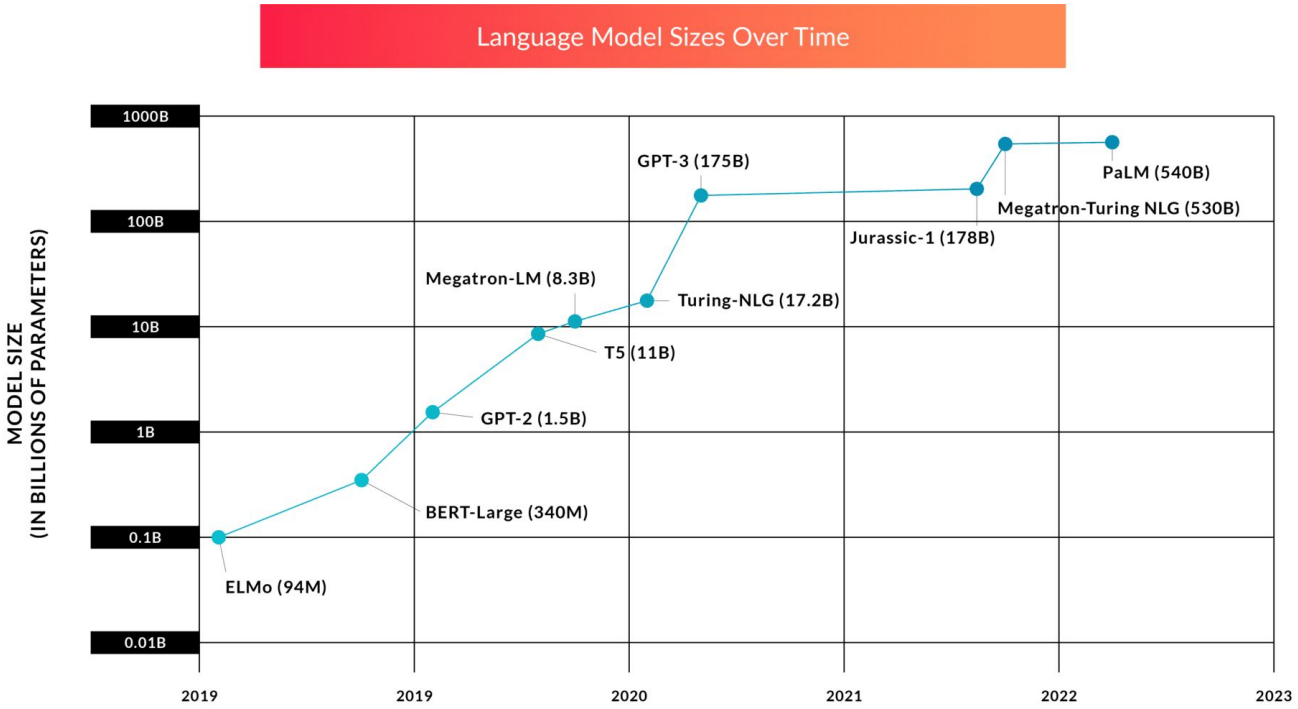
ORCA: A Distributed Serving System for Transformer-Based Generative Models

