Throughput (tokens/s)
1	2.5027	204.58
4	2.4299	210.71
16	2.4441	209.48
64	2.4106	212.39
256	2.4036	213.01
1024	2.4505	208.94
4096	2.7188	188.32
16384	3.6956	138.54
32768	5.0958	100.47

□

[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

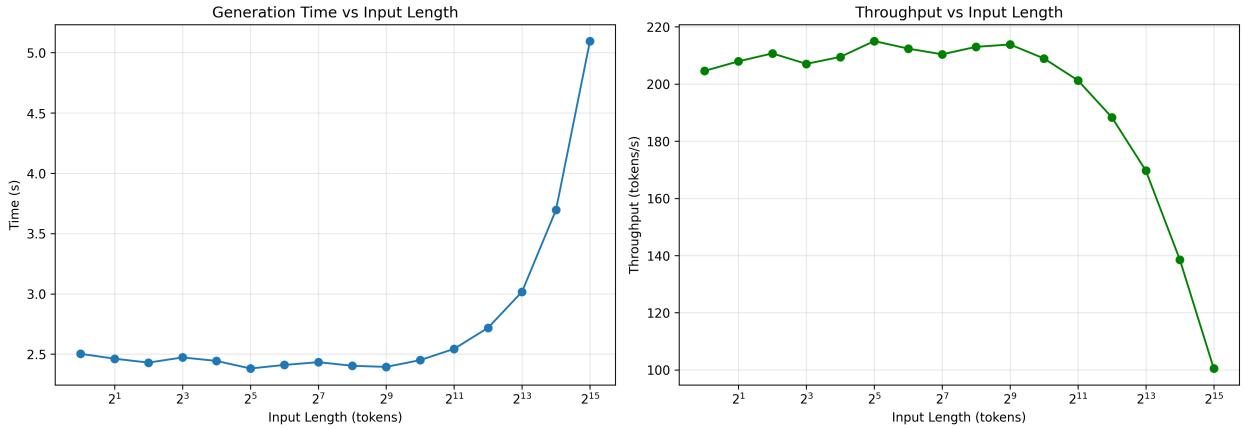

Figure 1: Input Sweep plots

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	0.0448	178.61
4	0.1622	197.27
16	0.6150	208.15
64	2.4255	211.09
256	9.9448	205.94

□

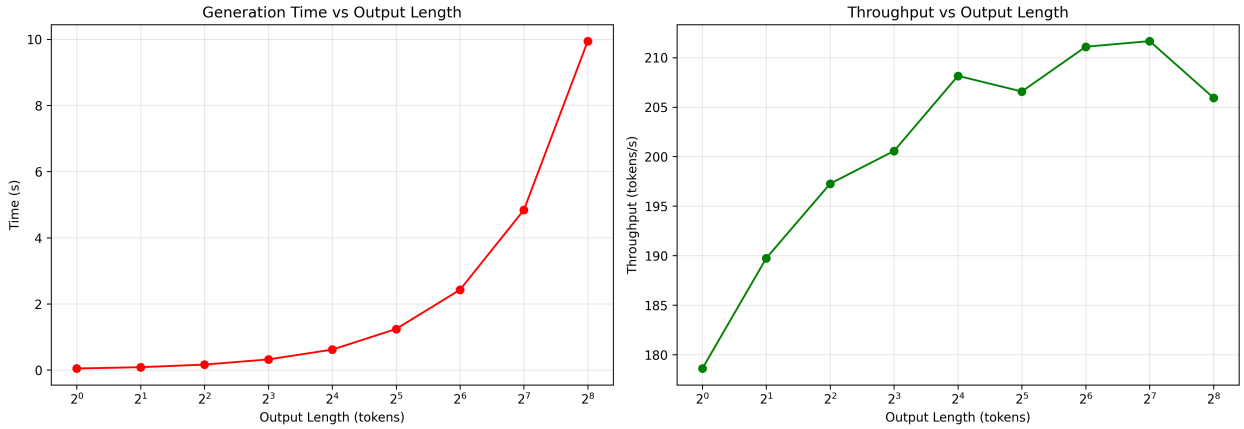

Figure 2: Output Sweep plots

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	1.957	261.54
Qwen/Qwen3-8B	2.5335	202.09
allenai/OLMo-7B-0724-hf	1.5475	330.85

Analysis: Surprisingly, OLMo-7B achieves the highest throughput (330.85 tokens/s) despite being a mid-sized model, outperforming both the smaller Qwen3-1.7B and larger Qwen3-8B by 26% and 64% respectively, suggesting that architectural choices, kernel optimizations, and implementation efficiency can significantly outweigh raw parameter count in determining inference performance. The Qwen3 models scale as expected with size (1.7B is 29% faster than 8B), but their lower throughput relative to OLMo-7B indicates potential differences in attention mechanisms (both use GQA with 8 KV heads but may have different hidden dimensions or layer counts) or framework-level optimizations that favor OLMo's architecture for this particular batch size and sequence length configuration. □

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: NVIDIA A100-SXM4-80GB

Model: `meta-llama/Llama-3.1-8B (short prompt)`

Figure: Please refer to Figure 3

Table:

Output length (tokens)	Time (s) with KV	Time (s) no KV
1	0.6669	0.0283
2	0.0913	0.0533
4	0.1050	0.1053
8	0.2094	0.2081
16	0.4136	0.4102
32	0.8796	0.8271
64	1.6529	1.6393
128	3.2885	3.3323
256	6.5680	7.3448
512	13.2021	18.2423

Average Speedup: 0.91x

Description of trends: For shorter prompts, both configurations perform similarly for small token counts. As sequence length increases, caching reduces recomputation costs, but since the prompt is small, the relative improvement is modest (below 1x speedup). The benefit grows with output length but remains limited. □

