Text-generation inference

Nicolas Patry - 20/03/2023

Overview

- Optimizing Bloom
  - TP vs PP
  - Transformers
- Optimizing the webserver
  - Latency vs Throughput
  - Minimizing latency AND maximizing throughput.
  - Generation with PastKeyValues
  - Testing

What is bloom ?

- Very large language model (LLM), of 170 billion parameters, on par with GPT-3
- Trained on many languages (13)
- Open sourced (including training, datasets and many other artifacts)
- Requires 340 GB of VRAM (8A100 80, or 16A100 40)

An inference server ?

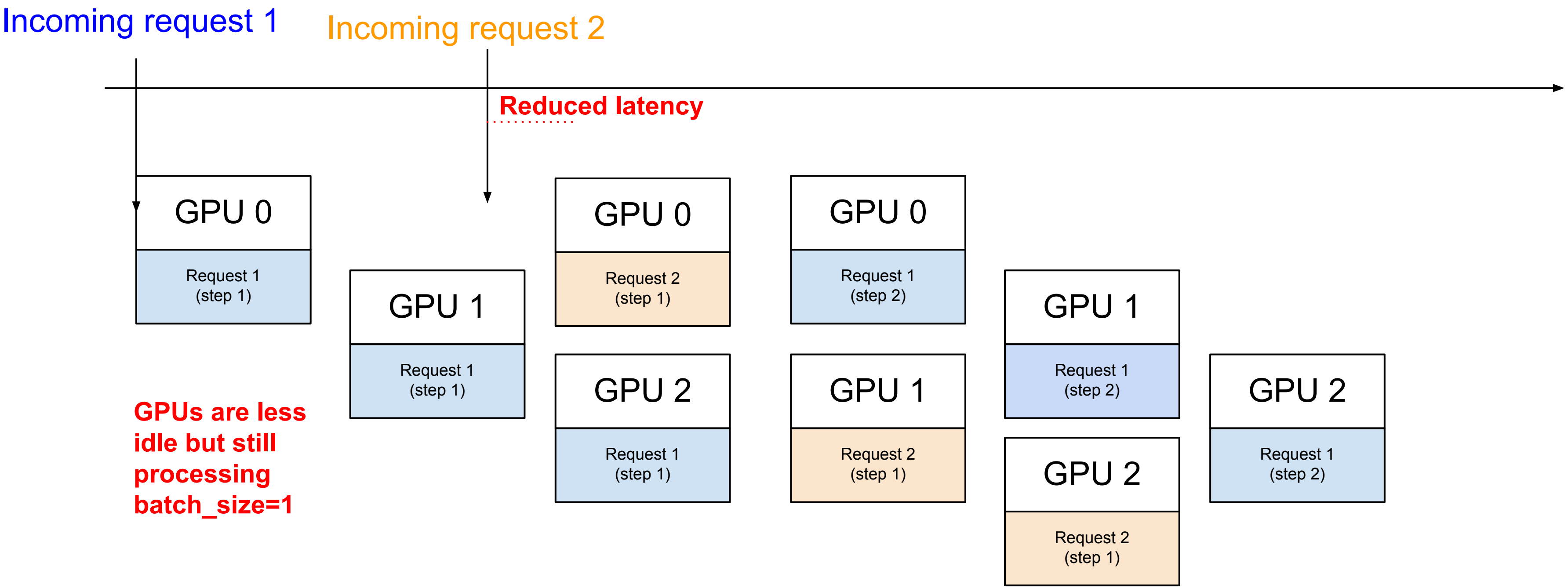
- Enable users to try out bloom without any hardware with an API
- Optimize for user perceived latency
- Give the best possible showcase for the model

Optimizing Bloom

Bloom Optimization

- Start from a CORRECT program version
- Think early about interactions between core optimizations:
  - Past key values
  - TP vs PP
  - Webserver orchestration
- Think about your endgoal first. Latency vs throughput, and targets
- Back of the envelope calculations are critical. How far are you from optimal code on given hardware.
  - Can it be done ? And how hard will it be ?
- Measure performance
- Identify bottlenecks
- Make bottlenecks faster
- Measure performance, and check it worked

