Multi-GPU Load Balancing and Elastic Batching in LLM Inference: A Queuing Theoretic Perspective

Niloufar Saeidi Mobarakeh

Abstract

Large-scale large language model (LLM) inference presents critical challenges in queuing delay, batch processing, and GPU resource allocation, impacting latency-sensitive applications. Due to the extensive capabilities of Large Language Models, and the huge bottleneck of hardware because of the amount of computation, there has been a significant amount of work done in on the efficiency of these models. Some of this work has been on Queuing Theory. This project focuses of one of the recent works [1] that explores queuing theory-based optimizations for LLM inference, analyzing dynamic batching, multi-GPU load balancing, and elastic batching techniques. This paper tries to use Queuing Theory for the modelling of GPU processes in a Specific application, inference of large language models. With the growing interest in large language models, the inference level in LLMs is a huge bottleneck and there is growing interest to work on it. LLM inference is computation intensive and memory intensive, and improper parameter configuration at LLM platforms may exacerbate the inference time. This paper analyzes the impact of LLM output token distribution on the inference queueing delay, where the max-token clipping and the batched inference are considered. I try to egenerate the results of this paper for GPT2 and LLaMa-2 models, modelling GPU-based inference as an M/G/1 queue, and identifying queuing delays caused by heavy-tailed token length distributions.

I. Background

A. GPT-2: Generative Pre-trained Transformer

GPT-2 is a language model developed by OpenAI, based on the Transformer architecture. It was one of the first large-scale autoregressive models to demonstrate impressive text generation capabilities across a wide range of tasks without task-specific fine-tuning. GPT-2 is open-source, easy to run locally, and has moderate hardware requirements, making it ideal for controlled experiments in performance benchmarking and queuing theory.

In our study, I use the GPT-2 model (specifically the 124M parameter version) as a baseline to examine how inference latency scales with batch size and output token length. Due to its smaller size and simpler architecture compared to LLaMA-2, GPT-2 provides a useful contrast that helps isolate the effect of queuing and scheduling policies from model-specific architectural complexity.

Why GPT-2?

- Accessibility: Fully open-source and available through Hugging Face Transformers.
- Reproducibility: Well-documented behavior and established benchmarks.
- Scalability: Small enough to run multiple configurations on a single GPU node.

These characteristics make GPT-2 suitable for baseline experiments before scaling to more complex models like LLaMA-2.

B. LLaMA: Large Language Model Meta AI

LLaMA is a series of state-of-the-art transformer-based language models developed by Meta. It is designed to provide high-efficiency inference while maintaining competitive accuracy compared to models such as OpenAI's GPT series. The LLaMA models are particularly optimized for reduced compute costs, making them an attractive choice for academic research and real-world applications.

LLaMA-2, the second-generation model, introduces enhancements in architecture, pretraining methods, and fine-tuning strategies. The model supports different parameter sizes (7B, 13B, 65B) to balance compute requirements and performance. Unlike closed-source alternatives, LLaMA is available for academic and research purposes, making it an excellent candidate for experimentation in queuing theory applications. I submitted an access request to Meta-AI for the 7B model via huggingface and was granted the access after two days.

1) LLaMa Architecture

LLaMA inference consists of two main stages, which directly impact waiting times and GPU utilization:

- **Prefill Phase (Slower, More Computationally Expensive):** This phase processes the input prompt as a batch. All token representations are computed and cached in the attention mechanism. It is computationally expensive and significantly affects latency, particularly for long input sequences. Processing time is proportional to input length L_{in}. Uses Transformer self-attention, which scales as $O(L_{\text{in}}^2)$. If batch processing is used, all requests must wait for the longest prefill time in the batch. Service time:

$$S_{\text{prefill}} = \frac{L_{\text{in}}}{\mu} \tag{1}$$

  where μ is the GPU token processing speed (tokens/sec).

- **Decode Phase:** Once the first output token is generated, the model enters the decode phase. In this phase, LLaMA generates output tokens one at a time using cached key-value (KV) pairs. Only the most recent token needs new computation, while previous tokens use cached KV values. Each token generation depends on the previous token (sequential processing). The autoregressive nature of token generation makes inference latency dependent on output token length. So processing time is proportional to output token length L_{out}. Short responses are forced to wait for longer responses in batch processing. If output token lengths are highly variable (heavy-tailed), queues can grow unpredictably. Service Time:

$$S_{\text{decode}} = \frac{L_{\text{out}}}{\mu} \tag{2}$$

LLaMA inference introduces queuing challenges due to:

1) High variability in token lengths: Some requests have much longer output sequences.
2) Batching effects: Inference time for a batch is determined by the longest request.
3) Multi-GPU considerations: Requests may be distributed across multiple GPUs, requiring M/M/k queuing analysis.

Short requests may get stuck behind longer requests in the queue. Dynamic batching can optimize GPU usage but adds queuing complexity. Elastic batching (allowing early exits for shorter requests) could improve latency.

Batching is often used to optimize throughput, but it introduces queuing dynamics where the longest request in the batch determines the processing time. Recent research explores elastic batching strategies to reduce waiting time for shorter sequences while maximizing GPU utilization.

2) Token Length Distribution and Its Effect on Queuing

Studies on LLaMA inference show that input token lengths follow a normal distribution, while output token lengths exhibit heavy-tailed characteristics (e.g., log-normal or Weibull distributions). This variability in output token lengths significantly impacts queuing delay, making an M/G/1 queue model more appropriate than an M/M/1 model.

3) Comparison with Other Language Models

While LLaMA is optimized for efficiency, other language models exhibit different queuing behaviors:

- **GPT-3.5/4**: Closed-source models with high inference latency due to extensive self-attention calculations.
- **Mistral-7B**: More optimized than LLaMA for specific inference tasks, with improvements in batching strategies.
- **Falcon-40B**: High-performance model with significant memory requirements, making it less feasible for single-GPU queuing studies.

LLaMA's efficiency and accessibility make it a suitable candidate for our queuing theory experiments.

C. Compute Canada and Narval Cluster

Compute Canada is a national platform for high-performance computing (HPC) in Canada, supporting academic research through access to advanced computing infrastructure. The **Narval** cluster, hosted by Calcul Québec, is one of Compute Canada's flagship GPU-based HPC systems. It is equipped with NVIDIA A100 GPUs, high-speed interconnects, and large memory nodes, optimized for AI/ML workloads.

1) Narval Features

- **GPU Nodes:** Equipped with A100 GPUs suitable for large-scale language model inference.
- **SLURM Scheduler:** Efficient job array scheduling and resource management for parameter sweeps.
- **Shared Filesystem:** Enables coordinated logging, result aggregation, and code reuse across jobs.

2) Why Narval?

- **Free academic access:** Enables running dozens of concurrent jobs.
- **Scalable experiments:** Ideal for evaluating queuing policies across large batch/token configurations.
- **Low-level control:** Ability to isolate GPU memory, runtime, and queue parameters for detailed study.

All GPT-2 and LLaMA-2 experiments in this report were executed using job arrays on Narval, allowing us to collect statistically significant latency and scheduling metrics across multiple configurations.

II. MODELING AND METHODOLOGY

A. LLM Inference as an M/G/1 Queue

I model the inference server as an M/G/1 queue with a single GPU as the server. User requests arrive as a Poisson process with rate λ (requests per second), which is a standard assumption for independent user requests in networked services. Each request's service time S is non-exponential and depends on the number of output tokens L that the model generates, hence a general service distribution.