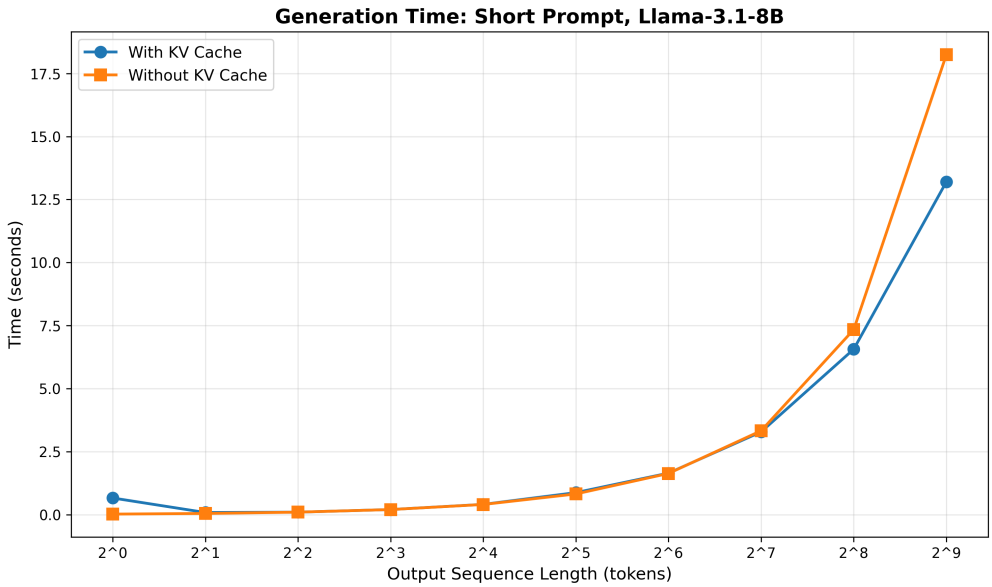

Figure 3: Short prompt output sweep with and without KV cache (Llama-3.1-8B).

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:

Model: `meta-llama/Llama-3.1-8B (long prompt)`

Figure: Please refer to Figure 4

Table:

Output length (tokens)	Time (s) with KV	Time (s) no KV
1	3.7533	3.7358
2	3.7638	7.4726
4	3.8258	14.9648
8	3.9345	29.9487
16	4.1535	59.9520
32	4.5890	120.0128
64	5.4544	–
128	7.1842	–
256	10.6284	–
512	17.5225	–

Average Speedup: 9.18x

Comparison and explanation: For long prompts, recomputation of attention grows quadratically without caching, making the no-cache setting extremely slow (up to 120s). With KV caching, time scales almost linearly with output length, giving over 9x speedup. This clearly demonstrates KV cache's effectiveness for large context windows. □

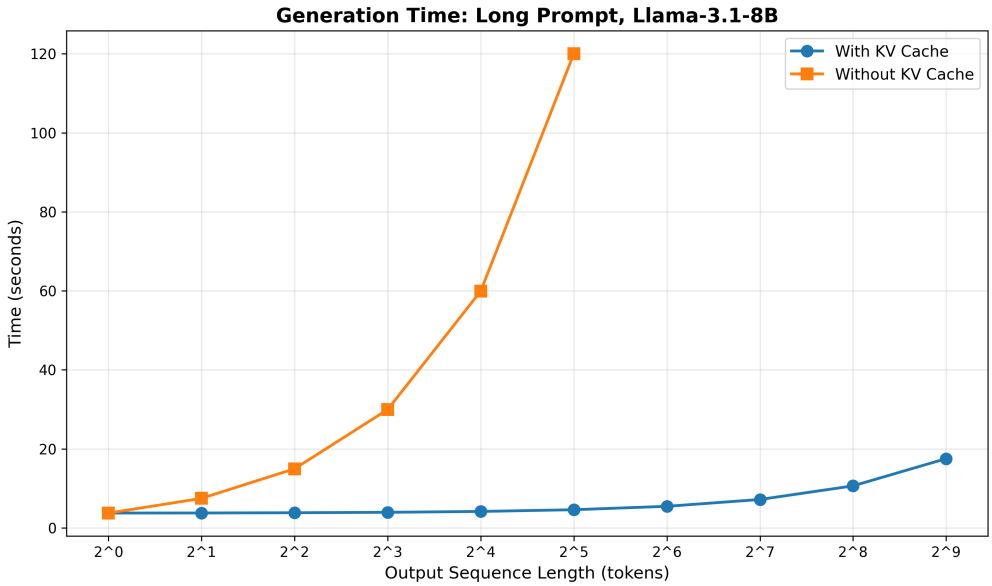

Figure 4: Long prompt generation times with and without KV cache (Llama-3.1-8B).

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:

Model: `meta-llama/Llama-3.2-1B (short prompt)`

Figure: Please refer to Figure 5

Table:

Output length (tokens)	Time (s) with KV	Time (s) no KV
1	0.0208	0.0166
2	0.0366	0.0306
4	0.0589	0.0592
8	0.1190	0.1154
16	0.2329	0.2191
32	0.4521	0.4530
64	0.8864	0.8890
128	1.7553	1.7535
256	3.5120	3.5086
512	7.0568	6.9885

Average Speedup: 0.95x

Comparison and explanation: For small models like Llama-3.2-1B, caching brings minimal performance difference because the computation overhead per token is low. The times are nearly identical across both setups, indicating that KV cache benefits scale with model size. □

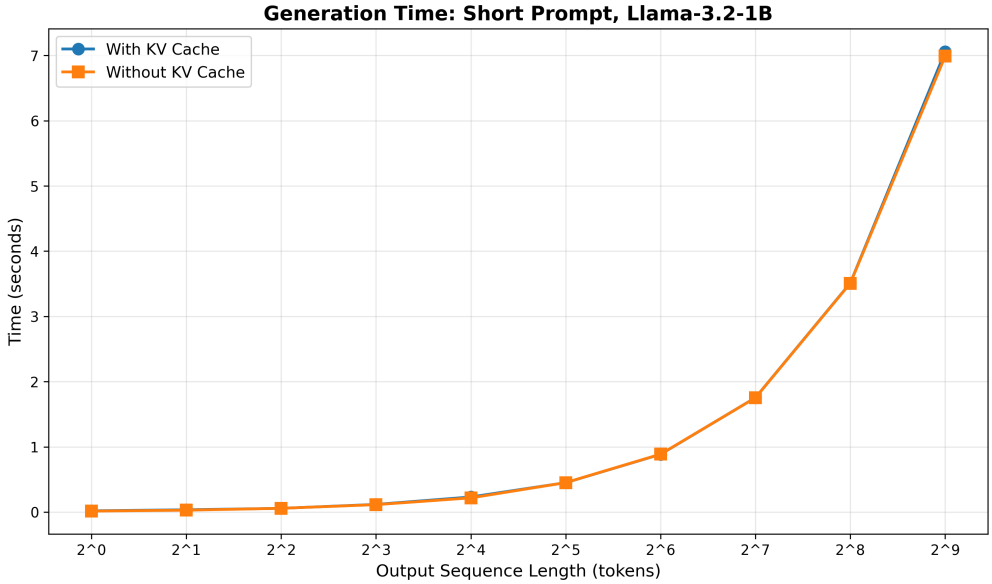

Figure 5: Short prompt output sweep (Llama-3.2-1B).