Presented by Aditya Prerepa

Some Background

Trends in Model Sizes



Costs of Serving Models

GPT3 175B instance requires ~$476,491.44/year

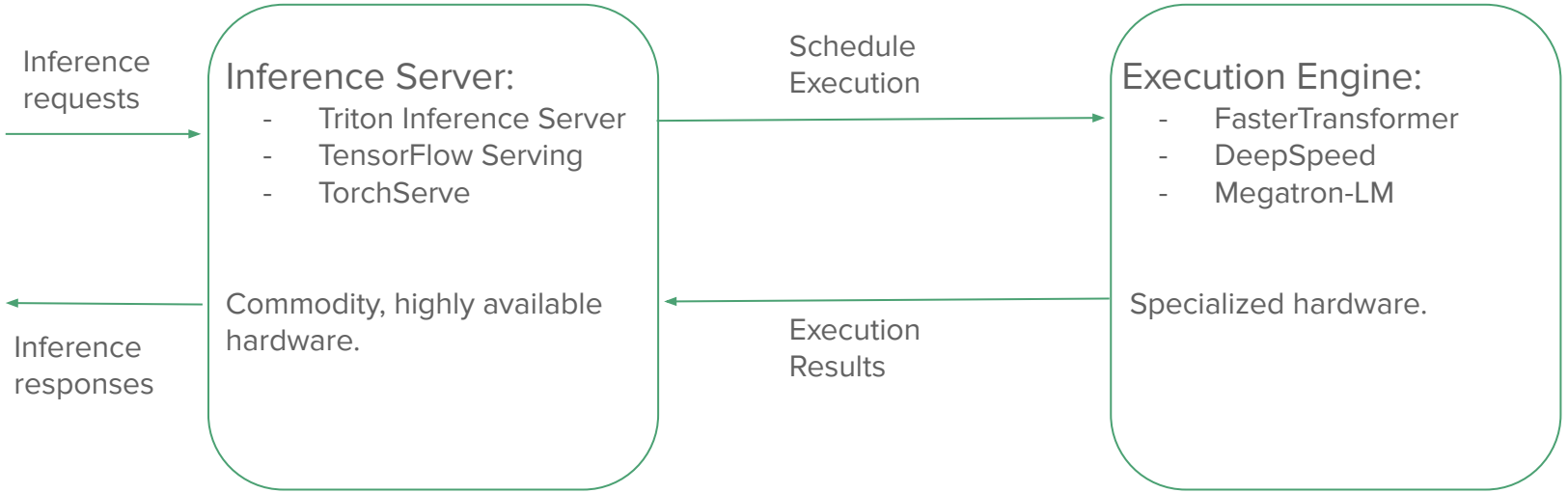
When companies serve many consumers in concurrent (1000 not implausible), it will cost $476M/year!

I’m terrified to look at how much OpenAI spends, just serving GPT for free.

We have to bring costs down!

Prior Work/Architecture

Model Serving Architecture

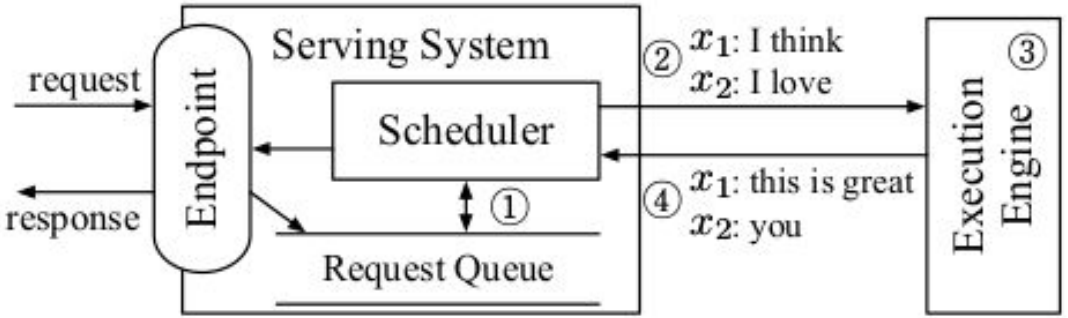


Inference Server <-> Engine Communication

Server pushes **batches** of requests (“prompts”) to the engine to exploit GPU parallelism.

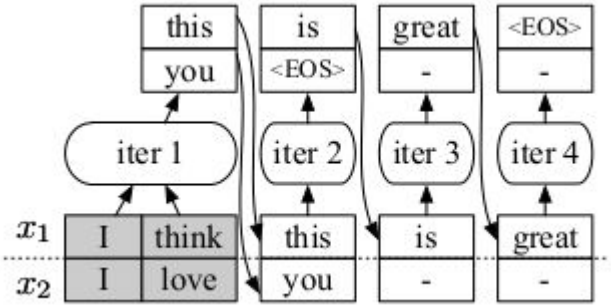
The engine executes every request, but has to wait until every request in the batch reaches EOS.