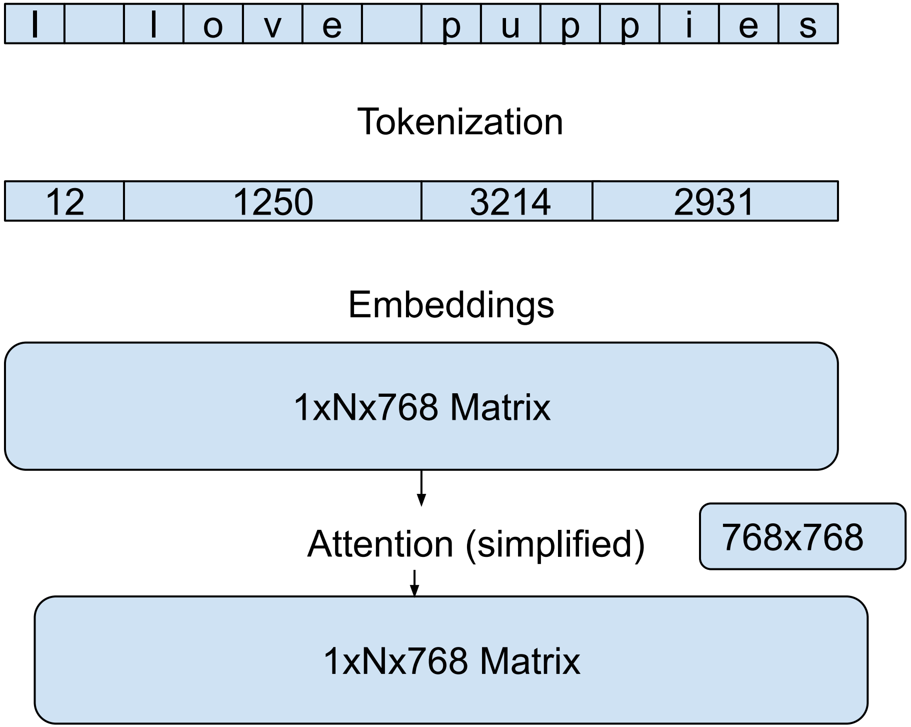
TP vs PP



Reduced latency

GPUs are less idle but still processing batch_size=1

Past key values



I		l	o	v	e		p	u	p	p	i	e	s

Tokenization

12	1250	3214	2931

Embeddings