Large Language Models (LLMs) process inference requests on GPUs, leading to queuing delays. I model this using an **M/G/1** queue:

- **M (Markovian Arrivals)**: Requests arrive according to a Poisson process with rate λ.
- **G (General Service Time)**: Service time depends on token length and GPU speed.

- **1 (Single Server)**: A single GPU processes requests sequentially.

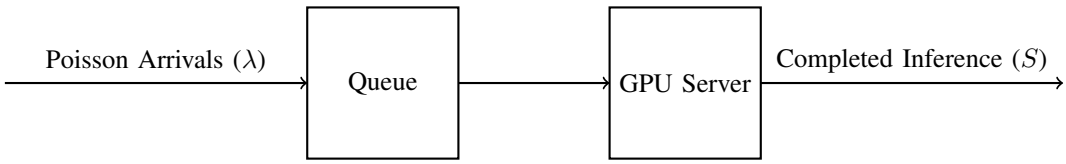


Fig. 1: M/G/1 Queue Model for LLM Inference

B. Arrival Process: Poisson Distribution

In practice, user queries to a language model inference system arrive at unpredictable times. This randomness is well-captured by a Poisson process, where the number of arrivals $N(t)$ in a time window t follows:

$$P(N(t) = k) = \frac{(\lambda t)^k e^{-\lambda t}}{k!} \tag{3}$$

In real-world inference systems, requests are often grouped into batches based on short, fixed-duration windows (e.g., 100 milliseconds). When arrivals follow a Poisson process with rate λ, the number of requests that arrive in each batching window of length t follows a Poisson distribution with parameter λt. Thus, the batch size itself becomes a Poisson-distributed random variable. In experiment B, we analyze how latency scales with increasing batch size, which approximates the queuing behavior in Poisson-driven batching systems.

The time between two consecutive arrivals, T_A, is exponentially distributed:

$$P(T_A \leq t) = 1 - e^{-\lambda t}, \quad \mathbb{E}[T_A] = \frac{1}{\lambda} \tag{4}$$

This Poisson-exponential arrival assumption is central to M/G/1 queuing models used to analyze queuing delays, batching strategies, and expected latency. It allows us to apply theoretical tools for:

- Evaluating trade-offs between batch size, service rate, and delay.
- Interpreting observed inference delays in Experiments A and B through stochastic models.

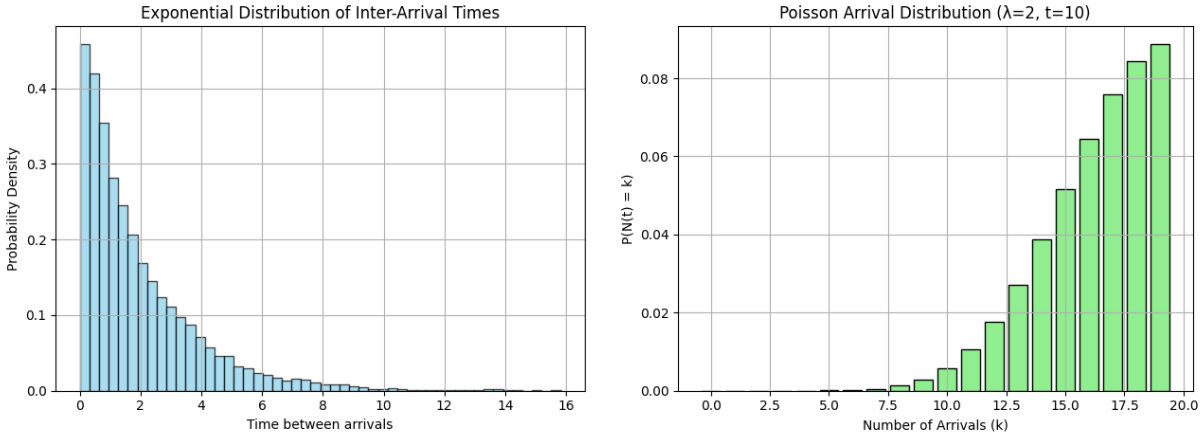


Fig. 2: (Left) Histogram of simulated inter-arrival times under a Poisson process; (Right) PMF of number of arrivals over time t. These are fundamental to the theoretical queuing delay and batching analysis discussed in this study.

C. Token-Length Dependent Service Time

The service time for each request is closely tied to its output token length. Specifically, the decode stage of autoregressive generation produces one token at a time, so longer outputs take more time to generate. In our model, I let L be the random variable for output token count. The service time S for a request is therefore a function of L (and potentially the input length), which I treat as a general distribution G. For instance, one simple model is $E[S|L] = c \cdot L + d$, where c is the average time per token and d is a fixed overhead per request. I will empirically validate this relationship for GPT-2 and LLaMA-2. Given the model's processing rate μ (tokens/sec), the service time for a request is:

$$S = \frac{L}{\mu} \tag{5}$$

For some more discussion see Figure 10 in the results section.
Empirically, output token lengths from LLMs follow a heavy-tailed distribution, such as the log-normal:

$$L \sim \text{Lognormal}(\mu_L, \sigma_L) \tag{6}$$

This introduces **high variability** in service times, which shifts our queuing model from M/M/1 to an M/G/1 queue. Such variability in L can be visualized and characterized by plotting histograms of output lengths. See Figure 8.
This formulation allows us to define the expected service time:

$$\mathbb{E}[S] = \frac{\mathbb{E}[L]}{\mu}, \quad \text{where} \quad \mathbb{E}[L] = e^{\mu_L + \frac{1}{2}\sigma_L^2} \tag{7}$$

and also the variance:

$$\text{Var}(S) = \frac{\text{Var}(L)}{\mu^2} \tag{8}$$

These quantities are key for analyzing queuing delay in M/G/1 models, especially when modeling tail-latency behavior or determining optimal batch sizes.

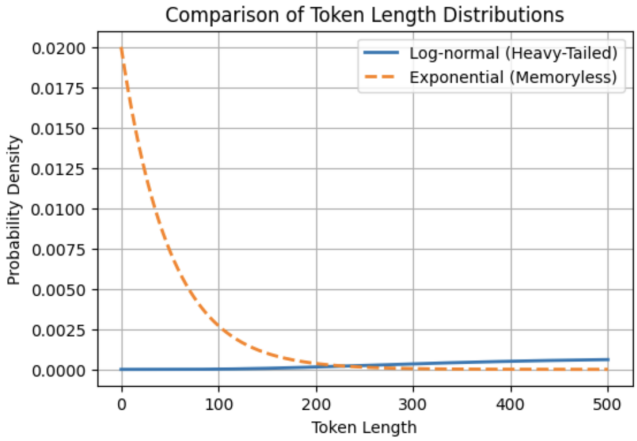


Fig. 3: Comparison of Token Length Distributions: Log-normal (heavy-tailed) vs Exponential (memoryless). LLM outputs tend to follow a log-normal distribution, which accounts for the long tail of high-token responses. This has significant implications for queueing delay due to the increased variance in service times.