Only then are the predictions returned to the inference server, and consequently to the user.



Request x2 will cause **useless computation**

and has **more latency than needed**.



Key Point: Execution Granularity

Transformer-based models are autoregressive in manner (require multiple iterations building on previous output).

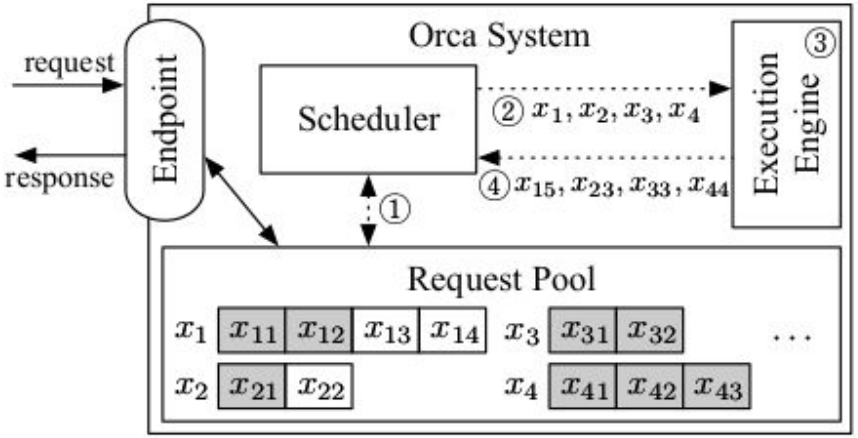
Can we do something better?

Instead of scheduling jobs at the batch/request level, can we schedule them at the **transformer iteration level**?

Iteration-Level scheduling

ORCA Scheduler:

1) Selects requests from pool to run next.
2) Invokes the engine to execute just **one iteration** of those requests.
3) Receives results for that iteration.



Scheduling Algorithm

Key details:

- First-come-first-served between requests.
- Has to be aware of remaining attention K/V slots.

Otherwise relatively intuitive scheduling algorithm.

Algorithm 1: ORCA scheduling algorithm

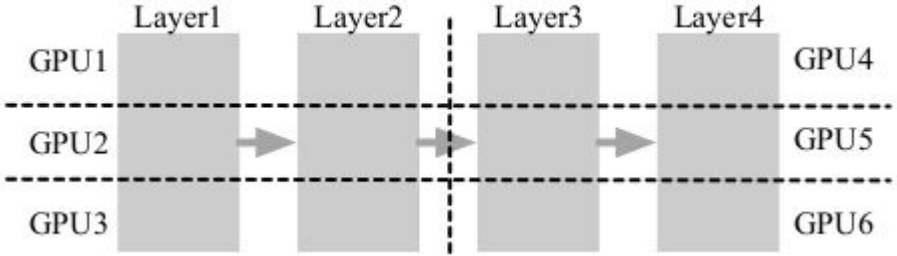
Params: *n_workers*: number of workers, *max_bs*: max batch size, *n_slots*: number of K/V slots

