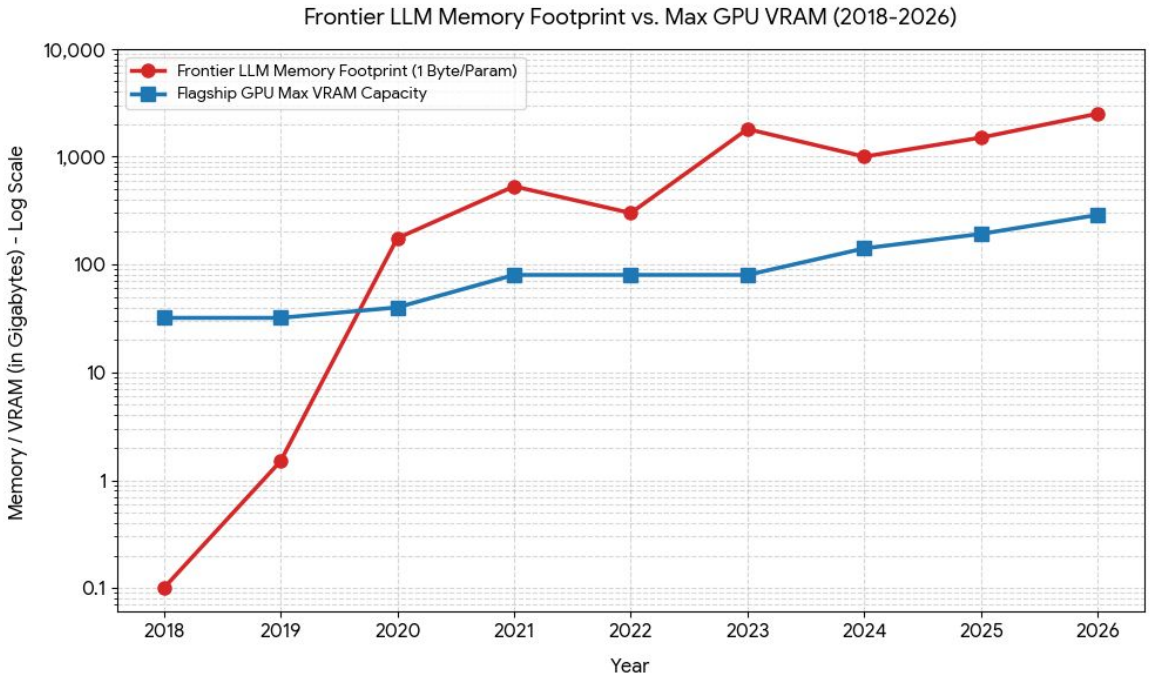
Performance Comparison Of Transport Mechanisms For LLM KVCache Transfers

Jamal Hadi Salim | Nabil Bitar | Pedro Tammela | Victor Nogueira

The Challenge: The Memory Wall Needs Networking



🔴 **State-of-the-Art Dense LLM Model Size (Solid Red Line)**

Assumes a conversion factor of 1 Byte per parameter (1 Billion Parameters = 1 GB).

- **2018: 0.11 GB** (110 Million Parameters)
- **2019: 1.5 GB** (1.5 Billion Parameters)
- **2020: 175.0 GB** (175 Billion Parameters)
- **2022: 540.0 GB** (540 Billion Parameters)
- **2024: 400.0 GB** (400 Billion Parameters)
- **2026: 800.0 GB** (800 Billion Parameters)

🟦 **Flagship GPU VRAM Capacity & Estimated Release Price (Solid Blue Line)**

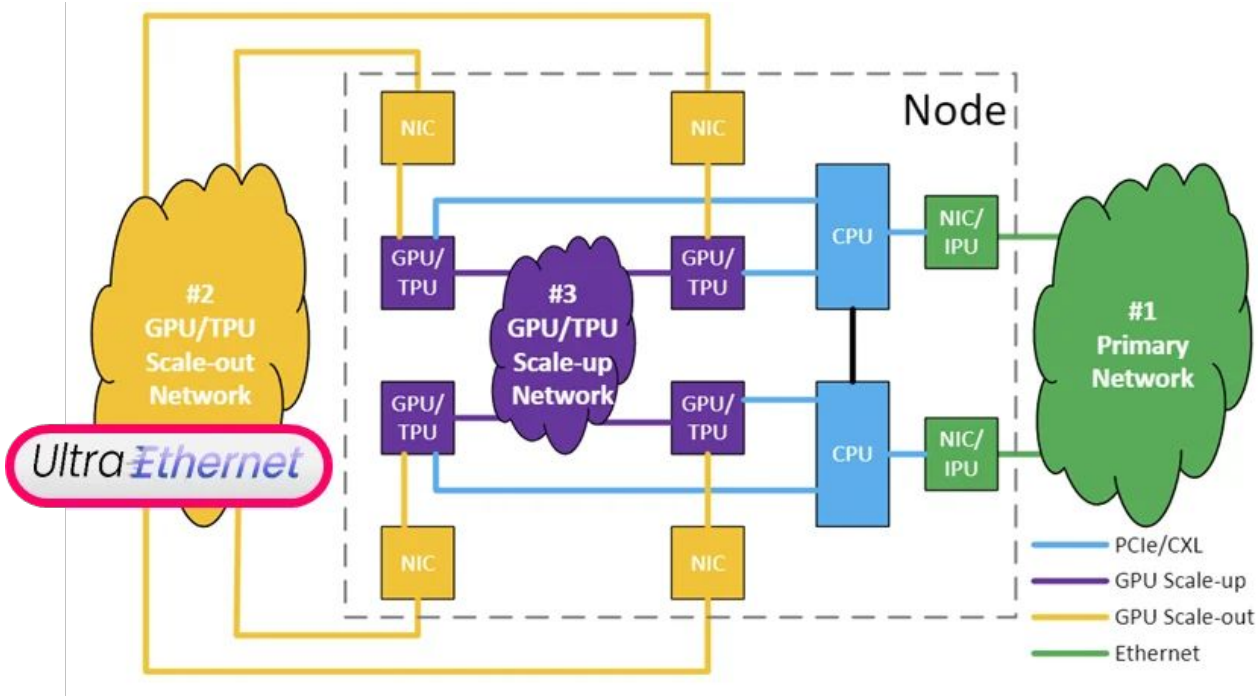
Prices reflect the official enterprise/datacenter board launch baseline at release. Jarvis Labs +2