The second moment of service time is needed for computing queuing delay:

$$E[S^2] = \frac{e^{2\mu_L + 2\sigma_L^2} - e^{2\mu_L + \sigma_L^2}}{\mu^2} \tag{9}$$

Queuing Delay Using Pollaczek–Khinchine Formula [2]: For an M/G/1 queue with Poisson arrivals and general service times:

$$W_q = \frac{\lambda E[S^2]}{2(1-\rho)} \tag{10}$$

where the traffic intensity ρ is:

$$\rho = \lambda E[S] \tag{11}$$

Substituting $E[S^2]$ yields:

$$W_q = \frac{\lambda \left(\frac{e^{2\mu_L + 2\sigma_L^2} - e^{2\mu_L + \sigma_L^2}}{\mu^2} \right)}{2(1 - \lambda E[S])} \tag{12}$$

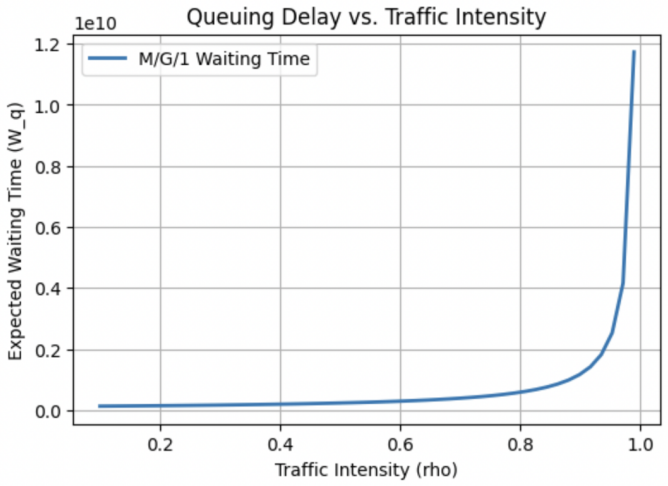


Fig. 4: Queuing Delay W_q as a function of traffic intensity ρ in an M/G/1 queue. As $\rho \to 1$, the delay increases sharply, indicating system saturation and the onset of instability.

The total response time is the sum of queuing delay and service time:

$$W = W_q + E[S] \tag{13}$$

Substituting prior terms:

$$W = \frac{\lambda\left(\frac{e^{2\mu_L+2\sigma_L^2}-e^{2\mu_L+\sigma_L^2}}{\mu^2}\right)}{2(1-\lambda E[S])} + \frac{e^{\mu_L+\frac{\sigma_L^2}{2}}}{\mu} \tag{14}$$

Given the above M/G/1 model, we can apply standard queueing theory results to quantify the latency. Let $\rho = \lambda E[S]$ be the server utilization. For stability we assume $\rho < 1$. Using the Pollaczek–Khinchine formula for an M/G/1 queue, the expected waiting time is

$$E[W] = \frac{\lambda E[S^2]}{2\big(1-\rho\big)}\,, \tag{15}$$

and therefore the expected total latency (waiting plus service) for a request is

$$E[T] = E[W] + E[S] = E[S] + \frac{\lambda E[S^2]}{2(1-\rho)}\,. \tag{16}$$

This derivation shows how service time variability (via token-length distribution) affects waiting time. If σ_L^2 is large, queueing delays increase significantly.

D. Some Discussion from the results of [1]

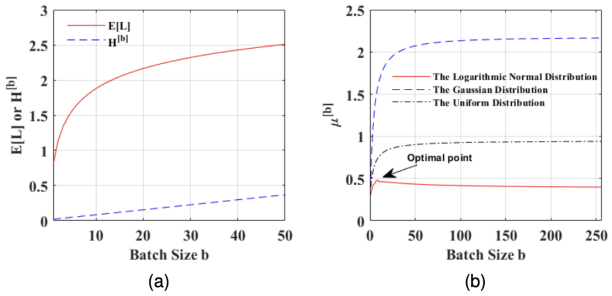


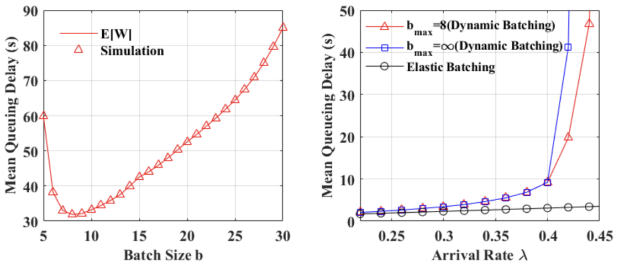
Fig. 3. The relationship between $E[L]$ (resp. $H^{[b]}$, $\mu^{[b]}$) and b.

Fig. 5: Figure 3 from [1] a) Expected maximum output length $\mathbb{E}[L]$ and logarithmic proxy $H^{[b]} = \log b$ as functions of batch size b. (b) Average latency per request $\mu^{[b]} = \mathbb{E}[L]/b$ for a log-normal output length distribution. As batch size increases, $\mu^{[b]}$ initially decreases due to amortized cost, but beyond a certain point it increases again due to longer tail samples. The optimal batch size balances these two effects, minimizing the average latency per request.

The figure above illustrates the relationship between expected maximum output length per batch and batch size under the assumption that output lengths follow a log-normal distribution. The expectation of the maximum L among b i.i.d. random variables is given by:

$$\mathbb{E}[L] = \int_0^\infty b f_{N_{\text{tr}}}(x) \left(F_{N_{\text{tr}}}(x)\right)^{b-1} dx \tag{17}$$

where $f_{N_{\text{tr}}}(x)$ and $F_{N_{\text{tr}}}(x)$ are the PDF and CDF of the token length distribution. I use a fitted log-normal distribution to model empirical output lengths. The latency per request $\mu^{[b]} = \mathbb{E}[L]/b$ captures the trade-off between batching efficiency and tail amplification.



(a) $\lambda = 0.43$ (b) Dynamic batching with max batch size and elastic batching

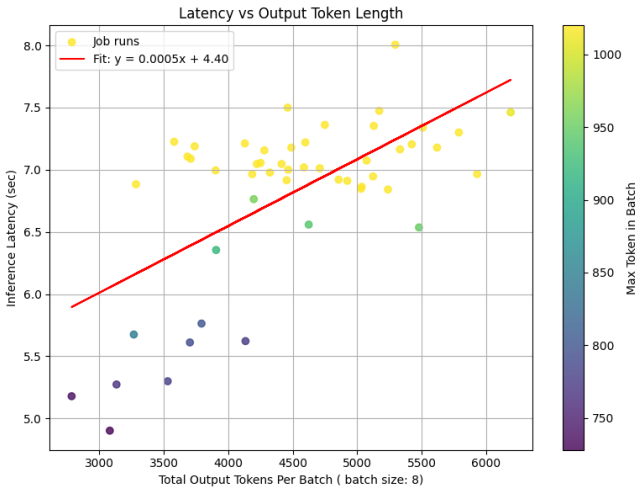
Fig. 6. The mean queueing delay of different batching inference for LLaMA-2-7b-chat on A100 GPU.

Fig. 6: Fig 6 from [1]; (a) Mean queuing delay $\mathbb{E}[W]$ as a function of batch size b for arrival rate $\lambda = 0.43$. The delay initially decreases due to batching amortization but increases beyond the optimal point due to longer wait times for batch formation. (b) Queuing delay under different batching policies as arrival rate λ increases. Elastic batching outperforms static dynamic batching by adaptively adjusting batch size to control latency under varying load. The divergence in delay for $\lambda > 0.4$ underlines the importance of dynamic latency-aware batching for high-throughput LLM inference.