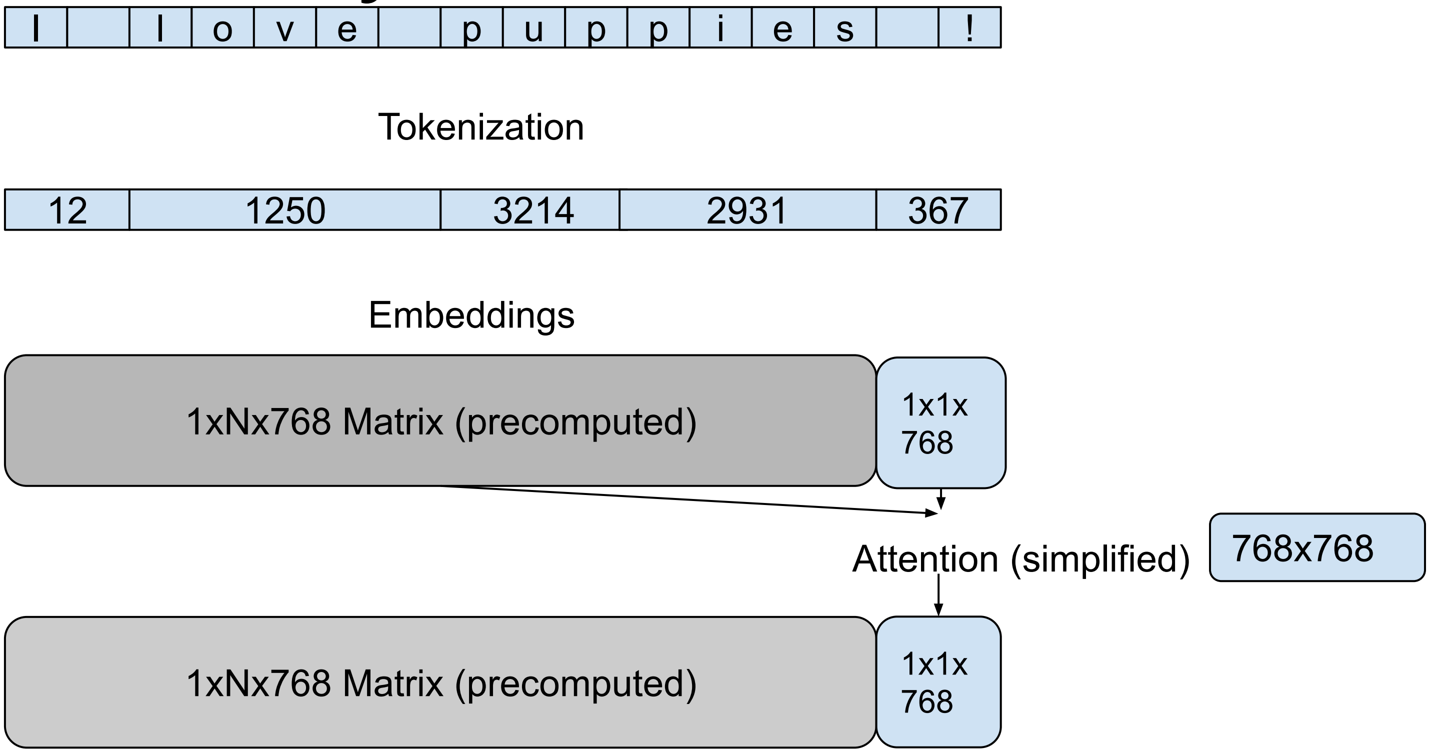
1xNx768 Matrix

Attention (simplified)

768x768

1xNx768 Matrix

Past key values



I		l	o	v	e		p	u	p	p	i	e	s		!

Tokenization

12	1250	3214	2931	367

Embeddings

1xNx768 Matrix (precomputed)