- **2018: 32 GB | ~$10,000** *(NVIDIA V100 SXM2 32GB)*
- **2019: 32 GB | ~$10,000** *(NVIDIA V100S PCIe refresh)*
- **2020: 80 GB | ~$15,000** *(NVIDIA A100 SXM4 80GB launch)*
- **2022: 80 GB | ~$30,000** *(NVIDIA H100 SXM5 base)*
- **2024: 141 GB | ~$40,000** *(NVIDIA H200 SXM)*
- **2026: 192 GB | ~$35,000 - $40,000** *(NVIDIA B200 / Blackwell system architecture*

Created by Gemini with prompt asking for trends of how the model memory requirement for 1B quantization fit in state of the art GPUs since 2018

*Model growth outliers: 2022 - Chinchilla Scaling Laws takes effect and 2023 - MOE becomes popular

The Challenge: The Memory Wall Needs Networking



https://netdevconf.info/0x18/sessions/bof/introduction-to-ultra-ethernet-consortium-10.html

Motivation And Goal

Avoid dealing with cost of:

- Ever growing (size) Models
- Evolving approaches to model optimizations
- Evolving approaches to model implementation

Essentially: Constantly changing GPU hardware and cluster sizes to keep up

Goal

Keep up with the fast moving environment while focusing on networking infra

- Experiment and innovate new networking technologies without need to keep up with newer GPUs and worrying about different model techniques/sizes
    - Assumptions is we are dealing with memory requirements

Scope

Talk Scope

Scope of the talk is on inference disaggregated prefil/decode with a focus on KVCache Network Transfer. Agenda:

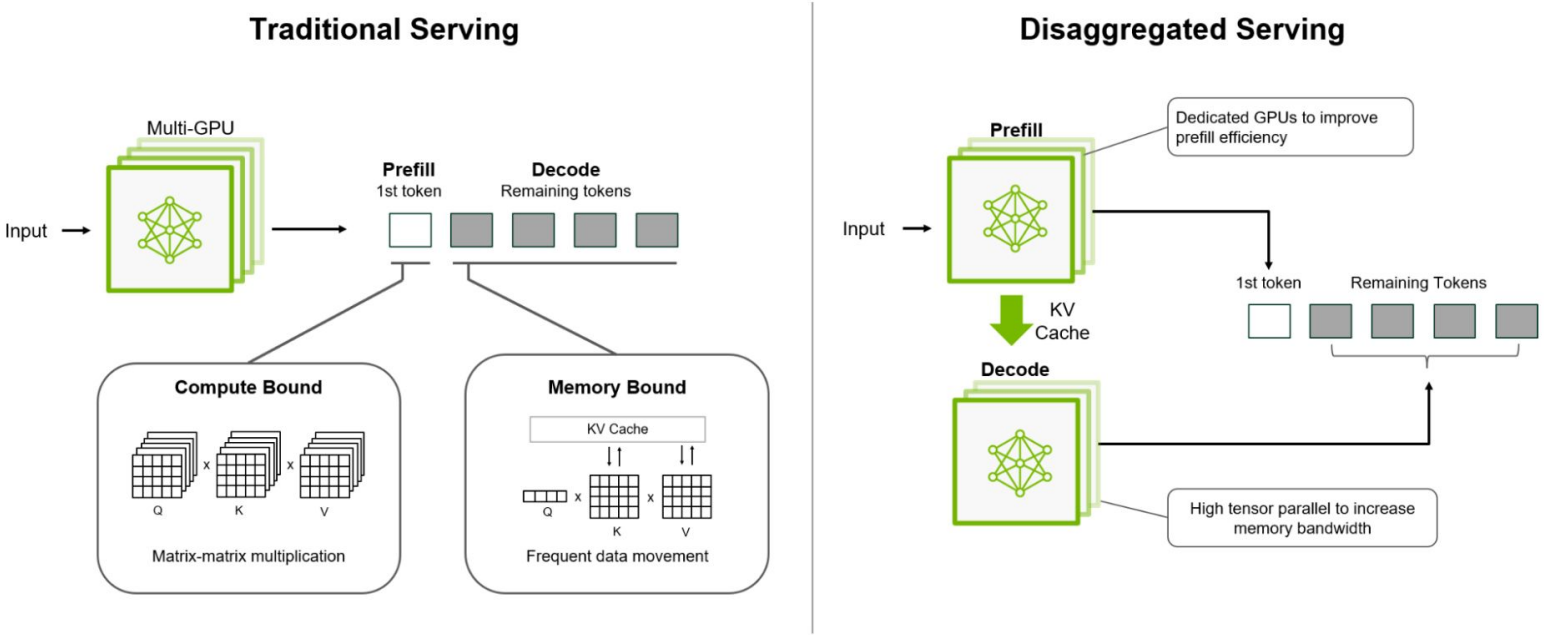
- Short description of inference/kvcache and where networking fits
- Quantization of the amount of kvcache network traffic generated by 3 different models
- Our tooling approach
- Experiment description and results
- Challenges along the way
- Recommendations