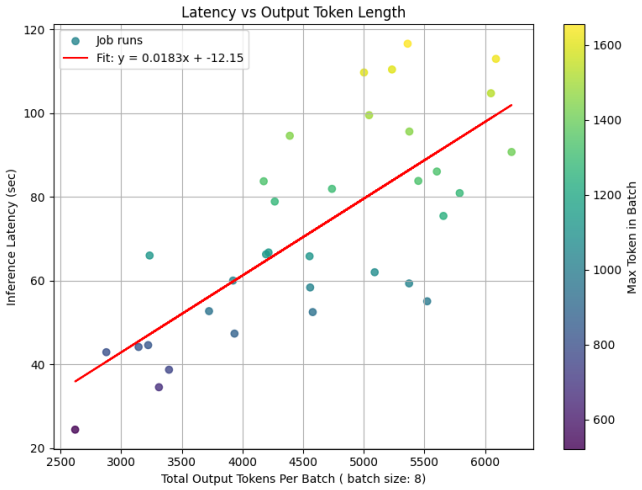
III. EXPERIMENTS AND RESULTS

A. *Experiment A: Impact of Token Length*

I fixed the batch size to 8 and varied the output token length. Using GPT-2-small [3] and LLaMA-2-7B-chat [4] (int8 quantized), I ran prompts designed to produce outputs ranging from 10 to 1000 tokens. I enforced exact lengths by appending a stop token at the desired length. For each length, I measured the average latency over 50 requests.



GPT-2



LLaMA-2

Fig. 7: **Latency vs. Total Output Tokens for GPT-2 and LLaMA-2 with Fixed Batch Size (8)**. Each point represents a batch of 8 requests with varying output lengths. The x-axis shows the total number of output tokens across all requests in the batch, and the y-axis shows the total inference latency. Points are colored by the *maximum output length* within each batch, which reflects the padding overhead due to the longest sequence. Since transformer models pad all sequences to the maximum length in a batch, even one long request can significantly increase total latency. This explains why jobs with similar total output tokens may exhibit different latencies depending on their internal padding inefficiency.

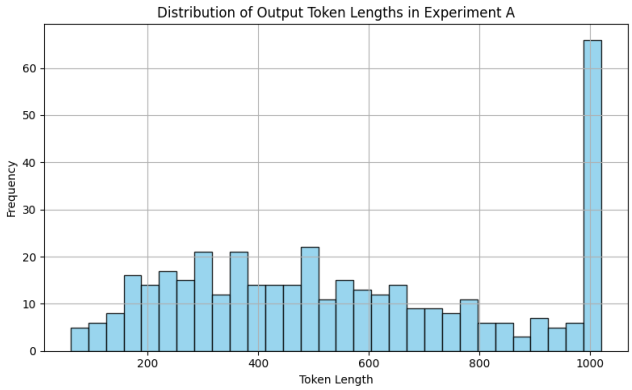


Fig. 8: Distribution of output token lengths in Experiment A from GPT-2. I trimmed the outputs at 1024 due to memory limitations. The long right tail reflects the log-normal nature, which introduces service time variability.

These heavy tails in token length increase the variance of the service time distribution, which directly increases queueing delay under the Pollaczek–Khinchine formula. Even in batched inference, the presence of a few high-length samples dominates latency due to padding.

B. Experiment B: Impact of Batch Size

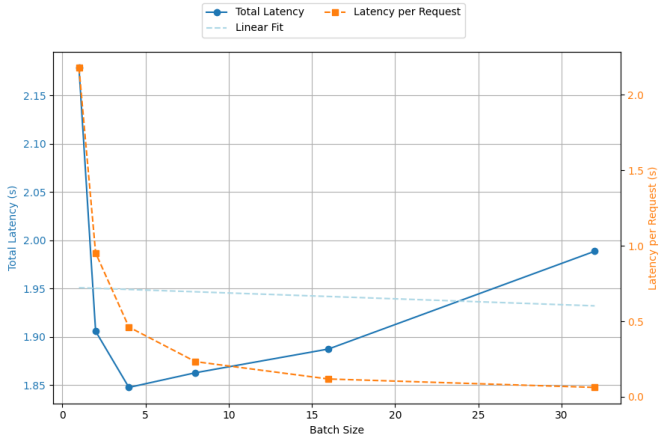
I fixed the output length to 256 tokens per request and varied the batch size. Batched inference was implemented using HuggingFace Transformers' 'generate()' with a specified batch dimension. Batch sizes tested were 1 (no batching), 2, 4, 8, 16, 32, and 64. I ensured the system was always busy (high arrival rate) so that a full batch of that size was available to process. Latency reported is per request (from dispatch to completion) averaged over 100 requests for each batch size.
In this experiment, I fix the output token length per request, $L_{\text{out}} = 256$, and vary the batch size $B \in \{1, 2, 4, 8, 16, 32, 64\}$. The goal is to investigate how batching affects the end-to-end inference latency on a single GPU, and to compare empirical results with theoretical predictions based on queuing models.
The inference process of a transformer model like GPT-2 can be modeled as a *bulk service queue*, where a batch of B requests are processed together. Let $T(B)$ denote the total latency required to process a batch of size B.
I decompose the total latency into two components:

- A fixed overhead t_0, which includes one-time setup costs such as kernel launch, attention mask construction, I/O operations, and memory allocation.
- A variable processing cost that depends on the output token length L and batch size B. Since transformer computation scales sublinearly with batch size due to GPU parallelism, I introduce a scaling exponent $\gamma \in (0, 1]$ to model imperfect parallelism.

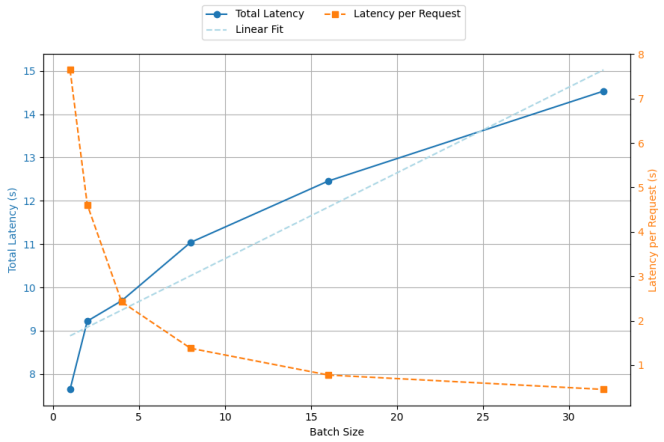
The total latency is then modeled as:

$$T(B) = t_0 + t_1 \cdot L \cdot B^{\gamma},$$

where t_1 is the per-token processing cost for a single request, and γ captures the efficiency of batching (with $\gamma = 1$ corresponding to linear scaling and $\gamma = 0$ to perfect parallelism).
This model reflects that increasing the batch size introduces more work overall, but also benefits from GPU parallelism. In practice, γ is typically estimated between 0.5 and 0.9 depending on hardware and model size.



GPT-2



LLaMA-2

Fig. 9: input size: 4 (”Once upon a time”), max_output_length: 256, total_tokens_per_request: 260

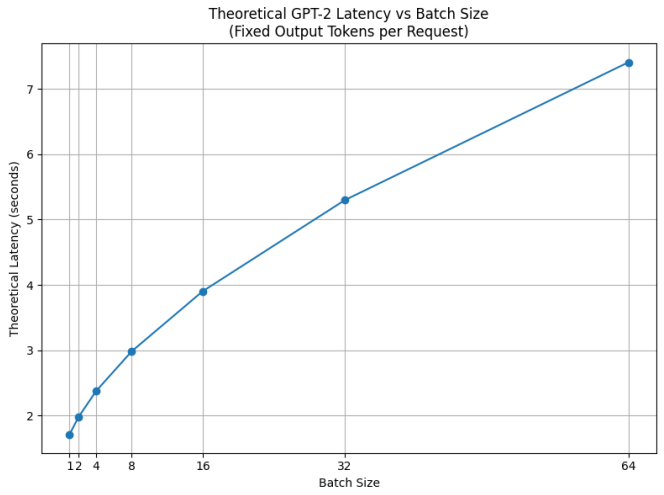


Fig. 10: Theoretical Latency Model: In synchronous batched inference, the total latency of a batch is dominated by the longest output sequence. Assuming a constant decoding throughput μ (tokens per second), the expected latency $\mathbb{E}[T]$ for a batch is given by: $\mathbb{E}[T] = \frac{L_{\max}}{\mu}$ where $L_{\max}$ denotes the maximum number of output tokens among all requests in the batch. This linear relationship highlights the performance impact of long output sequences in batch processing.