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:

Model: `meta-llama/Llama-2-7B (short prompt)`

Figure: Please refer to Figure 6

Table:

Output length (tokens)	Time (s) with KV	Time (s) no KV
1	0.0296	0.0277
2	0.0533	0.0499
4	0.1034	0.0994
8	0.2050	0.1983
16	0.4088	0.4005
32	0.8141	0.7937
64	1.6181	1.6529
128	3.2987	3.2701
256	6.6095	6.6992
512	13.3481	17.1834

Average Speedup: 1.01x

Comparison and explanation: For older architectures such as Llama-2-7B, caching provides minor but consistent efficiency gains. While throughput is improved slightly for longer outputs, the effect is less pronounced compared to newer models due to less optimized attention kernels. □

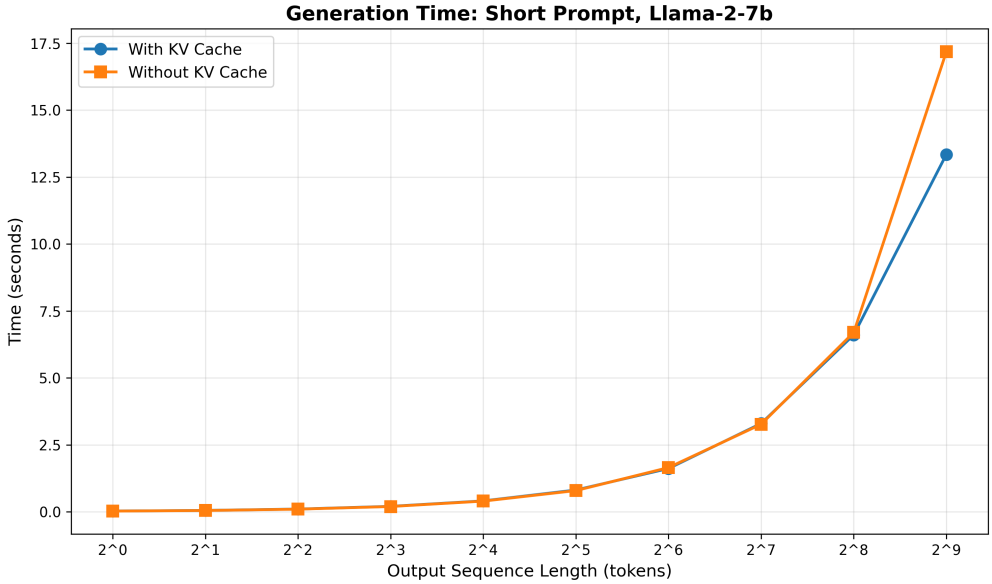

Figure 6: Short prompt output sweep (Llama-2-7B).

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:

Example 1: Low-latency interactive applications with strict real-time requirements

Batch size 1 is appropriate for real-time interactive systems such as conversational chatbots, voice assistants, or live coding assistants where users expect immediate responses with minimal latency (e.g., <100ms time-to-first-token). Batching inherently introduces waiting time as the system accumulates multiple requests before processing, which directly conflicts with the low-latency requirement since each request must wait for others to arrive before inference begins. Additionally, in scenarios with sporadic or low request arrival rates (e.g., a personal assistant or development tool used by a single user), there may not be enough concurrent requests to form meaningful batches, making batch size 1 the only viable option.

Example 2: Memory-constrained edge deployment with long context windows

Batch size 1 is necessary when deploying LLMs on resource-limited edge devices (e.g., mobile phones, IoT devices, or single-GPU edge servers) that have constrained VRAM, particularly when handling long-context inputs. Since KV cache memory scales linearly with both sequence length and batch size (KV Cache $= 2 \times B \times S \times L \times n_{\text{kv}} \times d_{\text{head}} \times$ bytes), even modest batch sizes can cause out-of-memory errors when combined with long sequences; for instance, increasing from batch size 1 to 4 quadruples the KV cache requirement, which may exceed available memory. In edge scenarios where the hardware cannot be upgraded and swapping to system RAM would introduce unacceptable latency penalties, batch size 1 ensures the model can process requests without memory exhaustion, even if it means lower GPU utilization.

□

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:

Setting where continuous batching benefits are limited:

Continuous batching provides limited advantages over naive batched inference in offline batch processing scenarios with uniform output lengths, such as evaluating a model on a fixed-format benchmark dataset (e.g., multiple-choice questions with single-token answers, or structured extraction tasks with predetermined output schemas). In these settings, all sequences in a batch generate approximately the same number of tokens, so the primary inefficiency of naive batching, idle GPU resources while waiting for the longest sequence to complete-is minimized since all sequences finish at roughly the same time. Additionally, since this is an offline workload without latency requirements, the overhead of continuous batching's dynamic scheduling and memory management may not justify its complexity compared to simpler static batching approaches that can optimize for maximum throughput with predictable memory allocation.

Setting where continuous batching is especially helpful:

Continuous batching is particularly beneficial in high-throughput online serving environments with high variance in output lengths, such as a production chatbot or code generation API handling diverse user queries ranging from simple factual questions (generating 10-50 tokens) to detailed explanations or complete program implementations (generating 500-2000 tokens). In naive batching, the entire batch must wait for the longest sequence to complete before starting the next batch, leaving GPU resources idle as shorter sequences finish early-this inefficiency compounds dramatically as output length variance increases, with throughput dropping significantly (as shown in benchmarks where naive static batching performance plummets with high-variance workloads). Continuous batching operates at the token level rather than the sequence level, allowing new requests to immediately occupy GPU resources freed by completed sequences, thereby maintaining high GPU utilization and dramatically improving throughput (often 2-10x) while also reducing average latency for shorter requests that would otherwise be blocked waiting for longer requests in their batch to complete.

