A PERFORMANCE MODEL

In this section, we analyze the trade-offs of various parallelism strategies by developing an analytical performance model. We break down the model's inference time (a forward pass) into several components and analyze how different forms of parallelism impact each one. The results demonstrate that the proportion of these components varies across different workloads, leading to distinct scaling behaviors for each parallelism strategy. Table 2 lists the used notations. We assume the data type is float16.

Table 2. Notations

N	number of output tokens	T_{dm}^{linear}	time of moving weights
r	number of requests	T_{dm}^{attn}	time of moving kvcache
b	(global) batch size	T_{c}^{linear}	time of computation
s	average sequence length	T_{c}^{attn}	computation of attention
h_q	number of heads	T_{nw}	time of communication
d	head dimension	h_{kv}	number of KV heads
L	number of layers	W	#parameters of one layer
PP	pipeline parallel degree	DP	data parallel degree
TP	tensor parallel degree		

A.1 Runtime Break-Down

The runtime of each decoding layer can be divided into three components: 1) data movement (T_{dm}) from GPU global memory (HBM) to compute units, which includes transferring weights (T_{dm}^{linear}) and KV cache (T_{dm}^{attn}), 2) computation T_{comp}, including T_{comp}^{linear} and T_{comp}^{attn}, and 3) communication cost T_{nw} (nw stands for network), primarily arising from the all-reduce operation in tensor parallelism. Based on the roofline model, the runtime of each layer can be approximated as $T_L = \max(T_{dm}^{\text{linear}}, T_{comp}^{\text{linear}}) + \max(T_{dm}^{\text{attn}}, T_{comp}^{\text{attn}}) + T_{nw}$.

Data Movement. The runtime of data movement can be approximated as transferred data volume divided by the bandwidth, which is the HBM bandwidth for GPUs. For linear layers, the transferred data is mostly weight matrices, of which the size is $2W$ bytes, which is constant. For attention layers, the transferred data primarily consists of the Q, K, V matrices, which is $2bs(h_q + 2h_{kv})d$ bytes in prefilling and $4bsh_{kv}d$ in decoding.

Compute. The computation time can be approximated as the number of floating operations (FLOPs) divided by the number of floating operations per second of the hardware (FLOP/s). For linear layers, the FLOPs is proportional to the weight parameters times the number of tokens, which is $2Wbs$ in prefilling and $2Wb$ in decoding. For attention layers, most operations come from computing the attention score, which is approximated as $bh_qs^2d^2$ in prefilling and $2bh_qsd^2$ in decoding.

Communication. The communication cost mostly comes from the all-reduce operation in tensor parallelism. It can be modeled as the transferred data volume divided by the bandwidth. We denote it as $T_{nw}(\text{TP})$, and approximate it as $b \cdot A/B_{ar}(\text{TP})$ where A is the size of the activation of one request within a batch and $B_{ar}(\text{TP})$ is the all-reduce bandwidth. $T_{nw}(\text{TP})$ is monotonically increasing with TP as additional GPUs and more replicas of activations are added to all-reduce. We omit peer-to-peer communication over in pipeline parallel since it is negligible compared to the all-reduce operation of tensor parallel.

Table 3. Different components of the runtime of a forward pass. The batch size b representing the batching effect is emphasized.

	T_{dm}^{linear}	T_{comp}^{linear}	T_{dm}^{attn}	T_{comp}^{attn}	$T_{nw}(TP)$
Prefill	$\frac{2W}{B_{\text{HBM}}}$	$\frac{2bWs}{\text{FLOPS}}$	$\frac{2bs(h_q+2h_{kv})d}{B_{\text{HBM}}}$	$\frac{bh_qs^2d^2}{\text{FLOPS}}$	$\frac{4bsh_qd}{B_{ar}(TP)}$
Decode	$\frac{2W}{B_{\text{HBM}}}$	$\frac{2bW}{\text{FLOPS}}$	$\frac{4bsh_{kv}d}{B_{\text{HBM}}}$	$\frac{2bh_qsd^2}{\text{FLOPS}}$	$\frac{4bh_qd}{B_{ar}(TP)}$

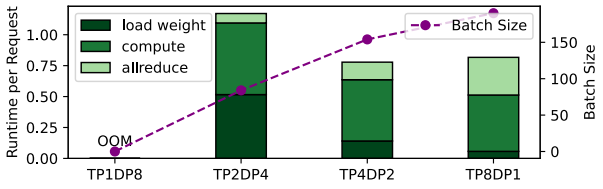


Figure 15. How data parallelism affects the decoding throughput. Data parallelism has minimal communication overhead but suffers from inefficient memory access caused by duplicating model weights. Model duplicates occupy more GPU memory, leaving less space for KV cache and smaller batch sizes. With more data parallelism, the overhead of loading data from GPU global memory to compute units significantly increases.