Note on Theory vs. Practice: While the theoretical model in Figure 10 captures the asymptotic latency behavior of batching,

our empirical results reveal important implementation effects. In particular, we observe:

- Higher latency at small batch sizes due to underutilization and runtime overheads.
- Slight increase or plateau at large batch sizes due to memory saturation or GPU contention.

This validates the general trend predicted by $\mathbb{E}[T] = \frac{L_{\max}}{\mu}$, but also highlights the need for empirical measurements to capture hardware-specific behavior. These observations align with findings from [5], which highlight significant latency variations across different hardware platforms and model configurations.

Comparison Between GPT-2 and LLaMA-2 Behavior

While both GPT-2 and LLaMA-2 exhibited the expected theoretical trends—namely, decreasing latency per request and increasing total latency with batch size—closer inspection reveals notable differences between the two models.

1. Total Latency Differences. LLaMA-2 consistently demonstrates significantly higher total latency compared to GPT-2 for the same batch sizes. This discrepancy arises due to LLaMA-2's deeper architecture, increased parameter count, and more intensive attention mechanisms. As batch size increases, LLaMA-2 incurs disproportionately higher latency costs, especially in the decode phase, where tokens are generated sequentially. In contrast, GPT-2, being shallower and more lightweight, scales more gracefully and maintains lower inference latency across batch sizes.

2. Batching Efficiency. The latency per request in GPT-2 drops sharply as batch size increases, which reflects effective amortization of overhead across multiple requests. For LLaMA-2, this amortization is less pronounced. Due to the model's greater memory footprint and internal complexity, batching efficiency is diminished. Larger batches may cause memory saturation, leading to reduced parallelism or slower computation due to GPU resource contention.

3. Implementation-Specific Factors. Architectural and implementation differences also contribute to this divergence:

- **KV Cache Overhead:** LLaMA-2 leverages more extensive key-value caching for attention layers, which improves token-wise throughput but increases memory usage and can bottleneck GPU performance under larger batches.
- **Prefill Costs:** The prefill phase (processing input prompt) is more expensive in LLaMA-2, as each token traverses a deeper stack of transformer layers.
- **Padding and Attention Masking:** Differences in padding behavior (e.g., max-length padding vs dynamic) and attention mask generation may lead to inefficiencies in LLaMA-2 for batches with uneven or long-tail sequence lengths.

4. Hardware Sensitivity.

These discrepancies are further magnified by hardware limitations. LLaMA-2 is designed and optimized for high-memory GPUs such as the A100. When run on less powerful GPUs, memory fragmentation, context switching overhead, or unoptimized tensor shapes can lead to degraded performance, especially under large batch sizes. GPT-2, being less demanding, performs more consistently even under limited resources.

Implications. These findings emphasize that while theoretical latency models (e.g., $T = \frac{L_{\max}}{\mu}$) provide general intuition, the empirical behavior of different LLMs can deviate due to architecture, padding behavior, and hardware constraints. Consequently, queuing-theoretic models must be validated separately for each model family.

IV. APPENDIX 1

A. SLURM: Simple Linux Utility for Resource Management

SLURM is an open-source workload manager and job scheduler used on most High-Performance Computing (HPC) clusters — like Narval, Cedar, Beluga, and others in the Compute Canada system. It is the middleman between the person and the cluster's compute resources. SLURM queues the job, finds an available node (machine), and runs it

B. Key Python Components of generate.py

Each job in my SLURM array will simulate one user inference request. That includes:

- Sampling a number of tokens to generate (based on a lognormal distribution)
- Simulating the time the request "arrived" and when it "got served"
- Using a real LLM (e.g., GPT2 or LLaMA-like model) to generate the tokens on GPU
- Logging key timing info: request start, GPU compute start, end, and total latency

This mimics a First-Come-First-Served (FCFS) GPU queue — since SLURM naturally queues the jobs in order and allocates them one-by-one as GPUs free up.

1) transformers Library: Provides pre-trained language models like gpt2-xl, LLaMA, falcon, etc. I'll use AutoTokenizer to encode prompts and AutoModelForCausalLM to generate text.
2) GPU Timing: I'll use time.time() to log:
    - When the job starts
    - When GPU compute starts (i.e., model.generate() is called)
    - When generation ends

3) Logging: Write a JSON log file per job: Includes token length, timestamps, delay, etc.
4) Argument Parsing: The script accepts command-line arguments:
   - $--output\_tokens$: Number of output tokens to generate
   - $--output\_dir$: Where to save logs

C. Using HugggingFace models

At a high level, Hugging Face provides a unified interface to load pre-trained deep learning models**, especially for NLP, like:

- GPT-2, GPT-J, GPT-NeoX
- LLaMA
- Falcon
- BERT
- T5

These models come with:

- A config file that defines architecture: layers, heads, etc.
- Pre-trained weights including big files: *.bin* or *.safetensors*
- A tokenizer that converts text to tokens and vice versa

Models are hosted on huggingface.co/models and downloaded automatically when you call from_pretrained().
explanation of the core code in generate.py script:

```
model_name = "gpt2-xl"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(model_name).cuda()
```

AutoTokenizer:

1) Downloads files like: tokenizer.json, vocab.json, merges.txt, *tokenizer_config.json*
2) Loads rules to tokenize text into subword tokens
3) Handles padding, truncation, and special tokens
4) Internally chooses the correct tokenizer class (GPT2Tokenizer)
5) example: tokens = tokenizer("What is the capital of France?", return_tensors="pt") Outputs a dictionary: {"input_ids": tensor([[15496, 318, 262, 1364, 286, 2637, 30]]),"attention_mask": tensor([[1, 1, 1, 1, 1, 1, 1]])}

AutoModelForCausalLM.from_pretrained("gpt2-xl"):

1) Downloads config.json to define architecture: hidden_size = 1600, n_layer = 48, etc.
2) Downloads model weights (e.g. pytorch_model.bin or model.safetensors): For GPT-2 XL, it's 6.43 GB of floating point weights
3) Loads the correct model class (like GPT2LMHeadModel)
4) Applies weights to the model structure
5) Places the model in eval() mode

How Inference Works in model.generate(), after loading, I use the model like this:

```
    output = model.generate(
    input_ids,
    max_new_tokens=output_len,
    do_sample=False,
    pad_token_id=tokenizer.eos_token_id
)
```

1) Input tokens are passed in the prompt
2) The model performs autoregressive decoding: predicts next token, appends it to input, and predicts the next one... up to max_new_tokens
3) For each step, the model uses: Self-attention, Past KV cache (to speed up later token generation), and Feed-forward layers

Each token generation becomes slower over time due to attention over longer sequence. That's why latency increases linearly with output length (as the Yang et al. paper models)

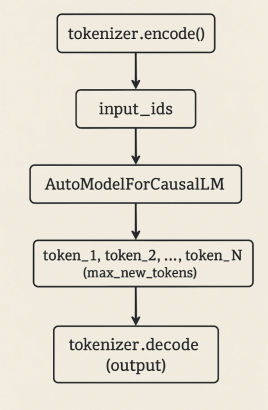


Fig. 11: Flow diagram of tokens

D. Model Access and Execution Environment

To simulate the latency characteristics of large language model (LLM) inference under variable output lengths, I used the `gpt2` model as a stand-in for larger LLMs such as LLaMA-2. While `gpt2` is significantly smaller in scale, it retains the essential architectural structure (Transformer-based autoregressive decoding) necessary for studying latency versus output token length relationships in queuing systems.