Experimental Scope

- Transports implemented:
  - Standard TCP (No special NIC features, Foundational NIC good enough)
  - Devmem ZC TCP (Header split, Intel 8xx has support)
  - Io_uring ZC (Header split, Intel 8xx has support)
  - RDMA via Infiniband/RoCEv2 (Offloaded Stack, Intel 8xx has support)
- GPU: H100
  - FP8: 3958 TFLOPs / FP16: 1979 TFLOPs
  - Mem Capaciy: 80GB
  - Mem BW: 3.3 TB/s
- Models (Pick a few recent models; all dense)
  - Gemma 4 31B
  - Qwen 3 32B FP8
  - Mistral 3.5M 128B

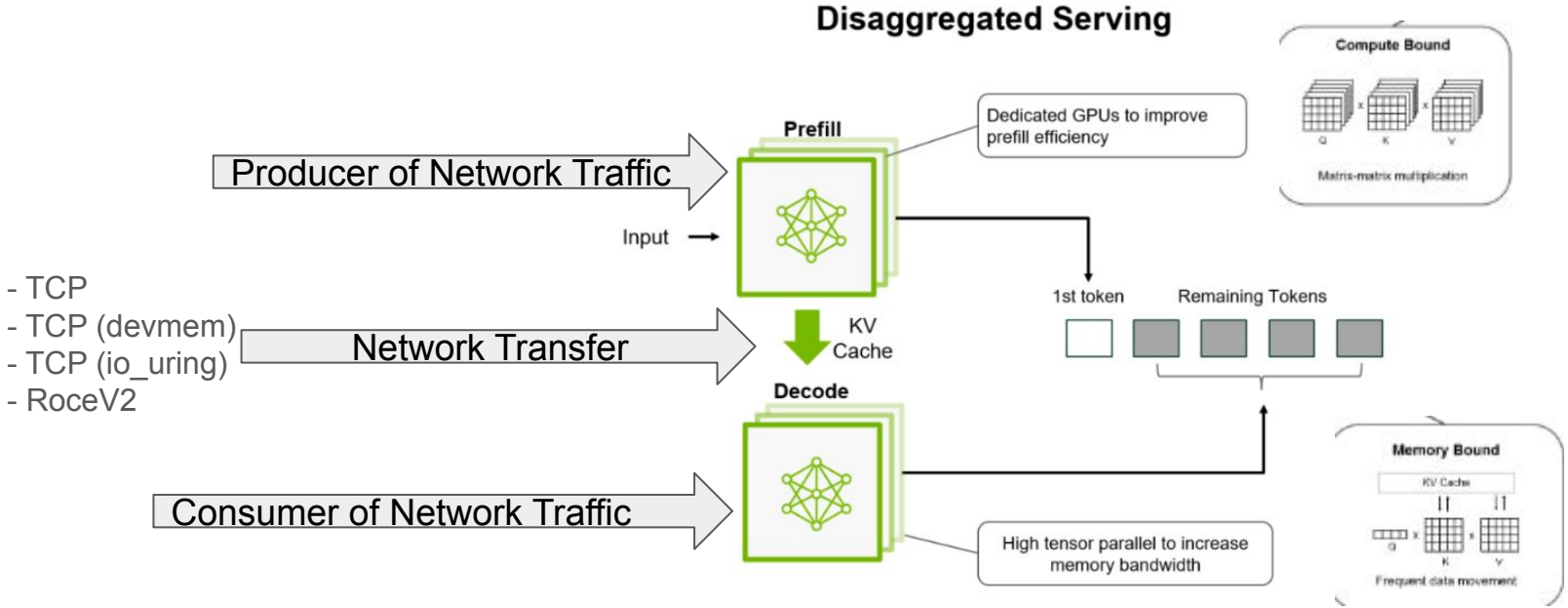
KVCache Production Vs Consumption

Inference Disaggregation Serving



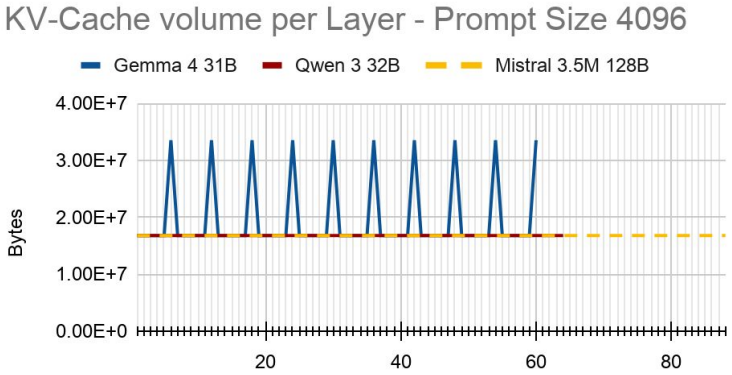
https://developer.nvidia.com/blog/introducing-nvidia-dynamo-a-low-latency-distributed-inference-framework-for-scaling-reasoning-ai-models/

