APPENDIX

A OPERATIONAL INTENSITY OF GQA

Table 3 describes the operational intensity (OI) of MatMul operations in Grouped Query Attention (GQA) (Ainslie et al., 2023) Transformer inference. Concurrent user count (U) is made distinct from batch (B) in order to separate the notion of batching in decode versus batching tokens from within a single user's prompt in prefill.

MatMul Operation	Operational Intensity (FLOPs/Byte)
Q, K, V, O	B
$QK^\top$, PV	$\frac{B}{U} \cdot \frac{n_h}{n_{kv}}$
FFN1/2/3	$max\left(B \cdot \frac{e_a}{e_r}, 1\right)$

Table 3. Operational intensity (OI) of MatMul operations in GQA Transformer inference. B: batch; U: concurrent users; n_h: attention heads; n_{kv}: KV heads; e_a: active experts; e_r: routed experts. $e_a = e_r = 1$ for non-MoEs. $U = B$ when performing decode only. $U = 1$ when performing prefill-only on a single prompt. Assuming 2-byte data type.

B COMMUNICATION COLLECTIVE ALGORITHMS

Three different AllReduce algorithms are considered by the LPU compiler: RedBcast (Algorithm 1), TiledRedBcast (Algorithm 2), and PAARD (Algorithm 3). Each algorithm can be interpreted with the following terms:

- Each LPU i holds a partial sum x_i. AllReduce computes $R = \sum x_i$ and distributes R to all LPUs.
- A tensor can be "sliced" into equally sized sub-tensors. Slice s of a tensor x_i is denoted by $x_i[s]$.
- A *node* is an LPU node with $N = 8$ LPUs. Point-to-point connection between LPUs of different nodes are called "bridge" links. There are B bridge links between two LPU nodes.
- TRANSMIT is a point-to-point transmit of a tensor.
- A BROADCAST of a tensor from LPU i consists of TRANSMITs of the tensor to all LPUs connected to i.
- A GATHER of tensors to LPU i consists of TRANSMITs of those tensors to LPU i.
- REDUCE of tensors a, b, c is $R = a + b + c$.

The three algorithms present a spectrum of latency-versus-bandwidth tradeoffs:

- RedBcast uses fewest hops but transmits most data. When tensor sizes are low, the performance of the collective is limited by hop latency; thus, the compiler chooses RedBcast to minimize the hops.
- TiledRedBcast transmits less data than RedBcast at the cost of an additional hop. For moderate sized tensors, the bandwidth advantage outweighs the cost of the additional hop.
- PAARD uses 3 more hops than TiledRedBcast, but is more bandwidth efficient; it is aware of constrained ("bridge") links and transmits less data both overall and on the constrained links.

Algorithm 1 RedBcast

input $x_i[S]$ on LPU i, $S = 1$
  Step 1: Reduce within node
  for all node n **do**
    for all LPU $i \in n$ **do**
      BROADCAST $x_i \rightarrow n$
    end for
    for all LPU $i \in n$ **do**
      $r_i \leftarrow$ REDUCE$(\{x_j : j \in n\})$
    end for
  end for
  Step 2: Gather across nodes
  for all nodes m, n, $m \neq n$ **do**
    $t_{mn} \leftarrow$ TRANSMIT $r_i \rightarrow n$ (over bridge links)
    BROADCAST $t_{mn} \rightarrow n$
  end for
  Step 3: Reduce to final sum
  for all node n **do**
    for all LPU $i \in n$ **do**
      $R_i \leftarrow$ REDUCE$(\{r_i\} \cup \{t_{mn} : m \neq n\})$
    end for
  end for

Algorithm 2 TiledRedBcast

input $x_i[S]$ on LPU i, $S = 8$
  Step 1: Reduce within node
  for all node n **do**
    for all LPU $i \in n$ **do**
      GATHER$(\{x_j[i] : j \in n\}) \rightarrow i$
      $r_i[i] \leftarrow$ REDUCE$(\{x_j[i] : j \in n\})$
    end for
    for all LPU $i \in n$ **do**
      BROADCAST $r_i[i] \rightarrow n$
    end for
  end for
  Step 2: Gather across nodes
  for all nodes m, n, $m \neq n$ **do**
    $t_{mn} \leftarrow$ TRANSMIT $r_m \rightarrow n$ (over bridge links)
    BROADCAST t_{mn} within n
  end for
  Step 3: Reduce to final sum
  for all node n **do**
    for all LPU $i \in n$ **do**
      $R_i \leftarrow$ REDUCE$(\{r_i\} \cup \{t_{mn} : m \neq n\})$
    end for
  end for