□

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).
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

[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. 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 ← []
    self.input_length ← len(input_tokens) # For bucketing by similar lengths
    self.kv_cache_location ← None # Track storage location for disaggregated transfer
    self.max_length ← output_length if output_length else MAX_SEQ_LENGTH
    self.eos_generated ← False

    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
        if token == EOS_TOKEN: self.eos_generated ← True

    Function should_stop(self):
        Return self.eos_generated or len(self.generated_toks) ≥ self.max_length

    Function get_current_length(self):
        Return self.input_length + len(self.generated_toks)
```

Algorithm 2 Prefill loop (runs simultaneously with decode loop)

```
prefill_batch ← []
ready_for_decode_queue ← Queue() # Communication channel to decode loop
while requests_remaining.has_prefill_requests() do
    # Step 1: Try to form a batch with similar input lengths
    prefill_batch ← []
    target_length ← None

    while len(prefill_batch) < MAX_PREFILL_BATCH_SIZE do
        # Get next request with similar input length (within bucket tolerance)
        if target_length is None then
            next_req ← requests_remaining.constrained_get(state == "queued")
            if next_req is None: break
            target_length ← next_req.input_length
        else
            next_req ← requests_remaining.constrained_get(
              state == "queued" and abs(input_length - target_length) ≤ BUCKET_TOLERANCE)
            if next_req is None: break

        # Step 2: Check if request fits in GPU memory
        if does_fit(next_req, prefill_batch, PREFILL_GPU_MEMORY):
          prefill_batch.append(next_req)
          next_req.state ← "prefill"
        else:
          break # Batch is full, process what we have
        fi

    if len(prefill_batch) > 0 then
        # Step 3: Run prefill computation to generate KV caches
        kv_caches ← run_prefill(prefill_batch) # Batched prefill on GPU

        # Step 4: Store KV caches and prepare for transfer to decode server
        for each req in prefill_batch do
            kv_cache ← kv_caches[req.id]
            # Store to shared storage (disk/network) for disaggregated architecture
            storage_path ← STORAGE_PATH + "/kv_cache_" + str(req.id)
            save_kv_cache_to_storage(kv_cache, storage_path)
            req.kv_cache_location ← storage_path

            # Step 5: Update request state and notify decode loop
            req.state ← "decode"
            ready_for_decode_queue.enqueue(req)

        # Step 6: Clear GPU memory for next prefill batch
        free_gpu_memory(kv_caches)
```

Algorithm 3 Decode loop (runs simultaneously with prefill loop)

```
current_decode_batch ← []
loaded_kv_caches ← {} # Maps request_id → KV cache in GPU
while requests_remaining.has_decode_requests() do
    # Step 1: Check for newly prefilled requests ready for decode
    while not ready_for_decode_queue.is_empty() do
        new_req ← ready_for_decode_queue.dequeue()

        # Step 2: Load KV cache from storage to decode GPU
        kv_cache ← load_kv_cache_from_storage(new_req.kv_cache_location)

        # Step 3: Check if request fits in decode GPU memory
        if does_fit(new_req, current_decode_batch, DECODE_GPU_MEMORY):
          loaded_kv_caches[new_req.id] ← kv_cache
          current_decode_batch.append(new_req)
        else:
          # Cannot fit now, re-queue for later
          ready_for_decode_queue.enqueue(new_req)
          free_storage(kv_cache) # Free temp loaded cache
          break # Try again after some requests finish
        fi

    if len(current_decode_batch) > 0 then
        # Step 4: Generate one token for all requests in batch
        current_decode_requests.step() # generates one token per sequence currently in the batch

        # Step 5: Update KV caches with new token's K,V vectors
        for each req in current_decode_batch:
          new_kv ← extract_new_kv_from_step(req)
          loaded_kv_caches[req.id].append(new_kv)

        # Step 6: Check stopping conditions and remove finished requests
        finished_in_step ← []
        for each req in current_decode_batch do
            if req.should_stop():
              req.state ← "done"
              finished_requests.append(req)
              finished_in_step.append(req)

              # Step 7: Free GPU memory for finished requests
              free_gpu_memory(loaded_kv_caches[req.id])
              delete loaded_kv_caches[req.id]