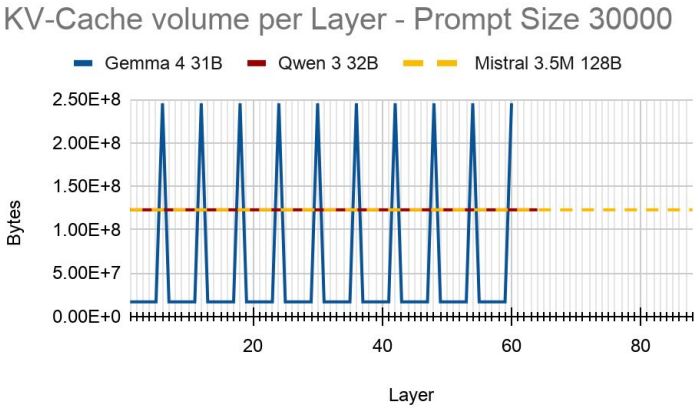
Talk Focus: KvCache Transfer

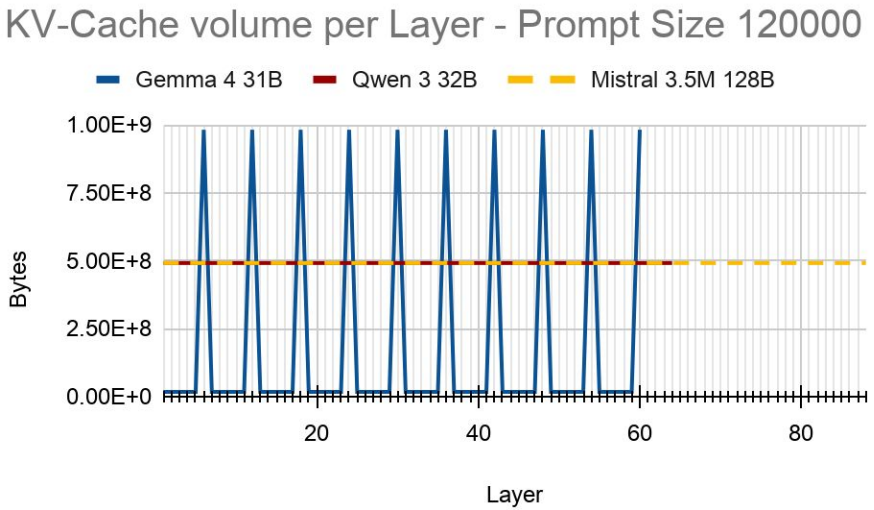


In production inference serving engines, like vLLM, KV-Cache is transferred between prefill and decode for each model layer