A.2 Batching Analysis

Batching is critical in decoding. It significantly affects the latency and throughput. Batch size represents how many requests are processed in one forward pass, and larger batch sizes can amortize the cost of transferring weights, thus improving the throughput.

Global and micro batch size. In distributed inference such, we define the *global batch size* b as the number of requests being actively processed by the whole cluster. It is a tunable hyper-parameter that represents the overall workload of the system. It is bounded by the memory budget and thus has a upper bound *maximal batch size*.

On the other side, the *micro batch size* is defined from the perspective of each device as the batch size of each forward pass. Tensor parallelism does not affect the micro batch size while DP and DP shrink the micro batch size.

A.3 Parallelism Analysis

We consider three types of parallelism: data parallelism, tensor parallelism, and pipeline parallelism, and denote their degree of parallelism as DP, TP, and PP respectively.

Tensor parallelism can accelerate both data moving (T_{dm}^{linear} and T_{dm}^{attn} are reduced to $1/\text{TP}$) and computation T_{comp} (reduced to T_{comp}/TP), at the cost of all reduce overhead T_{nw}.

Data parallelism distributes the global batch size b onto DP micro-batches processed in parallel. The model is duplicated so T_{dm}^{linear} remains unchanged. $T_{dw}^{\text{attn}}, T_{comp}^{\text{linear}}, T_{comp}^{\text{attn}}$, T_{nw} are reduced as the batch size is smaller. Due to the need to duplicate model weights, the GPU memory left for the KV cache is smaller. The spare space for KV cache on each GPU is $M_{kv} = M - \frac{2LW}{TP \cdot PP}$. The maximal batch size is

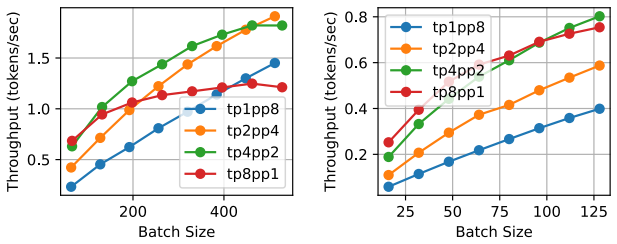
$$b_{\max} = \text{DP} \cdot \frac{M_{kv} \cdot \text{TP} \cdot \text{PP}}{4Lh_{kv}ds} = \text{DP} \cdot \frac{M \cdot \text{TP} \cdot \text{PP} - 2LW}{4Lh_{kv}ds}$$

While TP and PP can super-linearly scale the batch size, DP can only linearly scale the batch size. The trade-off between limited batch sizes and reduced communication overhead is shown in Figure 15.

Pipeline parallelism distributes different layers to different devices, and each device will have L/PP layers. It cannot reduce single-request latency but is more suitable for throughput-oriented scenarios as it introduces less communication overhead. However, it is not the ultimate answer of high-throughput applications because of an important observation that *pipeline parallelism harms maximal batch size.* A tricky nuance is that given a batch size b, pipeline parallelism can only process b/PP of them simultaneously in order to utilize and pipeline all PP GPUs, which is harmful to batching. If the workload is not uniformly distributed across GPUs, there will be bubbles, or in the worst case, some GPUs might be idle. When the pipeline is fully and stably pipelining, each time the last pipeline stage finishes its L/PP layers of forward pass, a micro-batch of b/PP will be finished.

Throughput. The micro-batch size on each GPU is $b/(\text{PP} \cdot \text{DP})$. The total runtime of generating one micro batch with size $b/(\text{PP} \cdot \text{DP})$ on one DP replica (or more specifically, the time of the last pipeline stage finishing a micro-batch) is

$$T_{\text{stage}} = \frac{L}{\text{PP}} \cdot \left[\max\left(\frac{T_{dm}^{\text{linear}}}{\text{TP}}, \frac{T_{comp}^{\text{linear}}}{\text{DP} \cdot \text{TP} \cdot \text{PP}}\right) + \frac{\max(T_{dm}^{\text{attn}}, T_{comp}^{\text{attn}})}{\text{DP} \cdot \text{TP} \cdot \text{PP}} + \frac{T_{nw}(\text{TP})}{\text{PP} \cdot \text{DP}} \right]$$



(a) prompt length=1000 (b) prompt length=4000

Figure 16. Decoding throughput in different scenarios. The optimal parallelism strategy depends on both hardware specifications and workloads. Some observations can be drawn such as 1) TP is efficient with smaller batch sizes, but degenerates with growing batch sizes; 2) The degeneration of TP depends on the ratio between GPU performance and network bandwidth; 3) As the batch size is bounded by the GPU memory and cannot scale infinitely, TP can achieve higher throughput in many cases.