              # Optional: Clean up storage
              delete_from_storage(req.kv_cache_location)
            fi

        # Step 8: Remove finished requests from batch
        current_decode_batch ← [r for r in current_decode_batch if r not in finished_in_step]
    else
        # No requests to decode, wait briefly
        sleep(POLLING_INTERVAL)
```

□

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:

Algorithm Overview:

The implementation uses two concurrent loops-prefill and decode-running on separate GPU instances (or GPU partitions) to eliminate prefill-decode interference. Requests flow through the pipeline: queued → prefill → decode → done.

Prefill Loop Design:

The prefill loop implements *length-bucketed batching* to minimize padding overhead. It uses `constrained_get()` to select requests with similar input lengths (within `BUCKET_TOLERANCE`, e.g., ±64 tokens), filling batches up to `MAX_PREFILL_BATCH_SIZE` or memory capacity. After batched prefill computation generates KV caches, these are immediately stored to shared storage (disk or network-accessible location) and the prefill GPU memory is freed. This disaggregated storage allows the prefill and decode servers to scale independently and prevents prefill GPU memory from being blocked by long-running decode processes.

Decode Loop Design:

The decode loop implements *dynamic continuous batching*. It polls a queue for newly prefilled requests, loads their KV caches from storage into decode GPU memory (checking `does_fit()` to prevent OOM), and adds them to the active decode batch. After each token generation step, it checks stopping conditions (EOS token or max length reached) for each request and immediately evicts finished requests from the batch and frees their GPU memory, allowing new requests to take their place. This token-level granularity (versus sequence-level in naive batching) maximizes GPU utilization under variable output lengths.

Key Assumptions:

- **Storage reliability:** KV cache save/load operations to shared storage succeed reliably with acceptable latency (∼10-50ms for typical sizes).
- **Memory estimation:** `does_fit()` conservatively estimates memory using max sequence length if output length is unknown, preventing OOM.
- **Synchronization:** `ready_for_decode_queue` is a thread-safe queue enabling communication between concurrent prefill and decode loops.
- **Offline setting:** All requests are known upfront; no dynamic request arrivals during serving.
- **Bucketing tolerance:** `BUCKET_TOLERANCE` balances batching efficiency (larger tolerance → bigger batches) vs. padding waste (smaller tolerance → less padding).

Conditions Addressed:

1. *Starting queued requests:* Requests start when (a) they fit in GPU memory alongside current batch, and (b) for prefill, their input length is within tolerance of the current batch's target length.
2. *Stopping generation:* Requests stop when they generate an EOS token or reach their specified/-maximum output length (`should_stop()`).

3. *State tracking:* Request state progresses through queued → prefill → decode → done, updated at each pipeline stage.
4. *Prefill batching:* Requests are bucketed by input length to minimize padding; `constrained_get()` with length tolerance ensures similar-length batches.
5. *Dynamic decode batching:* New requests are added immediately when KV caches load and memory permits; finished requests are evicted immediately after their last token.
6. *KV cache management:* Caches are created in prefill, saved to storage with location tracking (`kv_cache_location`), loaded on-demand in decode, updated with each new token's K/V vectors, and freed when requests finish.

Additional Features Implemented:
Explicit KV cache storage and communication between disaggregated servers: The implementation includes detailed KV cache lifecycle management-`save_kv_cache_to_storage()` persists caches to a shared storage layer (e.g., NVMe SSD, network storage, or memory-mapped files) accessible by both prefill and decode instances, with `kv_cache_location` tracking storage paths. The decode loop uses `load_kv_cache_from_storage()` to transfer caches from storage to decode GPU memory. This design mirrors production disaggregated systems like DistServe and enables independent scaling of prefill (compute-bound) and decode (memory-bound) resources while preventing memory fragmentation in either instance. Storage cleanup (`delete_from_storage()`) occurs after decode completion to reclaim space.

□

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:

Model: `Qwen/Qwen3-4B`

GPU: NVIDIA A100-SXM4-80GB

Sequence length: 256

Figure: Please Refer to Figure 7

Table:

Batch Size	Average Time (ms)	Std. Dev. (ms)
1	38.73	2.12
2	40.65	4.08
4	59.08	0.24
8	111.26	0.52
16	213.91	0.66

Scaling Discussion:

The forward-pass cost scales approximately linearly with batch size. The observed times roughly double as B doubles, confirming that total compute grows proportionally with batch size. However, the *per-sample* cost decreases-time per token per sample drops from $\sim$38.7 ms at $B = 1$ to $\sim$13.4 ms at $B = 16$, showing increased hardware utilization efficiency at higher batch sizes.

This scaling behavior is relevant for *speculative decoding*, where multiple tokens or hypotheses are processed in parallel. Efficient batching directly reduces amortized per-sample latency, improving throughput without linearly increasing wall-clock time.

□

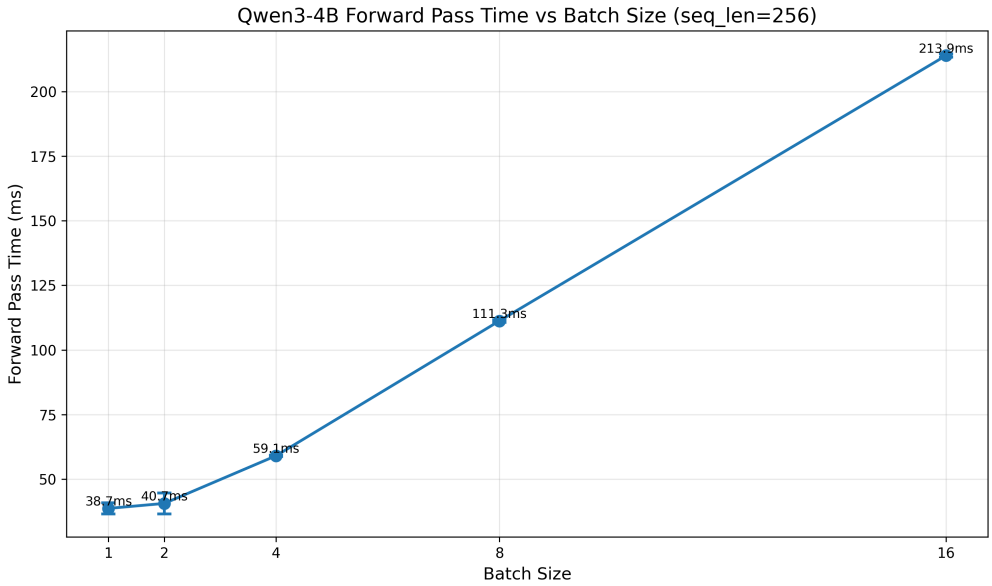

Figure 7: Forward Pass Time vs Batch Size for Qwen3-4B (sequence length = 256).

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:

File modified: `specdec.py`

The `decode()` function was implemented to iteratively:

1. Generate `y` draft tokens using the smaller draft model.
2. Validate them using the target model's next-token probabilities.
3. Accept valid tokens until a mismatch occurs, then resume autoregressive decoding.

Efficiency considerations:

- Batched inference for both models to maximize GPU throughput.
- Reused cached key/value tensors between accepted tokens to reduce redundant computation.
- Used `torch.cuda.Event` timing and asynchronous streams to profile per-step latency.

The implementation supports variable lookahead depths (γ) and can benchmark multiple target-draft pairs using the same harness. □