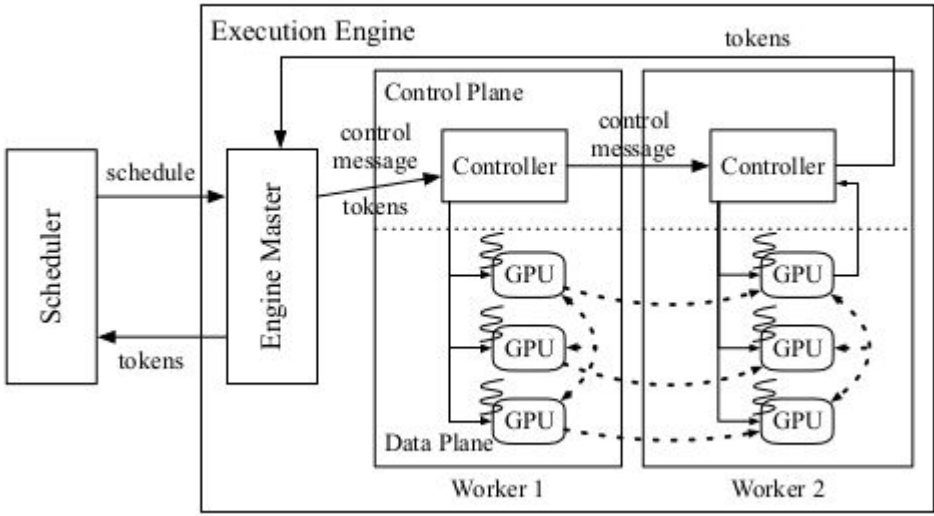
```
n_scheduled ← 0
n_rsrv ← 0
while true do
    batch, n_rsrv ← Select(request_pool, n_rsrv)
    schedule engine to run one iteration of the model for the batch
    foreach req in batch do
        req.state ← RUNNING
    n_scheduled ← n_scheduled + 1
    if n_scheduled = n_workers then
        wait for return of a scheduled batch
        foreach req in the returned batch do
            req.state ← INCREMENT
            if finished(req) then
                n_rsrv ← n_rsrv − req.max_tokens
        n_scheduled ← n_scheduled − 1

def Select(pool, n_rsrv):
    batch ← {}
    pool ← {req ∈ pool | req.state ≠ RUNNING}
    SortByArrivalTime(pool)
    foreach req in pool do
        if batch.size() = max_bs then break
        if req.state = INITIATION then
            new_n_rsrv ← n_rsrv + req.max_tokens
            if new_n_rsrv > n_slots then break
            n_rsrv ← new_n_rsrv
        batch ← batch ∪ {req}
    return batch, n_rsrv
```

(Not very interesting) System Architecture

- Split model parameters/computations across multiple GPUs and machines.
- Similar to FasterTransformer.





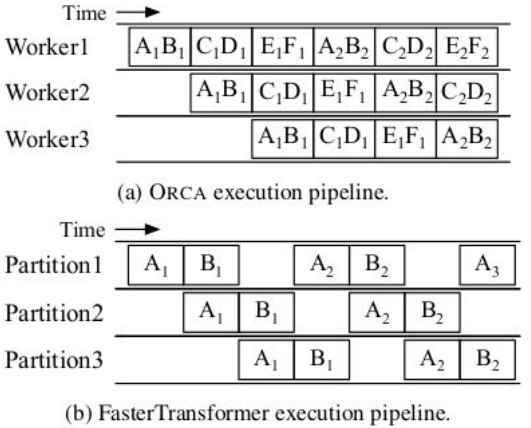
Added Benefit: Pipeline Parallelism

ORCA utilizes intra-layer and inter-layer parallelism (not very different from FasterTransformer).

- Each worker is responsible for an inter-layer partition of model.

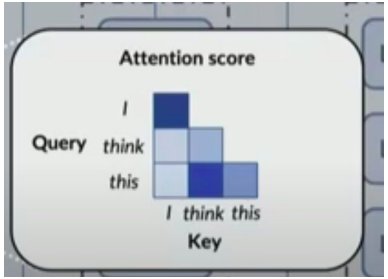
FasterTransformer+Triton have to wait for A and B to finish on all three partitions before moving on.

ORCA is free from this due to iteration-level scheduling.

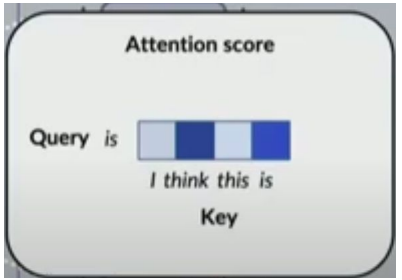


Problems with Iteration-Level Scheduling

Request Phases



1) Initiation Phase
   - First request to transformer, giving only user input.
   - Implemented as a single iteration, processing all input tokens in parallel, generate next token.
2) Iteration Phase
   - Process single token generated from previous iteration.
   - Use attention K/V of all previous tokens.