Per Layer Transfer Pattern: The Model Influence

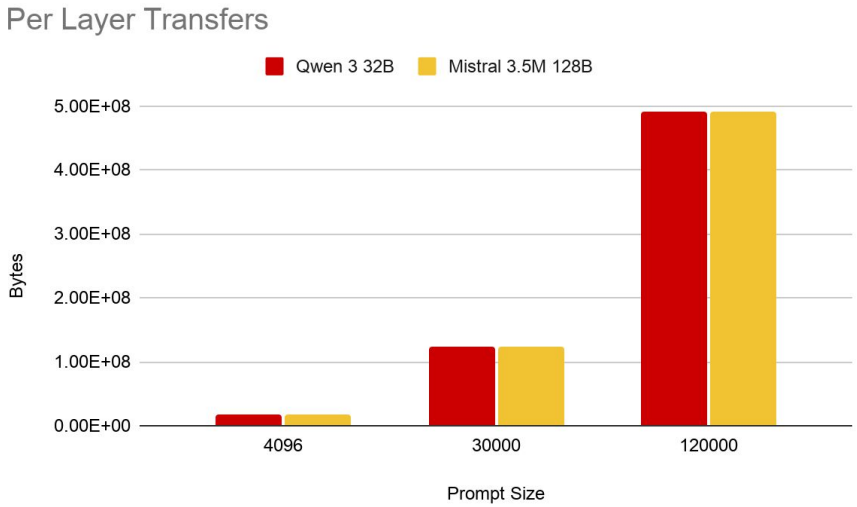






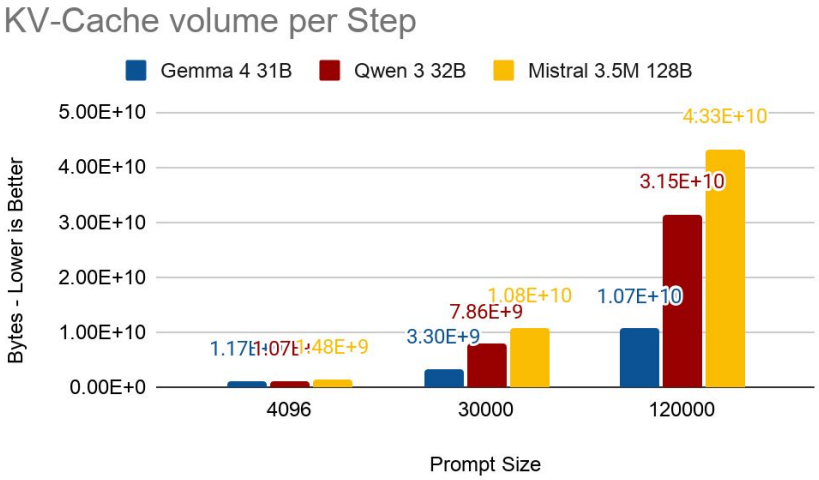
How Much Data Are We Talking About?

Per Layer Generated Data



- Mistral and Qwen3 send the exact same message size, the only difference is the number of layers each model contains

Generated Data To First Output Token



KV-Cache volume has an effect on TTFT, RAM needs and network data requirements

As observed: Gemma 4, regardless of the prompt size, always generates the first token faster than the other models as the amount of data transferred is much smaller

Compute/Communication Overhead Assumptions

Computation time considers the following overheads:
- memory bandwidth of an H100 (~ 3.3TB/s)
- time to wake up cpu + overhead of setting up the transfer

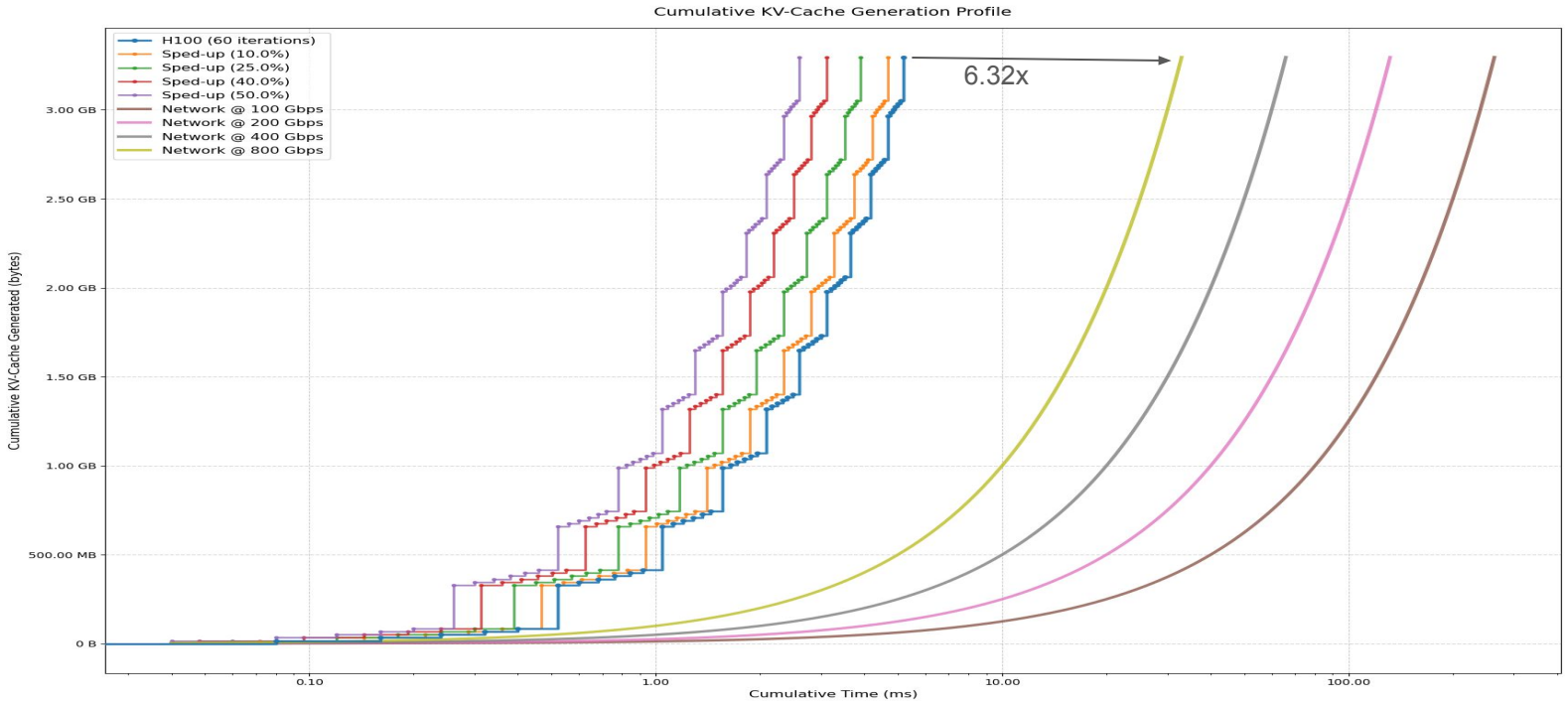
Gemma window sampling:
- Sliding layer: 80 usecs
- Full layer: 120 usecs