1x1x
768

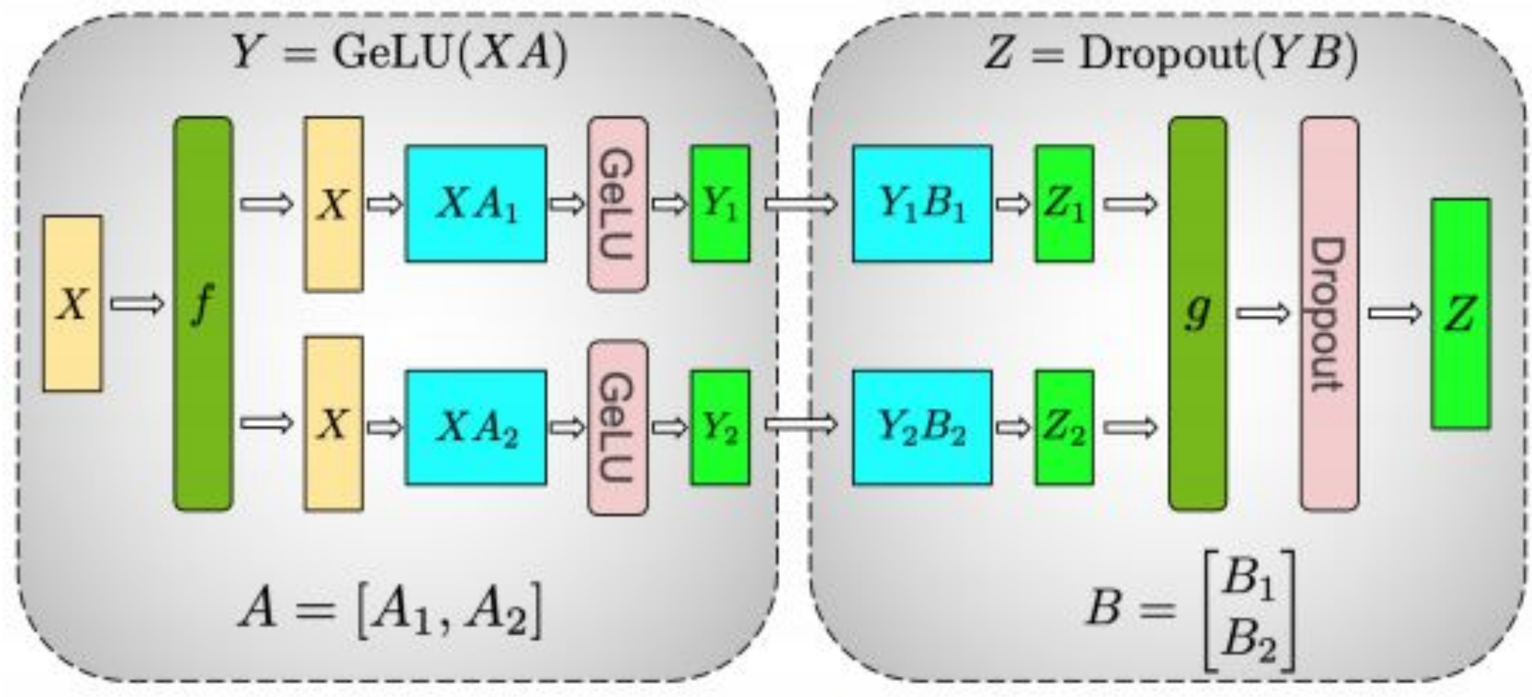
Attention (simplified)

768x768

1xNx768 Matrix (precomputed)

1x1x
768

TP vs PP



$Y = \mathrm{GeLU}(XA)$
X
f
X
XA_1
GeLU
Y_1
X
XA_2
GeLU
Y_2
$A = [A_1, A_2]$
$Z = \mathrm{Dropout}(YB)$
Y_1B_1
Z_1
Y_2B_2
Z_2
g
Dropout
Z
$B = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}$

(a) MLP

TP vs PP

PP

- Simple modeling
- Little GPU communication
- Good for throughput

TP

- More complex modeling (transformers is quite easy)
- Lots of inter GPU communication (new bottleneck)
- Good for latency

Transformers optimization

- Start by profiling:
  https://github.com/pytorch/kineto/blob/main/tb_plugin/README.md

- Usually start by removing "stupid ops":
  - Remove unnecessary reshapes, tensor creations
  - Fuse multiple small operations (@torch.jit.script)
  - Aim for maximum GPU utilization, maximum tensor cores utilization

- Find the new bottleneck (usually attention or MLP in transformers)
  - Create new kernels

Transformers optimization

- PP to TP:
  Latency 350ms/token -> 100ms/tokens (16A100 40Go)
- Remove extra reshapes (mostly past key values reshapes):
  Latency 100 ms -> 90ms
- Fuse GELU op (@torch.jit.script)
  Latency 90ms -> 80ms
- Rewrite custom kernel for attention (prevent softmax f32 tensor creation)
  Latency 80ms -> 70ms/token
- Better hardware (8A100 80):
  Latency 70ms -> 45ms

Transformers optimization

- PP to TP:
  Half a day (expert)

- Remove extra reshapes (mostly past key values reshapes):
  2 days

- Fuse GELU op (@torch.jit.script)
  10s

- Rewrite custom kernel for attention (prevent softmax f32 tensor creation)
  1 day (expert)

- Better hardware (8A100 80):
  Half a day (modifying hard coded code around, and checking infra)

Big Takeaways

- Optimize of your feedback latency (time it takes to test an idea).
- Be in control of your tools - Drop Torch or Python if they cause issues (custom kernel, or GIL)
- Measure, measure, measure
- Profiling is a way to get insights for slow parts/bottlenecks, NEVER measure anything in profiling mode