Algorithm 3 PAARD

input $x_i[N][B]$ on LPU i
  for all node n_d **do**
    Step 1: Reduce and gather to bridge links per node
    for all node $n_s \neq n_d$ **do**
      bridges $=$ { bridge from n_s to n_d }
      for all bridge b **do**
        GATHER$(\{x_j[n_d][b] : j \in n_s\}) \rightarrow b$
        $g_b \leftarrow$ REDUCE$(\{x_j[n_d][b] : j \in n_s\})$
        $t_b \leftarrow$ TRANSMIT $g_b \rightarrow n_d$
      end for
    end for
    Step 2: Reduction at the destination LPU
    for all bridge $b = (i, j)$ **do**
      $p_j[b] \leftarrow$ REDUCE$(\{x_j[n_d][b]\} \cup \{t_b\})$
    end for
    Step 3: Reduce at destination node
    for all LPU $i \in n_d$ **do**
      GATHER$(\{p_j[i] : j \in n_d\}) \rightarrow i$
      $R_i[i] \leftarrow$ REDUCE$(\{p_j[i] : j \in n_d\})$
    end for
    for all LPU $i \in n_d$ **do**
      BROADCAST $R_i[i] \rightarrow n_d$
    end for
  end for
  Step 4: AllGather
  for all node n_s **do**
    for all node $n_d \neq n_s$ **do**
      bridges $=$ { bridge from n_s to n_d }
      for all bridge b **do**
        $t_b \leftarrow$ TRANSMIT $R_b[n_s][b] \rightarrow n_d$
      end for
      for all LPU $i \in n_d$ **do**
        BROADCAST $t_b \rightarrow n_d$ (for bridge $b = (j, i)$)
      end for
    end for
  end for

C Runtime Mask Encoding Vector

The *mask encoding* is a vector of integers representing the causal bit mask for the attention mechanism. The runtime manages the mask encoding, and sends it to the LPU to generate the full bit mask for the current given request. The mask encoding is sized to the number of pages P, with each element ranging from $[0, page\ size]$ representing the context used per page. This representation allows for reconstruction of the bit mask without scaling directly with the context length, enabling lower runtime overheads from IO.

Figure 11 shows an example where a prompt of seven tokens first occupies two pages, fully occupying page 0, then ¾ of page 6. Two outputs are generated: the first ("fathers") is allocated to the last slot of page 6, before requiring another page for the second output ("brought").

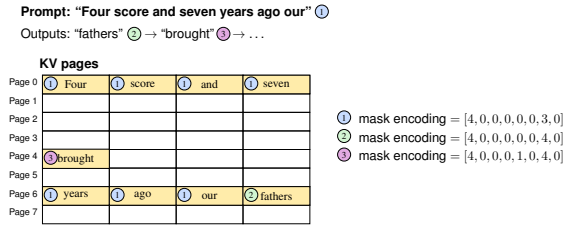


Figure 11. An example prompt and response's *mask encoding* for a page table with $P = 8$ pages, and a $page\ size = 4$ tokens. On the left, the KV pages allocation is shown for the prompt and generated tokens. On the right, the mask encoding vector sent from the runtime to the LPU is shown per timestep.

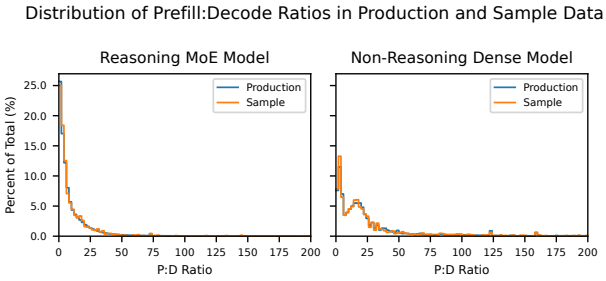


Figure 12. Histogram of P:D token ratios in 30 days of production requests and 2K sampled requests.

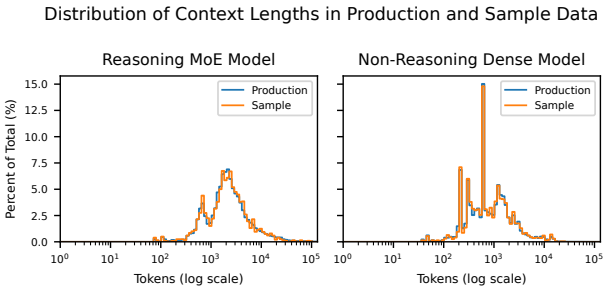
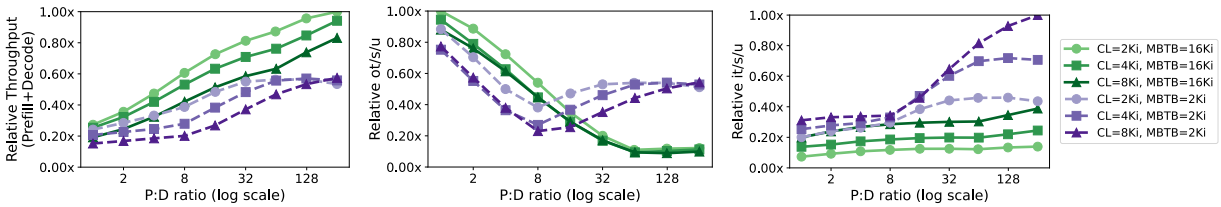


Figure 13. Histogram of context lengths in 30 days of production requests and 2K sampled requests.