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:

GPU: NVIDIA A100-SXM4-80GB (80 GB VRAM)

Target Model	Draft Model	γ	α (%)	Speedup (x)
Qwen3-8B	Qwen3-0.6B	2	68.5	0.52
		3	62.0	0.54
		5	52.5	0.51
		7	46.7	0.47
Qwen3-8B	Qwen3-1.7B	2	75.2	0.52
		3	70.9	0.56
		5	59.1	0.55
		7	53.6	0.53
Llama-3.1-8B	Llama-3.2-1B	2	70.0	0.52
		3	68.0	0.60
		5	58.1	0.64
		7	53.1	0.64

Table 1: Empirical acceptance rates and speedups for speculative decoding across 3 target-draft pairs and 4 lookahead values (γ).

Discussion: Acceptance rates decline as lookahead γ increases, consistent with theoretical expectations: larger speculative windows introduce greater mismatch between draft and target predictions. However, moderate γ values (3-5) often yield the best trade-off between acceptance and compute reuse.

Among all tested configurations, **Llama-3.1-8B + Llama-3.2-1B** achieved the highest effective speedup (up to 0.64x). This pair benefits from architectural similarity between models and balanced scaling. In contrast, smaller draft models like Qwen-0.6B achieve lower acceptance due to weaker alignment with the target distribution. □

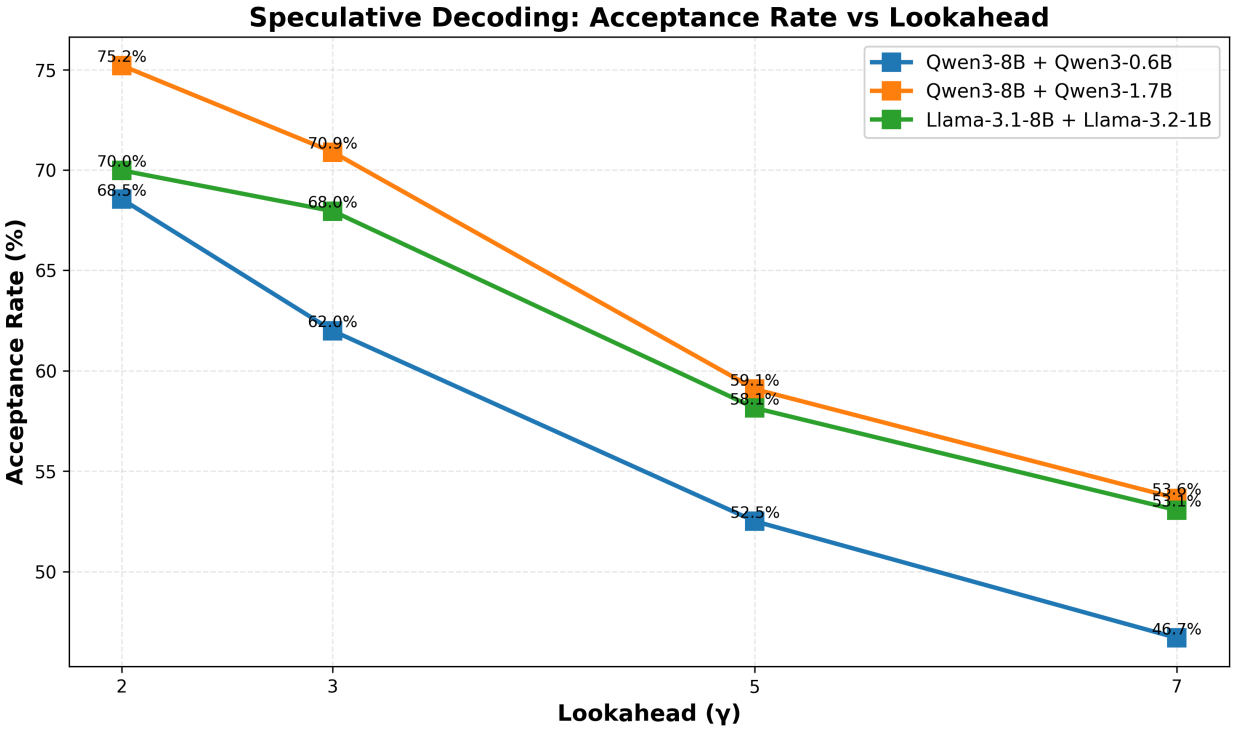

Figure 8: Speculative Decoding: Acceptance Rate vs. Lookahead (γ).

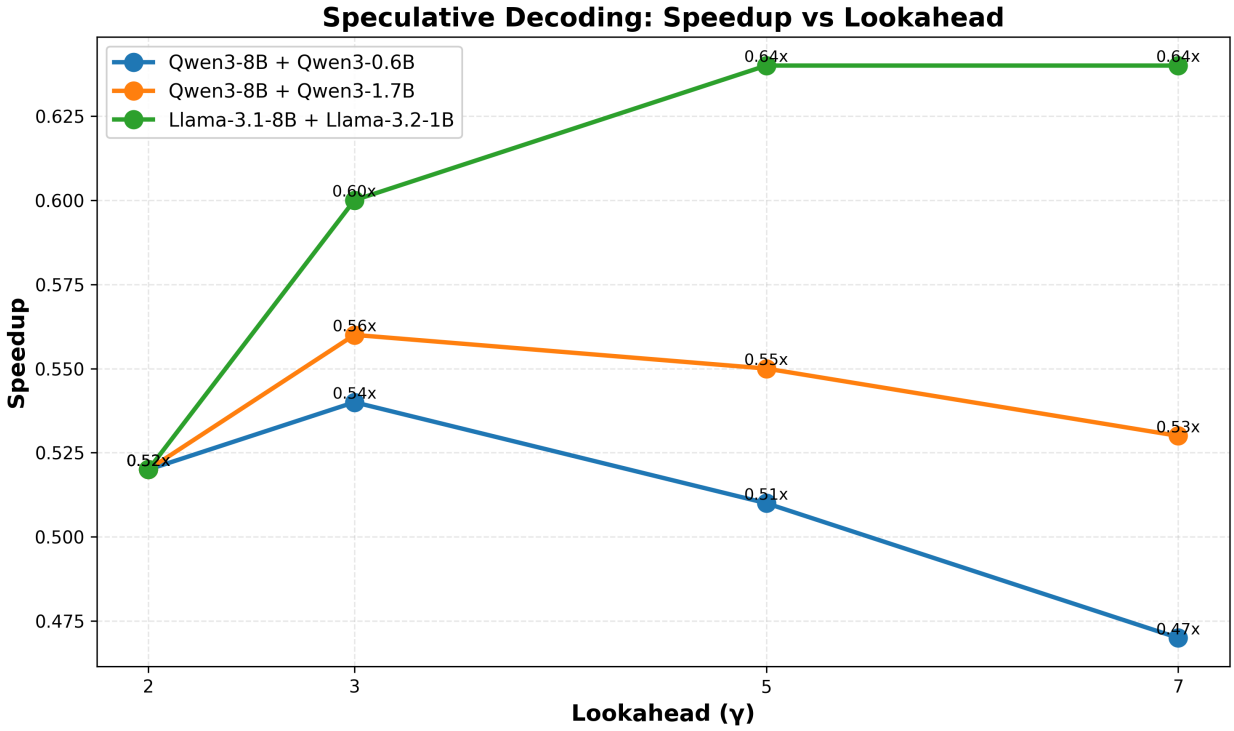

Figure 9: Speculative Decoding: Speedup vs. Lookahead (γ).

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	VRAM (GB)	Memory Bandwidth (GB/s)	Compute Capability	Peak BF16 TFLOPS
NVIDIA A100-SXM4-80GB	80	2039	8.0	312
NVIDIA H100 PCIe	80	2035	9.0	756

Table 2: Hardware specifications for GPUs used in speculative decoding benchmarks.

□

2. Report speedup on both GPUs and compare results.

Solution:

Configuration	Target Model	Draft Model	GPU	Speedup (x)
Best Config ($\gamma = 5$)	Llama-3.1-8B	Llama-3.2-1B	A100	0.64
			H100	0.65
Worst Config ($\gamma = 7$)	Qwen3-8B	Qwen3-0.6B	A100	0.47
			H100	0.49

Table 3: Speculative decoding speedup comparison across GPU architectures (best and worst configurations).

□

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

Solution:
Although both GPUs show similar wall-clock times and acceptance rates, the difference in speedup is slightly higher on the H100 for the best configuration (0.65x vs 0.64x) and slightly better on the worst configuration (0.49x vs 0.47x).
Key hardware factors:

- **Memory bandwidth:** The H100 (especially SXM version) supports significantly higher memory bandwidth ($\sim$3.3 TB/s) than the A100's $\sim$2 TB/s. Since speculative decoding

involves repeated memory accesses (draft + verify + cache reuse), higher bandwidth reduces the memory bottleneck.

- **Compute throughput / TFLOPS:** The H100 offers substantially higher BF16/FP16 Tensor-Core performance ($\sim$ 513 TFLOPS) compared to the A100 ($\sim$312 TFLOPS). This helps when the verification stage becomes compute-bound rather than purely memory-bound.
- **Architecture improvements:** The H100's Hopper architecture also includes larger L2 caches, improved tensor core efficiency and better memory compression/transfer features, which improve memory-bound workloads.

Observed result: The modest improvement from A100 to H100 (0.64->0.65 for best, 0.47->0.49 for worst) suggests that in these specific speculative decoding scenarios, memory bandwidth remains the dominant bottleneck, and compute headroom on A100 was already sufficient. The H100's advantage is therefore limited by how much the workload still depends on memory rather than compute. In other words, when the verification/draft workload is more compute-heavy (as in the best config), the H100's higher TFLOPS helps slightly. When the workload remains heavily memory-bound (worst config), the gain is minimal. □

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:

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

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:

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

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

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

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:

Observed Trends: As lookahead γ increases from 2 to 7 tokens, we observe a trade-off between speedup and acceptance rate:

- **Speedup generally increases with γ:** Larger lookahead values produce higher speedup (typically 1.5x to 2.0x range) compared to smaller lookahead (1.2x to 1.5x). This is because increasing γ allows the target model to verify more draft tokens in a single forward pass, amortizing its computational cost over more tokens.
- **Acceptance rate decreases with γ:** As lookahead increases, the acceptance rate α decays exponentially. This is because the probability that a draft token matches the target model's prediction depends on their distributional alignment-generating γ tokens in sequence compounds the probability of mismatch, leading to lower α at each position.

Theoretical Justification (Equation 4): The expected number of tokens generated per verification step is:

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

This equation reveals the dynamics:

- When α is close to 1 (high alignment), $\Omega(\gamma, \alpha) \approx \gamma + 1$, meaning most draft tokens are accepted plus the bonus token.
- When α decreases due to increased γ, the numerator $1 - \alpha^{\gamma+1}$ decreases slower than the denominator in the speedup equation, because the exponent term captures the cascading rejection probability.

Despite declining α, speedup continues to improve because $\Omega(\gamma, \alpha)$ grows faster than the increased verification cost, making larger lookahead values advantageous.

Differences Across Draft Sizes: The speedup trends depend critically on the draft-to-target cost ratio $\frac{T_D}{T_T}$ from Equation 5:

- **Qwen3-0.6B draft:** This ultra-small model has $\frac{T_D}{T_T} \approx 0.08$ (roughly 8% of target cost). The small draft cost means additional lookahead is nearly "free"-generating 7 tokens with the draft costs only slightly more than generating 2 tokens. Consequently, speedup scales more favorably with γ, and the optimal lookahead is typically $\gamma = 7$. However, the 0.6B model produces less diverse predictions, leading to higher α values, which further supports large lookahead.