Usually save attention K/V to avoid recomputation.

Problems with Iteration-Level Scheduling

To exploit parallelism with batching, **all requests need to have the same number of tokens processed.**

This is because the Attention mechanism requires input tensors to have the same shape (which is based on the number of processed tokens).

Need to Coalesce B * [L, H] tensors into [B, L, H] tensor for batching.

B -> batch size, L -> input length, H -> hidden size.

When can’t requests be batched together?

Requests cannot be batched together when:

1) Both requests are in the initiation phase and have a different number of tokens.
    - Length not equal.
2) Both requests are in the incremental phase but are processing tokens at different indexes.
    - Attention Keys/Values have different shape.
3) Both requests are in different phases (one initiation, one incremental).
    - Initiation phase processes all tokens in parallel, incremental just one.

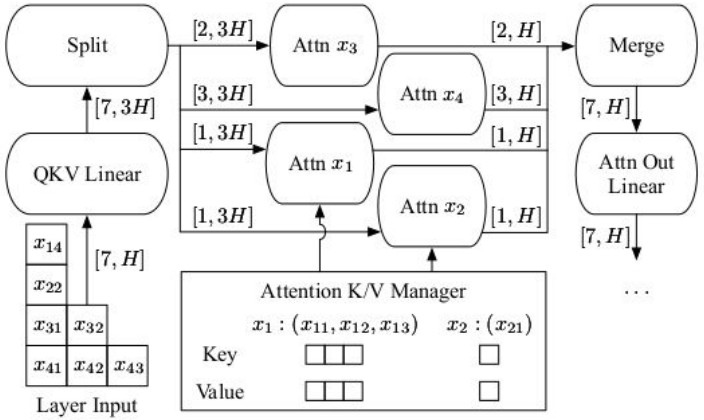
Solution: Selective Batching

Instead of batching **all** tensor operations, only batch the ones you can.

Flatten input tensors into 2-D tensor of shape [∑L, H], and feed into non-Attention operations (Linear, LayerNorm, Add, GeLU).

For Attention operations, split flattened tensor into individual requests and run Attention operation separately. Once completed, merge requests back.

Attention K/V manager provides Keys/Values separately for each request.



Implementation & Evaluation

Implementation

- 13K lines of C++, based on CUDA.
- gRPC for Control Plane<->Engine, NCCL for Data Plane.
- Implemented fused kernels for LayerNorm, Attention, and GeLU.
    - Also fused kernels of split attention operators by concatenating all thread blocks of the kernels for different requests.

Evaluation

Experiments ran on 8 NVIDIA 40GB A100 GPUs connected over NVLink.

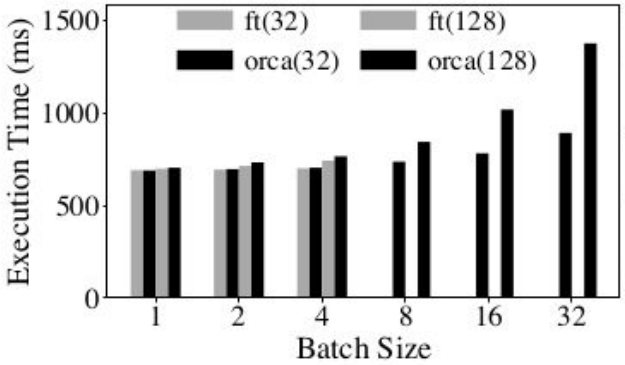
Tested on GPT-3 Models, parameter sizes of 13B, 101B, 175B, and 341B.

Baseline: Triton Inference Server + FasterTransformer.