D Experiment Methodology

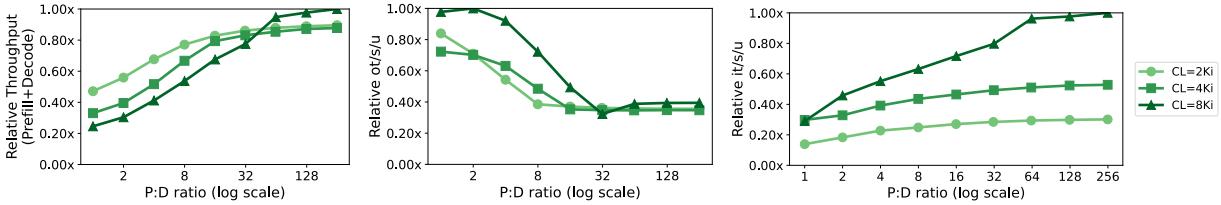
SGLang server was run with `--enable-mixed-chunk` to enable mixed prefill+decode batching when chunked prefill is active (SGLang Documentation, 2025). Under mixed chunking, `chunked_prefill_size` does not act solely as a per-request prefill chunk size; instead, it functions as a shared token budget for each mixed GPU iteration, with both prefill and decode tokens consuming that budget. We therefore treat `chunked_prefill_size` as the mixed-batch token budget (MBTB) in these experiments. The `max_prefill_tokens` parameter is left at its default value of 16Ki.

Prefix caching and speculative decoding (SD) are disabled to avoid content-dependent performance behavior, ensuring isolation of performance characteristics due to P:D and CL.



(a) B200 DGX system throughput (b) B200 DGX decode ot/s/u (c) B200 DGX prefill it/s/u

Figure 14. Measured throughput and latency of serving Qwen3-235B-A22B at varying prefill:decode token ratios (P:D) and context lengths (CL) for a B200 DGX using SGLang v0.5.8 at different mixed-batch token budgets (MBTB). Relative results normalized to the peak measured. Max concurrency limited to 256.



(a) B200 DGX system throughput (b) B200 DGX decode ot/s/u (c) B200 DGX prefill it/s/u

Figure 15. Measured throughput and latency of serving Qwen3-235B-A22B at varying prefill:decode token ratios (P:D) and context lengths (CL) for a B200 DGX using vLLM v0.17.1 with default max_num_batched_tokens. Relative results normalized to the peak measured.

vLLM experiments in Appendix D.2 are done similarly, but with no manual override of vLLM defaults (beyond disabling prefix caching and SD). In all experiments, queuing time is removed from it/s/u and TTFT calculations.

D.1 Production-Sampled Traffic

To approximate real-world workloads, we performed weighted sampling to collect 2K $(t_{\text{in}}, t_{\text{out}})$ token count pairs over 30 days of production traffic for a reasoning MoE model and non-reasoning dense model. The samples closely fit the observed P:D ratios (Fig. 12) and context length distributions (Fig. 13). Summary statistics for both samples are shown in Table 4.

	Reasoning MoE Model		**Non-Reasoning Dense Model**	
	Context Length	P:D Ratio	Context Length	P:D Ratio
min	73.00	0.01	37.00	0.06
mean	3513.48	4.92	1376.33	8.68
max	101261.00	988.69	24878.00	1699.00

Table 4. Summary statistics for context length and prefill:decode token ratio (P:D) for 2K sampled requests of production traffic.

In accordance with privacy laws, real user prompts were not recorded. Inputs were generated using t_{in} random tokens, and outputs constrained via stop-token masking with `max_tokens` $= t_{\text{out}}$. Public prompt datasets lacked the granularity to match the sampled distribution, and with prefix caching and speculative decoding disabled, random sequences were sufficient.

D.2 Other GPU Results

For completeness, we include here Qwen3-235B-A22B results for Section 6.1 using SGLang at a smaller client-side maximum concurrency of 256, to show that similar trends continue to appear across prefill:decode token ratios even when concurrency limits are set (Fig. 14).

We also include experiment results using vLLM (v0.17.1) (vllm project, 2025) in Figure 15. Throughput trends are similar to SGLang, with varying ot/s/u and it/s/u tradeoffs due to the different default scheduling policy employed by vLLM.