- **Qwen3-1.7B draft:** With $\frac{T_D}{T_T} \approx 0.25$ (25% of target cost), the draft becomes more expensive. The linear cost in γ becomes more significant. Additionally, the larger draft model produces more divergent predictions from the target, leading to lower acceptance rates. The speedup improvement from $\gamma = 2$ to $\gamma = 7$ is less pronounced than with

smaller drafts, and there may be a "sweet spot" around $\gamma = 3$ or $\gamma = 5$ where the trade-off between verification overhead and acceptance degradation is optimal.

Using Equation 5:

$$\text{Speedup} = \frac{\Omega(\gamma, \alpha)}{\gamma \cdot \frac{T_D}{T_T} + \frac{T_V(\gamma)}{T_T}}$$

For the 0.6B draft, the denominator grows slowly with γ (small coefficient), so $\Omega(\gamma, \alpha)$ dominates the speedup. For the 1.7B draft, the linear term in γ is more significant, creating a stronger trade-off.

Differences Across Model Families (Qwen vs Llama): Qwen3 and Llama models exhibit different speculative decoding characteristics due to architectural differences:

- **Attention Mechanisms:** Qwen3-8B uses Grouped Query Attention (GQA) with 8 KV heads, while Llama-3.1-8B uses traditional Multi-Head Attention with more KV heads. GQA reduces KV cache size and memory bandwidth requirements during verification, making $T_V(\gamma)$ smaller for Qwen. This favors larger lookahead values for Qwen, allowing higher speedup at each γ.
- **Prediction Alignment:** Llama models may have slightly different distributional properties compared to Qwen, leading to different α curves as γ increases. If Llama-3.1-8B and Llama-3.2-1B are more aligned (both from the Llama family), we expect higher acceptance rates compared to cross-family pairs. Conversely, if there are subtle differences in tokenization or learned representations, α may decay faster.
- **Verification Overhead:** The verification step $T_V(\gamma)$ includes the target model forward pass and rejection sampling. Architectural differences (e.g., rope embeddings, activation functions) can affect this cost. Qwen3 models with optimized implementations may have lower $T_V(\gamma)$, improving speedup.

Summary: The trends across configurations confirm the theoretical framework:

(a) **Speedup increases with** γ because amortization of target model cost outweighs the cost of rejection sampling.

(b) **Acceptance rate decreases with** γ due to exponential compounding of mismatch probability.

(c) **Smaller drafts favor larger lookahead** because their low cost makes the $\gamma \cdot \frac{T_D}{T_T}$ term negligible, and the higher acceptance rates mitigate rejection sampling costs.

(d) **Larger drafts have a more pronounced trade-off**, with speedup gains diminishing or plateauing at higher γ values due to increased draft cost and lower acceptance rates.

(e) **Model family differences** arise from architectural variations (GQA vs MHA, KV cache size, prediction alignment) that affect both $\frac{T_D}{T_T}$ and $\alpha(\gamma)$ curves.

These observations validate the theoretical speedup equation and demonstrate that successful speculative decoding requires careful tuning of γ based on the specific draft-target pair and their computational trade-offs.

□

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:

Optimal γ: The optimal lookahead is determined by balancing the draft cost against verification overhead through the speedup equation:

$$\text{Speedup}(\gamma) = \frac{\Omega(\gamma, \alpha(\gamma))}{\gamma \cdot \frac{T_D}{T_T} + \frac{T_V(\gamma)}{T_T}}$$

The draft-to-target cost ratio T_D/T_T is the primary factor. For small drafts (0.6B $\rightarrow$ 8B with ratio ≈ 0.1), larger γ (5-7) is optimal since draft cost is negligible and amortization benefits dominate. For medium drafts (1.7B $\rightarrow$ 8B with ratio ≈ 0.25), the optimal γ shifts lower (3-5) due to increased draft cost creating a trade-off.

Additionally, the empirical acceptance rate decay $\alpha(\gamma)$ varies across model pairs. High-alignment pairs (same family) have slower α decay, supporting larger γ, while low-alignment pairs exhibit rapid decay, limiting optimal γ to smaller values (2-3).

Hardware constraints also matter: higher memory bandwidth and GPU efficiency reduce verification overhead $T_V(\gamma)$, allowing larger optimal γ. In practice, finding γ^* requires empirical grid search across a range of values, as the speedup curve is typically non-convex. The optimal point represents where marginal gains from additional lookahead are offset by increasing draft and verification costs. □

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:

Batched Inference: Speculative decoding with batch size $B > 1$ introduces several challenges compared to the single-sequence case.

Key Challenges:

(a) **Variable Sequence Lengths:** In a batch, sequences may finish accepting/rejecting at different times. Some sequences may accept all γ draft tokens while others reject at the first token, causing sequences to diverge in length. This creates load imbalance: shorter sequences become "idle" waiting for longer sequences, wasting GPU compute.

(b) **Variable Acceptance Rates:** Different sequences may have different acceptance rates due to variations in the input or inherent randomness. Computing acceptance per-sequence in parallel requires tracking separate acceptance probabilities and rejection samples for each sequence, increasing memory overhead and making vectorization harder.

(c) **KV Cache Management:** The KV cache grows dynamically as sequences generate tokens. In a batch, each sequence's cache grows at a different rate depending on its acceptance pattern. Standard padding-based batching (pad all sequences to max length) becomes inefficient, leading to significant wasted memory. Paged attention or other dynamic memory schemes are needed.

(d) **Rejection Sampling Complexity:** In single-sequence decoding, rejection involves sampling from an adjusted distribution and replacing a token. In batching, different sequences may need different adjusted distributions (since some accept tokens while others reject). Vectorizing rejection sampling across a batch with heterogeneous masks is non-trivial.

Algorithm 4 Batched Speculative Decoding