Factor and emulate computed per-layer “KVCache Production” time it takes to produce the cache (compute phase)

Communication phase (not emulated) is the time it takes to transfer the cache for given link capacity

Scale Out Network Link Speed Vs KVCache Production

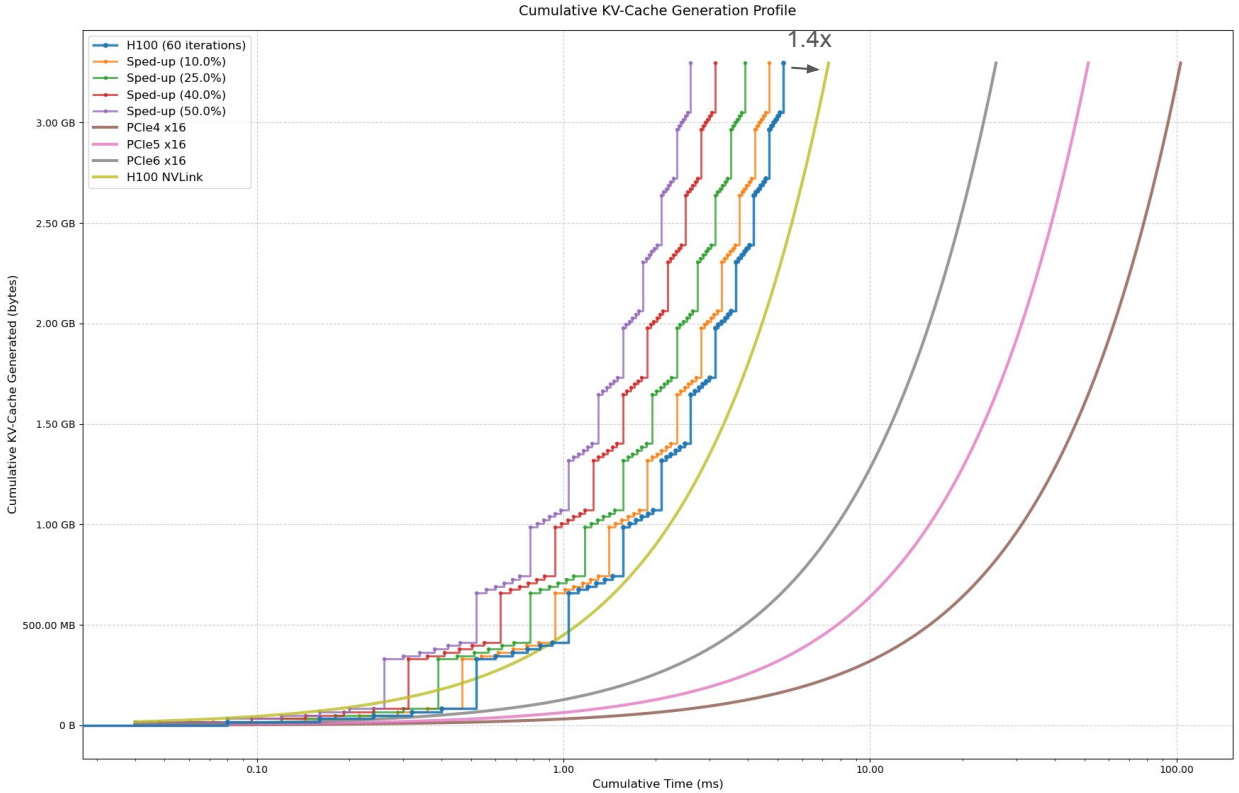
Gemma 4, prompt size 30000



Scale UP Link Speed Vs KVCache Production

Gemma 4, prompt size 30000

*H100 NVlink ~3.6Tbps



KVCache Production vs Network Link speed Conclusion

General

1. The model optimization techniques have huge influence on the amount of kvcache produced {number of layers, window sampling, etc}
2. The GPU produces data much faster than the network can consume. Meaning network link is always the bottleneck (regardless of link capacity)

Note: The traffic pattern difference between the different models (bursty or not) doesn’t show any visible issue in any transport

Effect Of Network Link Speed on TTFT

what is considered a good value for TTFT

For most user-facing applications, a **Time to First Token (TTFT)** of **under 200 milliseconds (ms)** is considered excellent, while anything **under 500 ms** is considered a good, snappy target for production systems.

Because TTFT directly impacts the human perception of speed (how quickly the UI reacts after clicking "Submit"), optimizing this metric is crucial for conversational interfaces.

TTFT Performance Benchmarks

The quality of your TTFT is heavily tied to your use case and user expectations:

- ⚡ **Excellent (<200 ms):** Feels instantaneous. Ideal for highly interactive chatbots, live customer support agents, and voice-to-voice AI systems.
- ✅ **Good (200 ms – 500 ms):** Noticeable but acceptable pause. Standard target for enterprise web applications and search tools.
- ⚠️ **Acceptable (500 ms – 1,000 ms):** Noticeable delay. Acceptable for complex background tasks (e.g., summarizing a long document) but causes slight user friction in conversational chat.
- ❌ **Poor (>1,000 ms / 1 second):** Feels sluggish. Users may assume the system is frozen, leading to dropped sessions or repeated button clicks.