Optimizing the webserver

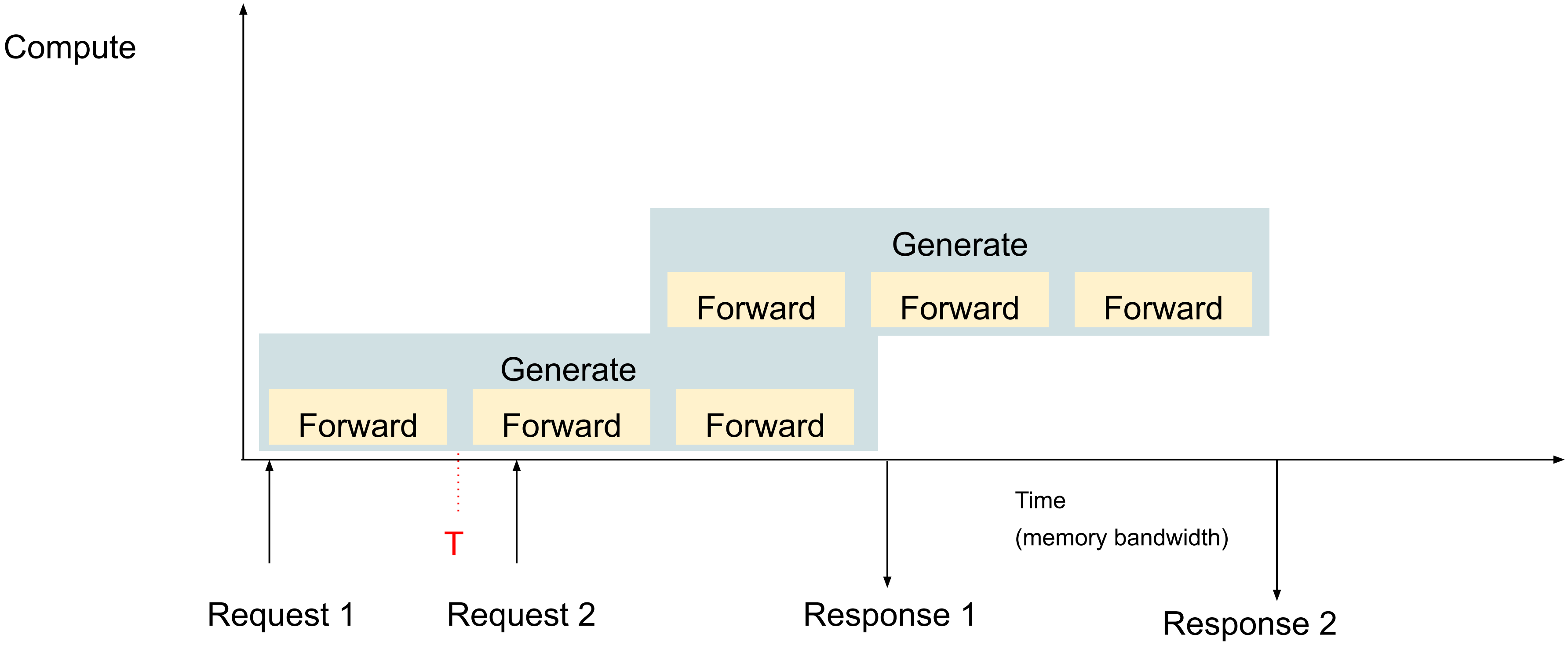
Webserver

- Lots of incoming requests, at random times
- Incoming requests have different profiles
  - Different parameters (sampling, greedy, temperature etc..)
  - Different lengths + max_new_tokens
- We want to minimize latency for EVERYONE