All experiments were conducted using resources provided by the **Narval cluster** hosted by the Digital Research Alliance of Canada. The compute environment includes access to A100 GPUs and high-memory compute nodes, managed via the SLURM workload manager.

E. Offline Model Access

Compute nodes on Narval do not allow outbound internet traffic. Therefore, models must be downloaded and cached manually on login nodes prior to job submission. I used the Hugging Face `transformers` and `huggingface_hub` libraries to handle model downloads and caching.

Download Step (Login Node):

```
export HF_HOME=/project/6006459/huggingface_cache

python -c "from transformers import AutoTokenizer, AutoModelForCausalLM; \
AutoTokenizer.from_pretrained('gpt2', cache_dir='$HF_HOME'); \
AutoModelForCausalLM.from_pretrained('gpt2', cache_dir='$HF_HOME')"
```

This command populates the shared project directory cache with the model files, which SLURM jobs can later access in offline mode. The cache directory used in all jobs was `/project/6006459/huggingface_cache`.

F. Script Design: `generate.py`

I developed a Python script, `generate.py`, to measure the end-to-end latency of inference calls under variable output token lengths. The script accepts the following arguments:

- `--job_id`: a unique job identifier (used for logging)
- `--model_name`: the Hugging Face model identifier (e.g., `gpt2`)
- `--output_tokens`: number of output tokens to generate (or `-1` to sample randomly)
- `--output_dir`: path to save output metrics per job

The script initializes the model and tokenizer in offline mode using the pre-cached directory:

```
cache_dir = os.environ.get("HF_HOME", "/project/6006459/huggingface_cache")

tokenizer = AutoTokenizer.from_pretrained(model_name,
    cache_dir=cache_dir,
    local_files_only=True)

model = AutoModelForCausalLM.from_pretrained(model_name,
    cache_dir=cache_dir,
    local_files_only=True)
```

It then performs inference on a fixed prompt (e.g., "Once upon a time") and logs the number of generated tokens and total latency (wall time) in a JSON file. If the argument --output_tokens is set to -1, the script samples a token length from a log-normal distribution to simulate natural variation observed in real-world LLM usage:

```
sampled_output_tokens = int(np.random.lognormal(
    mean=np.log(512), sigma=np.log(2)))
```

G. Job Submission: SLURM Script

Inference experiments were executed in parallel via SLURM job arrays. The job submission script, generate_job.slurm, launches 50 jobs with up to 4 running concurrently. Each task executes generate.py with a unique job ID and stores results in a shared logs/ directory.

SLURM Job Configuration:

```
#SBATCH --array=1-50%4
#SBATCH --gres=gpu:a100:1
#SBATCH --mem=48G
#SBATCH --cpus-per-task=4
#SBATCH --output=logs/out_%A_%a.log
#SBATCH --error=logs/err_%A_%a.log
```

Output logs and per-job results are stored under:

```
logs/job_<ID>.json
```

containing:

```
{
  "job_id": "12",
  "input_tokens": 5,
  "output_tokens": 576,
  "total_latency": 3.3314363956451416
}
```