The throughput (number of processed requests per unit time) is $b/\text{PP}/T$. For simplicity, we calculate the inverse of it as

$$\text{throughput}^{-1} = \frac{T_{\text{stage}}}{b/\text{PP}} = \frac{L}{b} \cdot \left[\max\left(\frac{T_{dm}^{\text{linear}}}{\text{TP}}, \frac{T_{comp}^{\text{linear}}}{\text{DP} \cdot \text{TP} \cdot \text{PP}}\right) + \frac{\max(T_{dm}^{\text{attn}}, T_{comp}^{\text{attn}})}{\text{DP} \cdot \text{TP} \cdot \text{PP}} + \frac{T_{nw}(\text{TP})}{\text{PP} \cdot \text{DP}} \right] \quad (1)$$

If we approximate the roof-line model with a simplified additional model, this expression can be simplified as:

$$\text{throughput}^{-1} \propto \frac{T_{dm}^{\text{linear}}}{\text{TP}} + \frac{T_{comp}^{\text{linear}} + T_{dm}^{\text{attn}} + T_{comp}^{\text{attn}}}{\text{DP} \cdot \text{TP} \cdot \text{PP}} + \frac{T_{nw}(\text{TP})}{\text{PP} \cdot \text{DP}} \quad (2)$$

B Artifact Appendix

B.1 Abstract

The artifact includes an example where the LLM inference engine transitions between two parallelization strategies (PP=4 for prefilling and TP=2 for decoding, respectively). We provide a Dockerfile for building the Docker image used in the experiments. The experiments require a multi-GPU machine with a CUDA-enabled environment. We include an example configuration that has been tested on a 2×L4 instance on GPU. Once the Docker image is built, a benchmark script can be executed within it to demonstrate the artifact's functionality.

B.2 Artifact check-list (meta-information)

- **Program:** Python library called `cgen`.
- **Compilation:** DockerFile
- **Data set:** ShareGPT, arxiv-summarization

- **Run-time environment:** Ubuntu 22.04 + CUDA 12.1 + PyTorch 2.4.0, built as a docker image
- **Hardware:** Single-node-multi-GPU machine. Tested on GCP `g2-standard-24` with two L4 GPUs.
- **Experiments:** Transition between TP=2 and PP=2
- **How much disk space required (approximately)?:** 64 GiB
- **How much time is needed to prepare workflow (approximately)?:** 30 min
- **How much time is needed to complete experiments (approximately)?:** 15 min
- **Publicly available?:** Yes
- **Archived (provide DOI)?:** 10.5281/zenodo.14991055

B.3 Description

More information can be found in the `readme.md` file.

B.3.1 How delivered

This artifact can be found on Zenodo: https://zenodo.org/records/14991055

B.3.2 Hardware dependencies

The example configuration has been tested on a Google Cloud Platform (GCP) `g2-standard-24` instance with two L4 GPUs (24 GiB) and 96 GiB RAM (64 GiB is required). For smaller hardware configurations, the configuration file needs to be modified to fit the hardware.

B.3.3 Software dependencies

The dependencies are included in the Dockerfile and the Python package, which will be automatically installed. A docker environment and NVIDIA container toolkits need to be set up. More specifically, this artifact depends on

- Ubuntu 22.04
- CUDA 12.1
- PyTorch 2.4.0
- vLLM 0.5.5

B.3.4 Data sets

We include two different datasets for benchmarking, namely ShareGPT and arxiv-summarization. We provide a script to download ShareGPT, which is done during building the docker image. The arxiv-summarization dataset can be downloaded directly through the Huggingface `datasets` library.

B.4 Installation

First, install docker and NVIDIA Container Toolkit. Then, build the docker image from `Dockerfile` under the `cgen` directory.

```
$ docker build -t cgen:latest .
```

If you are using GPU architectures other than the default one (`8.9+PTX`), a building argument can be used to assign this value:

```
$ docker build -t cgen:latest . \
  --build-arg TORCH_CUDA_ARCH_LIST="8.9+PTX"
```

B.5 Experiment workflow

Before the experiment, ensure the access to the Huggingface model `meta-llama/Llama-2-7b-hf`. Then save your HuggingFace token as an environment variable `HF_TOKEN`:

```
$ export HF_TOKEN=<your hf_token>
```

We provide a script to launch the experiment through Docker:

```
$ bash benchmark/run_benchmark.sh
```

Use `sudo` to run docker if necessary.

B.6 Evaluation and expected result

An offline text generation task is launched. You can observe from the log that the system is transitioning between two phases, and the change of the number of sequences in CPUs and GPUs.

B.7 Experiment customization

More description of customizable parameters can be found in the `readme.md` file or the codebase.