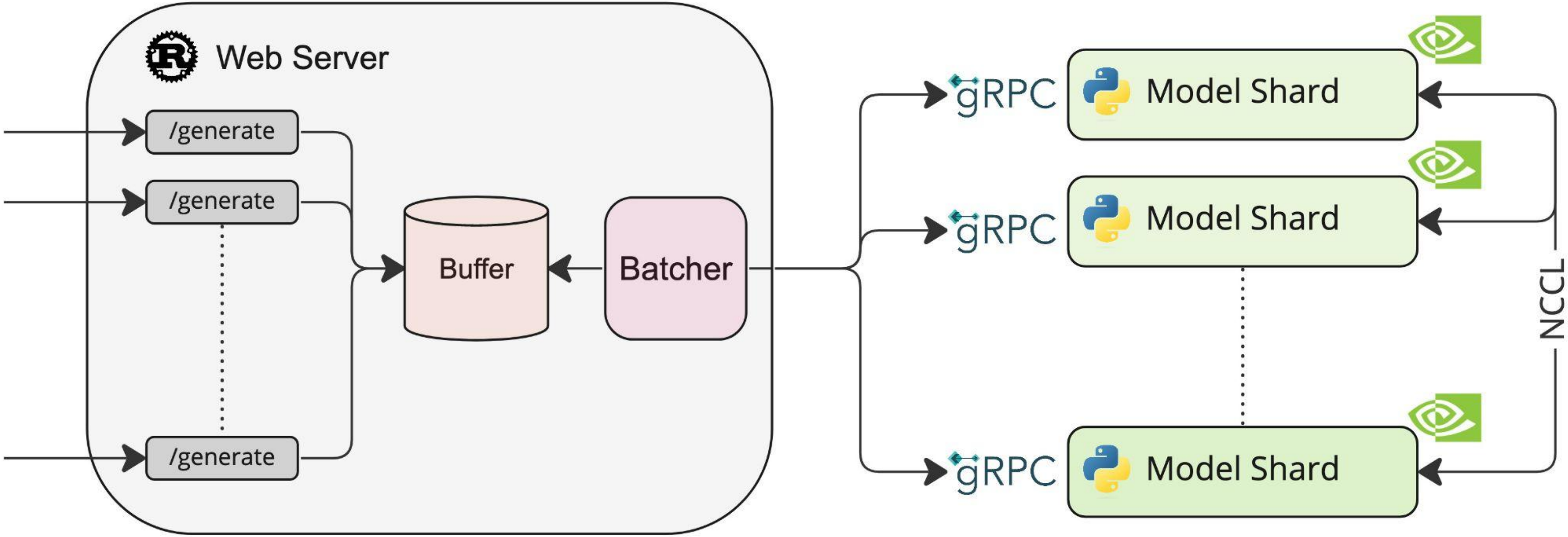
Webserver

- User perceived performance:
  - 100ms: Instant
  - 1s: Fast
  - 10s: slow
  - 1mn: Too long

Webserver


Compute
Generate
Forward
Forward
Forward
Generate
Forward
Forward
Forward
T
Time
(memory bandwidth)
Request 1
Request 2
Response 1
Response 2

Webserver


Web Server
/generate
/generate
/generate
Buffer
Batcher
gRPC
gRPC
gRPC
Model Shard
Model Shard
Model Shard
NCCL

Webserver

- Post process (LogitsProcessor) is much faster than the forward loop, so we can do per request treatment
- Long requests (max_new_tokens > 100) are slow anyway (>4.5s) so we can delay it more by allowing small requests (max_new_tokens < 20) to enter the batch midway. This requires running a forward WITHOUT past key cache.
- Every ~1s (20tokens) allow small requests to enter, delays large requests by ~5% (more in practice) but it CAPS the overall latency of small requests.
- Streaming new tokens allows for even better UX, time to something happening is reduced to 500ms (average delay)

Big Takeaways

- Load times were the developper bottleneck (10mn to load the weights)
- Access to big machines is sparse (try code correctness on smaller models *before* testing on larger models)
- Be in control of your tools - Drop Torch or Python if they cause issues (custom kernel, or GIL)
- Measure, measure, measure
- Have clear targets in minds, keep thinking about what you are optimizing for.