KVCache Production vs Network Link speed Conclusion

Reminder: The GPU produces data much faster than the network can consume.

Lets assume zero overhead to get data out of the GPU memory onto the network link

- Implies: The Network link is always 100% busy (no matter what link size!)
- Ergo the network link and transport protocol/approach used to transfer the kvcache dominates the TTFT result

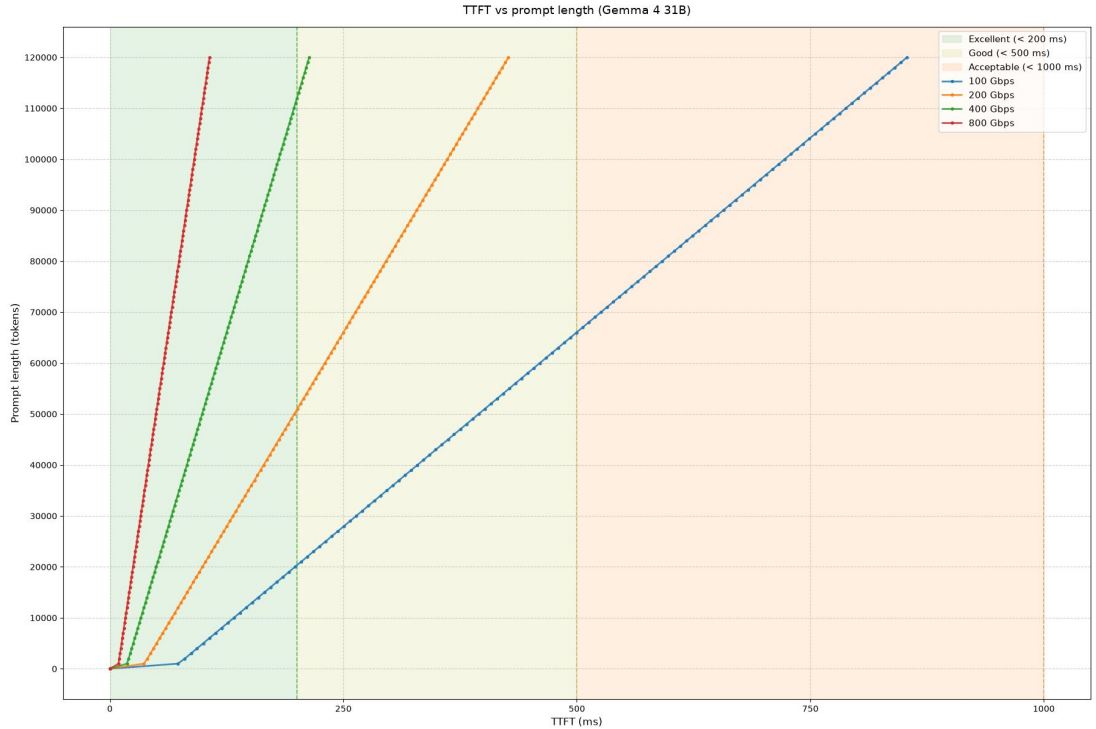
The major factors influencing the KVCache size (ignore multi-users and batching) are:

- The input size - Sashiko code review for example could chew as much as 1M tokens input!
- The model as we have illustrated so far..

TTFT Network Effect

Gemma 4 31B

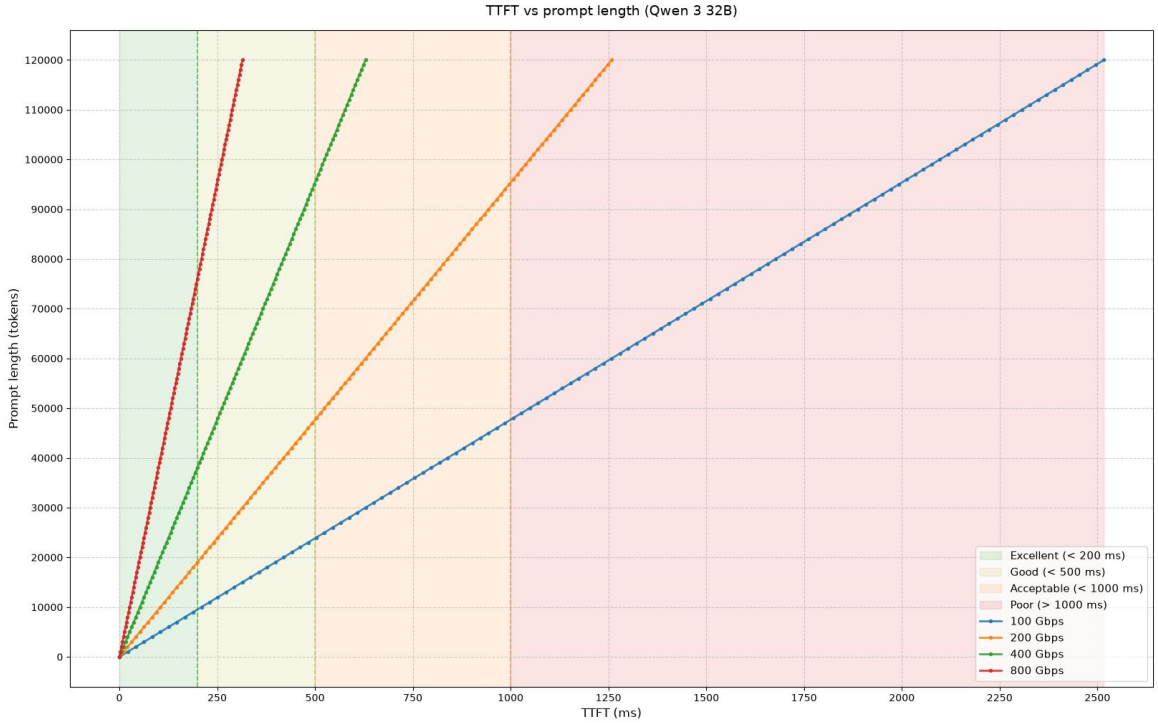
**Red zone Effect on TTFT
at 100gbps ~65K input