These records form the empirical dataset for our queuing analysis. In the following, I have attached the scripts to download, submit and do the experiments and to analyze the results and generate the plots.

```
from transformers import AutoTokenizer, AutoModelForCausalLM
from huggingface_hub import login
import os

# Optional: Read token from environment variable
#hf_token = os.environ.get("HUGGINGFACE_TOKEN", None)

# Step 1: Authenticate (optional if already done via CLI)
#if hf_token:
#    login(token=hf_token)

# Step 2: Define model
#model_name = "meta-llama/Llama-2-7b-hf"
#cache_dir = os.path.expanduser("~/.cache/huggingface/transformers")

# Step 3: Download tokenizer and model to cache
#print(f"    Downloading tokenizer for {model_name}...")
#tokenizer = AutoTokenizer.from_pretrained(model_name, cache_dir=cache_dir, token=True)
#print("    Tokenizer downloaded and cached.")

#print(f"    Downloading model weights for {model_name}...")
#model = AutoModelForCausalLM.from_pretrained(model_name, cache_dir=cache_dir,token=True)
#print("    Model weights downloaded and cached.")
from transformers import AutoTokenizer, AutoModelForCausalLM
import torch
```

```
model_id = "meta-llama/Llama-2-7b-hf"

tokenizer = AutoTokenizer.from_pretrained(model_id)

model = AutoModelForCausalLM.from_pretrained(
    model_id,
    device_map="auto",
    #load_in_4bit=True,  # Or use load_in_8bit=True
    torch_dtype=torch.float16
)

#!/bin/bash
#SBATCH --job-name=gpt2-fized_batch
#SBATCH --output=llama_fixed_batch_logs/out_%A_%a.log
#SBATCH --error=llama_fixed_batch_logs/err_%A_%a.log
#SBATCH --array=1-50%4
#SBATCH --gres=gpu:1
#SBATCH --mem=60G
#SBATCH --cpus-per-task=4
#SBATCH --time=00:10:00


module load python/3.10
source ~/env/bin/activate
mkdir llama_fixed_batch_logs
# Replace with your actual path
#MODEL_PATH="/project/6006459/huggingface_cache/models--gpt2/snapshots/607a30d783dfa663caf39e0
MODEL_PATH="$HOME/.cache/huggingface/hub/models--meta-llama--Llama-2-7b-hf/snapshots/01c7f73d7
srun python generate_fixed_batch2.py \
    --job_id $SLURM_ARRAY_TASK_ID \
    --output_dir llama_fixed_batch_logs \
    --batch_size 8 \
    --model_path $MODEL_PATH

.

# generate_fixed_batch.py

import time
import os
import json
import argparse
import numpy as np
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM

def parse_args():
    parser = argparse.ArgumentParser()
    parser.add_argument("--job_id", type=str, required=True)
    parser.add_argument("--output_dir", type=str, default="logs")
    parser.add_argument("--batch_size", type=int, default=8)
    parser.add_argument("--model_path", type=str, required=True)
    parser.add_argument("--lognormal_mu", type=float, default=np.log(512))
    parser.add_argument("--lognormal_sigma", type=float, default=np.log(2))
    return parser.parse_args()

def main():
```

```
args = parse_args()
os.makedirs(args.output_dir, exist_ok=True)

# Load tokenizer and model
tokenizer = AutoTokenizer.from_pretrained(args.model_path, local_files_only=True)
model = AutoModelForCausalLM.from_pretrained(args.model_path, local_files_only=True).cuda
model.eval()

# GPT-2 has no pad_token by default, so assign EOS
tokenizer.pad_token = tokenizer.eos_token
model.config.pad_token_id = tokenizer.pad_token_id

prompt = "Once upon a time"

# Generate prompts for batch
batch_prompts = [prompt for _ in range(args.batch_size)]

# Tokenize with padding
encoded = tokenizer(batch_prompts, return_tensors="pt", padding=True, truncation=True)
input_ids = encoded["input_ids"].cuda()
attention_mask = encoded["attention_mask"].cuda()

input_length = input_ids.shape[1]

# Sample output lengths from log-normal
output_lengths = np.random.lognormal(
    mean=args.lognormal_mu,
    sigma=args.lognormal_sigma,
    size=args.batch_size
).astype(int)

# Clip so input + output does not exceed max context window
MAX_TOTAL_LENGTH = model.config.max_position_embeddings
max_allowed_output = MAX_TOTAL_LENGTH - input_length
output_lengths = np.clip(output_lengths, 1, max_allowed_output)
max_output_length = int(output_lengths.max())

print(f"[INFO] Input length: {input_length}, Max output: {max_output_length}")

# Inference
start = time.time()
with torch.no_grad():
    outputs = model.generate(
        input_ids=input_ids,
        attention_mask=attention_mask,
        max_new_tokens=max_output_length,
        do_sample=False,
        pad_token_id=tokenizer.pad_token_id
    )
end = time.time()
latency = round(end - start, 4)

# Save results
result = {
    "job_id": args.job_id,
    "batch_size": args.batch_size,
    "output_lengths": output_lengths.tolist(),
```

```
        "max_output_length": int(max_output_length),
        "total_latency": latency
    }

    with open(f"{args.output_dir}/job_{args.job_id}.json", "w") as f:
        json.dump(result, f, indent=2)

    print(f"[    ] Job {args.job_id} finished. Latency: {latency:.3f} sec")

if __name__ == "__main__":
    main()

#!/bin/bash
#SBATCH --job-name=llama2-batch1to128
#SBATCH --output=llama_logs/llama_batch_out_%A_%a.log
#SBATCH --error=llama_logs/llama_batch_err_%A_%a.log
#SBATCH --array=0-8%4        # 7 batch sizes: index 0 to 6
#SBATCH --gres=gpu:1
#SBATCH --mem=64G
#SBATCH --cpus-per-task=4
#SBATCH --time=00:10:00

module load python/3.10
source ~/env/bin/activate
export PYTORCH_CUDA_ALLOC_CONF=expandable_segments:True
MODEL_PATH="$HOME/.cache/huggingface/hub/models--meta-llama--Llama-2-7b-hf/snapshots/01c7f73d7
#BATCH_SIZES=(1 2 4 8 16 32 64)
BATCH_SIZES=(1 2 4 8 16 32 64 96 128)

BATCH_SIZE=${BATCH_SIZES[$SLURM_ARRAY_TASK_ID]}
mkdir -p llama_logs
echo "[INFO] Running on: $(hostname)"
echo "[INFO] Batch size: $BATCH_SIZE"
nvidia-smi

srun python llama_generate_fixed_token.py \
    --job_id "11B${BATCH_SIZE}" \
    --output_dir llama_logs \
    --batch_size $BATCH_SIZE \
    --model_path $MODEL_PATH


import time
import os
import json
import argparse
import numpy as np
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM

def parse_args():
    parser = argparse.ArgumentParser()
    parser.add_argument("--job_id", type=str, required=True)
    parser.add_argument("--output_dir", type=str, default="logs")
    parser.add_argument("--batch_size", type=int, default=8)
    parser.add_argument("--model_path", type=str, required=True)
    parser.add_argument("--lognormal_mu", type=float, default=np.log(512))
```

```
    parser.add_argument("--lognormal_sigma", type=float, default=np.log(2))
    return parser.parse_args()

def main():
    args = parse_args()
    os.makedirs(args.output_dir, exist_ok=True)

    # Load tokenizer and model
    tokenizer = AutoTokenizer.from_pretrained(args.model_path, local_files_only=True)
    model = AutoModelForCausalLM.from_pretrained(args.model_path, local_files_only=True).cuda
    model.eval()

    # GPT-2 has no pad_token by default, so assign EOS
    tokenizer.pad_token = tokenizer.eos_token
    model.config.pad_token_id = tokenizer.pad_token_id

    prompt = "Once upon a time"

    # Generate prompts for batch
    batch_prompts = [prompt for _ in range(args.batch_size)]

    # Tokenize with padding
    encoded = tokenizer(batch_prompts, return_tensors="pt", padding=True, truncation=True)
    input_ids = encoded["input_ids"].cuda()
    attention_mask = encoded["attention_mask"].cuda()

    input_length = input_ids.shape[1]

    # Sample output lengths from log-normal



    # output_lengths = np.random.lognormal(
     #    mean=args.lognormal_mu,
     #    sigma=args.lognormal_sigma,
     #    size=args.batch_size
    #).astype(int)
    output_lengths = np.full(args.batch_size, 256)

    # Clip so input + output does not exceed max context window
    MAX_TOTAL_LENGTH = model.config.max_position_embeddings
    # max_allowed_output = MAX_TOTAL_LENGTH - input_length
    max_allowed_output = 256
    output_lengths = np.clip(output_lengths, 1, max_allowed_output)
    max_output_length = int(output_lengths.max())

    print(f"[INFO] Input length: {input_length}, Max output: {max_output_length}")

    # Inference
    start = time.time()
    with torch.no_grad():
        outputs = model.generate(
            input_ids=input_ids,
            attention_mask=attention_mask,
            max_new_tokens=max_output_length,
            do_sample=False,
            pad_token_id=tokenizer.pad_token_id
```

```
        )
    end = time.time()
    latency = round(end - start, 4)

    # Save results
    result = {
        "job_id": args.job_id,
        "batch_size": args.batch_size,
        "output_lengths": output_lengths.tolist(),
        "max_output_length": int(max_output_length),
        "total_latency": latency,
        "input_length": int(input_length),
        "total_tokens_per_request": int(input_length + max_output_length)

    }

    with open(f"{args.output_dir}/job_{args.job_id}.json", "w") as f:
        json.dump(result, f, indent=2)

    print(f"[    ] Job {args.job_id} finished. Latency: {latency:.3f} sec")

if __name__ == "__main__":
    main()

import json
import os
import glob
import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import linregress
import pandas as pd

# Load and parse all LLaMA-related logs (assumes job_id starts with "L" or has "llama" in nam
def load_llama_data(log_dir="llama_logs"):
    all_files = glob.glob(os.path.join(log_dir, "job_11B*.json"))
    data = []
    for file in all_files:
        with open(file) as f:
            try:
                entry = json.load(f)
                job_id = entry.get("job_id", "")
                if job_id.startswith("11B"):
                    output_lengths = entry.get("output_lengths")
                    latency = entry.get("total_latency")
                    batch_size = entry.get("batch_size")

                    if not output_lengths or latency is None:
                        continue

                    data.append({
                        "job_id": job_id,
                        "total_output_tokens": sum(output_lengths),
                        "max_output_length": max(output_lengths),
                        "latency": latency,
                        "batch_size": batch_size,
                        "output_lengths": output_lengths
                    })
```

```
            except Exception as e:
                print(f"[       ] Failed to parse {file}: {e}")
    return data

# Plot total output tokens vs latency, color-coded by batch size
def total_latency_only_plot_llama_latency_vs_tokens(data, save_path="latency_vs_tokens_llama2
    data.sort(key=lambda x: x["batch_size"])
    x = [d["batch_size"] for d in data]
    y = [d["latency"] for d in data]
   # colors = [d["batch_size"] for d in data]

    slope, intercept, r_value, _, _ = linregress(x, y)
    line = [slope * xi + intercept for xi in x]

    plt.figure(figsize=(8, 6))
    scatter = plt.scatter(x, y, alpha=0.8, label="LLaMA-2 jobs")
  #  cbar = plt.colorbar(scatter)
  #  cbar.set_label("Batch Size")

    plt.plot(x, line, color="red", label=f"Theory Fit: y = {slope:.4f}x + {intercept:.2f}")
    plt.xlabel("Batch Size")
    plt.ylabel("Inference Latency (sec)")
    plt.title("LLaMA-2 Latency vs Batch Size")
    plt.grid(True)
    plt.legend()
    plt.tight_layout()
    plt.savefig(save_path)
    plt.show()

    print(f"   LLaMA-2 latency plot saved to {save_path}")
    return pd.DataFrame(data)
def plot_llama_latency_vs_tokens(data, save_path="latency_vs_tokens_llama2.png"):
    import matplotlib.pyplot as plt
    from scipy.stats import linregress

    data.sort(key=lambda x: x["batch_size"])
    batch_sizes = [d["batch_size"] for d in data]
    total_latencies = [d["latency"] for d in data]
    latency_per_request = [d["latency"] / d["batch_size"] for d in data]

    # Linear fit for total latency (optional)
    slope, intercept, r_value, _, _ = linregress(batch_sizes, total_latencies)
    latency_fit = [slope * b + intercept for b in batch_sizes]

    fig, ax1 = plt.subplots(figsize=(9, 6))

    # Left Y-axis: Total Latency
    color = 'tab:blue'
    ax1.set_xlabel("Batch Size")
    ax1.set_ylabel("Total Latency (s)", color=color)
    ax1.plot(batch_sizes, total_latencies, 'o-', color=color, label="Total Latency")
    ax1.plot(batch_sizes, latency_fit, '--', color='lightblue', label="Linear Fit")
    ax1.tick_params(axis='y', labelcolor=color)
    ax1.grid(True)

    # Right Y-axis: Latency per Request
    ax2 = ax1.twinx()
```

```
    color = 'tab:orange'
    ax2.set_ylabel("Latency per Request (s)", color=color)
    ax2.plot(batch_sizes, latency_per_request, 's--', color=color, label="Latency per Request"
    ax2.tick_params(axis='y', labelcolor=color)

    # Title and legends
   # fig.suptitle("LLaMA-2: Total vs. Per-Request Latency")
    fig.tight_layout()
    fig.subplots_adjust(top=0.88)
    fig.legend(loc="upper center", ncol=2)

    # Save + show
    plt.savefig(save_path)
    plt.show()
    print(f"   LLaMA-2 dual latency plot saved to {save_path}")

    return pd.DataFrame(data)

llama_data = load_llama_data("llama_logs")
if llama_data:
#    import ace_tools as tools; tools.display_dataframe_to_user(name="LLaMA-2 Log Data", dataf
    plot_llama_latency_vs_tokens(llama_data)
else:
    print("No LLaMA logs found in logs/ directory.")
```