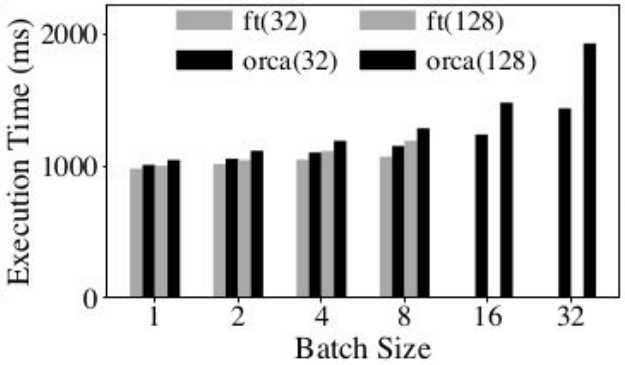
Tested on synthesized trace where each request is randomly generated by sampling the number of input tokens and a max_gen_tokens attribute.

Per-Request Scheduling

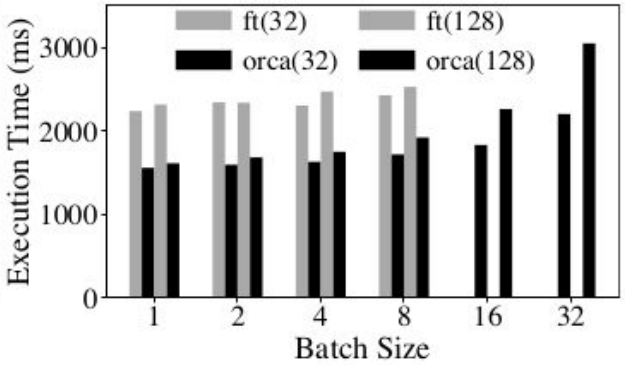
No iteration-level scheduling, just batched requests with ORCA vs FastTransformer.



(a) 13B model, 1 GPU.

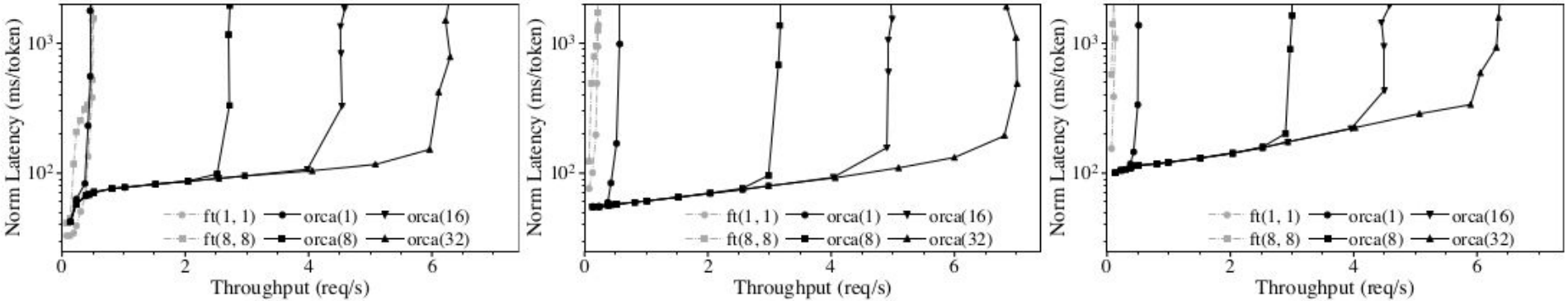


(b) 101B model, 8 GPUs.



(c) 175B model, 16 GPUs.

End-to-End Performance



(a) 101B model, 8 GPU. (b) 175B model, 16 GPUs. (c) 341B model, 32 GPUs.

orca(x) or ft(x) represents running orca/ft with max batch size x.

ft(x,y) represents FastTransformer with max batch size x and microbatch size y.

My Final Thoughts

- Overall excellent innovations - seems like a no-brainer.
- Authors evaluated on randomly generated traces. It’s unclear how far from random real LLM requests tend to be. In other words, what is the distribution of sizes for real requests and does that influence system characteristics?
- ORCA Attention K/V manager seems to cache request K/V values until the scheduler tells it to destroy, as opposed to FastTransformer/request-level where they can be destroyed immediately after the tokens return. What impact does this have on memory?

Thank you!

Extra Slides

Varying Batch Size