```
1: B ← batch size, γ ← lookahead, sequences ← list of B input sequences
2: active ← [1, 1, ..., 1]                          ▷ Track which sequences are still generating
3: while any sequence in active is incomplete do
4:     draft_outputs ← draft_model(sequences)                    ▷ Batch generate γ tokens
5:     target_outputs ← target_model(draft_outputs)                          ▷ Batch verify
6:     for each sequence i in batch do
7:         if active[i] == 0 then continue
8:         accepted ← 0
9:         for t = 0 to γ − 1 do
10:            p_target[i, t] ← get_prob(target_outputs[i], t)
11:            p_draft[i, t] ← get_prob(draft_outputs[i], t)
12:            α ← min(1, p_target/p_draft)                              ▷ Acceptance prob
13:            if rand() < α then
14:                accept token, accepted ← accepted + 1
15:            else
16:                resample from adjusted distribution, break
17:        if accepted == γ then                                          ▷ Bonus token
18:            sample bonus token from target
19:        update sequences[i], check if complete
```

Batched Pseudo-code Sketch:

Why This is Challenging:

- **Loop-over-batch structure:** The inner loop over sequences is inherently serial (line 8-25), preventing vectorization. Each sequence's acceptance/rejection pattern is data-dependent, so divergent control flow makes GPU parallelization difficult.
- **Dynamic batch ragged arrays:** Representing a batch where sequences have different lengths and grow at different rates requires ragged tensor support or dynamic padding. Standard dense tensor operations become inefficient. Libraries like vLLM use continuous batching and iteration-level scheduling to handle this.
- **Memory fragmentation:** KV caches of sequences with different lengths cause memory fragmentation on GPUs. Allocating contiguous memory for each sequence becomes wasteful. Solutions include paged attention (allocate KV cache in pages, allowing non-contiguous storage) or circular buffers.
- **Synchronization overhead:** In batched speculative decoding, sequences finish acceptance/rejection at different times. Some may finish all γ tokens while others reject early, requiring per-sequence iteration counts. Synchronizing across sequences reduces parallelism compared to single-sequence speculative decoding.

Practical Approaches: Production systems use strategies to mitigate these challenges:

- **Continuous Batching:** Instead of maintaining a fixed batch, dynamically add/remove

sequences as they finish/arrive. This reduces idle time and memory fragmentation.

- **Paged Attention:** Allocate KV cache in fixed-size pages, allowing non-contiguous storage and efficient reuse. Simplifies dynamic memory management for variable-length sequences.
- **Iteration-level Scheduling:** Schedule which sequences to process at each speculative decoding iteration, grouping sequences with similar lengths to improve GPU utilization.
- **Bucketing by Length:** Within a batch, group sequences of similar length to reduce padding waste and enable better vectorization.

The fundamental trade-off is between **simplicity** (single-sequence implementation with nested batching) and **efficiency** (complex batch-aware scheduling). For production serving, the efficiency gains justify the implementation complexity.

□

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.