V. APPENDIX 2

A. Avoiding Model Loading Errors in Offline Environments

During early experiments, I encountered persistent errors when launching inference jobs on SLURM compute nodes due to Hugging Face attempting to access the internet despite the use of `local_files_only=True`. The traceback included:

```
huggingface_hub.errors.LocalEntryNotFoundError: Cannot find the requested files
in the disk cache and outgoing traffic has been disabled.
```

Problem Summary

Even after downloading the model (e.g., `gpt2`) to a shared cache directory (`/project/6006459/huggingface_cache`), SLURM jobs failed because the model was still referenced by its name (e.g., `"gpt2"`) in the call to `from_pretrained()`.

Internally, this triggers an attempt to resolve metadata from Hugging Face's online hub — which is not possible on compute nodes.

Resolution: Use Explicit Snapshot Path

*** to find the name: /.cache/huggingface/hub/ then use the name to get the hash: ls /.cache/huggingface/hub/name/snapshots/ then sent model path to the address of that hash

The correct solution is to load the model and tokenizer using the full path to the downloaded snapshot directory.

To locate the snapshot hash:

```
ls /project/6006459/huggingface_cache/models--gpt2/snapshots/
```

This returns a unique hash, e.g.:

```
e7da7f68fa...
```

I then set:

```
model_path = "/project/6006459/huggingface_cache/models--gpt2/snapshots/e7da7f68fa..."
```

And load the model with:

```
tokenizer = AutoTokenizer.from_pretrained(model_path, local_files_only=True)
model = AutoModelForCausalLM.from_pretrained(model_path, local_files_only=True)
```

This guarantees that no online request is made, and SLURM jobs succeed in fully offline mode.

Notes

This issue is easy to reproduce when:

- You reference the model by ID (e.g., `"gpt2"`) instead of a path
- You do not pass a valid snapshot directory to `from_pretrained()`
- The job runs on a compute node with outbound traffic disabled

For all future experiments in Alliance clusters, prefer direct snapshot paths and confirm that the model loads correctly with `local_files_only=True` on the login node before submitting SLURM jobs.

B. Implementation Issues and Fixes: CUDA Indexing Error in Batched Inference

During the implementation of batched inference jobs using GPT-2 on the Narval cluster, several SLURM jobs failed with a CUDA error originating from the PyTorch ‘generate()‘ function during batched generation. Below I document the problem and the implemented solution.

C. Observed Error

The error message was:

```
/shared_tmp/.../Indexing.cu:1369: indexSelectSmallIndex: block: [x,0,0], thread: [y,0,0]
Assertion `srcIndex < srcSelectDimSize` failed.
```

This was accompanied by standard output logs indicating failures in batch generation, especially with larger batch sizes and random log-normal output token lengths.

D. Root Cause Analysis

The error originates from GPU kernel assertions in PyTorch’s CUDA backend. It occurs when the model’s ‘generate()‘ method accesses a token index beyond the expected input length, typically due to one or more of the following:

- Mismatched input lengths across batch entries.
- Absence of an attention mask.
- Missing `pad_token_id`, causing unexpected padding behavior.
- Output length exceeding the context window (e.g., ¿1024 tokens for GPT-2).

In our case, the combination of fixed-length input prompts and sampled output lengths (from a log-normal distribution) led to input + output token sequences exceeding the model’s maximum allowed sequence length.

E. Fixes Applied

The following fixes were applied in the ‘generate_fixed_batch.py‘ script:

1) **Defined padding token:**

```
tokenizer.pad_token = tokenizer.eos_token
model.config.pad_token_id = tokenizer.pad_token_id
```

2) **Enabled batch-level padding and masking:**

```
encoded = tokenizer(batch_prompts, return_tensors="pt", padding=True, truncation=True)
input_ids = encoded["input_ids"].cuda()
attention_mask = encoded["attention_mask"].cuda()
```

3) **Clipped sampled output lengths:** To ensure that the sum of input and output tokens stayed within GPT-2’s context limit of 1024 tokens:

```
output_lengths = np.clip(output_lengths, 1, max_allowed_output)
```

4) **Explicit use of attention mask during generation:**

```
outputs = model.generate(
    input_ids=input_ids,
    attention_mask=attention_mask,
    max_new_tokens=max_output_length,
    ...
)
```

F. Outcome and Relevance

After implementing these fixes, all SLURM jobs completed successfully across variable output lengths. This step was critical in generating valid latency measurements for batched inference, enabling accurate queuing-theoretic analysis in the following sections.

This experience also illustrates the importance of respecting architectural constraints (such as model context windows) when simulating production-scale batched inference, especially when stochastic output lengths are involved.

REFERENCES

[1] Y. Yang, Y. Xu, and L. Jiao, "A queueing theoretic perspective on low-latency llm inference with variable token length," 2024.

[2] J. Medhi, *Stochastic Models in Queueing Theory*. Academic Press, 2nd ed., 2002.

[3] H. Touvron, L. Martin, K. R. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[4] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," *OpenAI Blog*, 2019.

[5] K. T. Chitty-Venkata, S. Raskar, B. Kale, F. Ferdaus, A. Tanikanti, K. Raffenetti, V. Taylor, M. Emani, and V. Vishwanath, "Llm-inference-bench: Inference benchmarking of large language models on ai accelerators," *arXiv preprint arXiv:2411.00136*, 2024.