TTFT Network Effect

Qwen3 32B

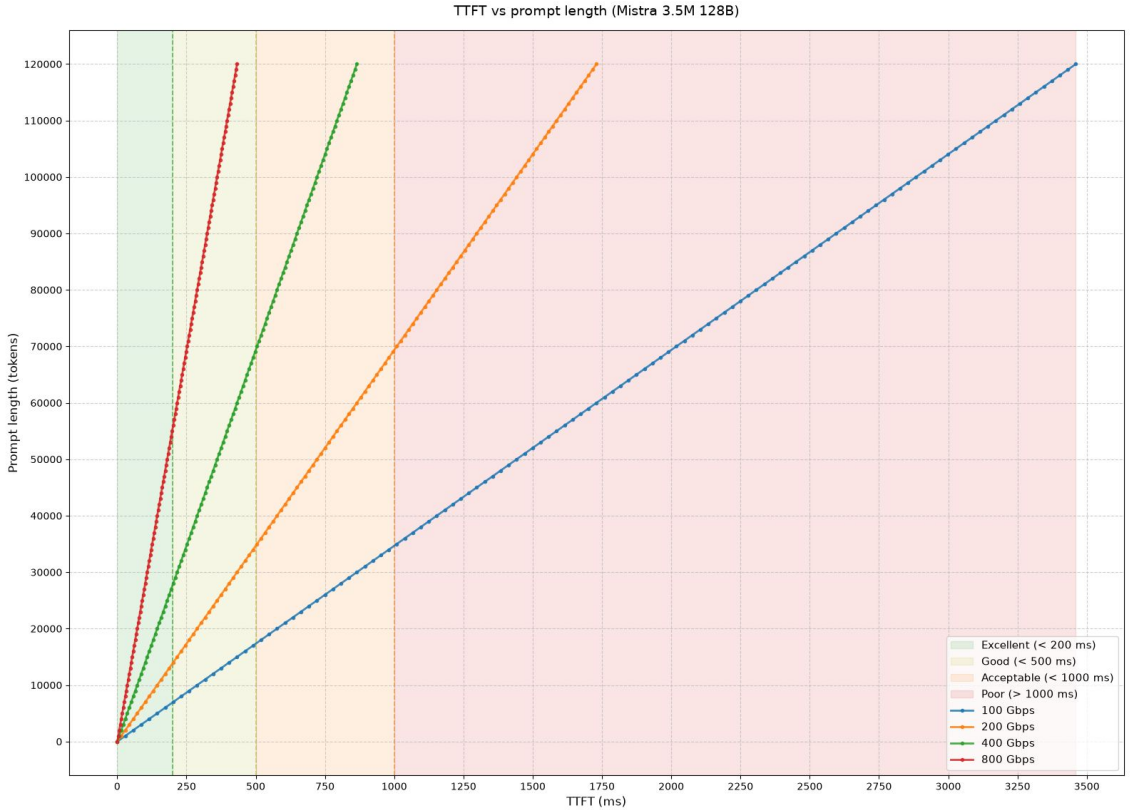
**Red zone Effect on TTFT
at 200gbps ~95K input
At 100gbps ~45K



TTFT Network Effect

Mistral 3.5M

**Red zone Effect on TTFT
at 200gbps ~70K input
100gbps ~35K inputt



TTFT vs Network Link speed Discussion

The model optimization techniques have huge influence on the amount of kvcache produced {number of layers, window sampling, etc}

Observation: most inference serving engines (vLLM in our test scenario) care only about kvcache transfer. The first generated token is discarded and regenerated on decode side.

So main task we focus on is the transfer of kvcache between the two nodes

Since we have established that link speeds are the bottleneck… Link speeds and network transport bottlenecks have huge impact on the TTFT

Formulating the problem: KVCache transfer is a bulk unidirectional movement of a very large amount of data (which occupies all the bandwidth you can throw at it!). IOW, this is a throughput bottleneck (and not latency)

Implementation: Challenges/Decisions

Solution: Build Own Tooling

simulates “Real” LLM distributed network traffic based on LLM Model characteristics

Means No GPUs needed but real network gear is needed

- NICs, distributed topology etc

Reminder of Stated Goal:

Keep up with the fast moving environment while focusing on networking infra

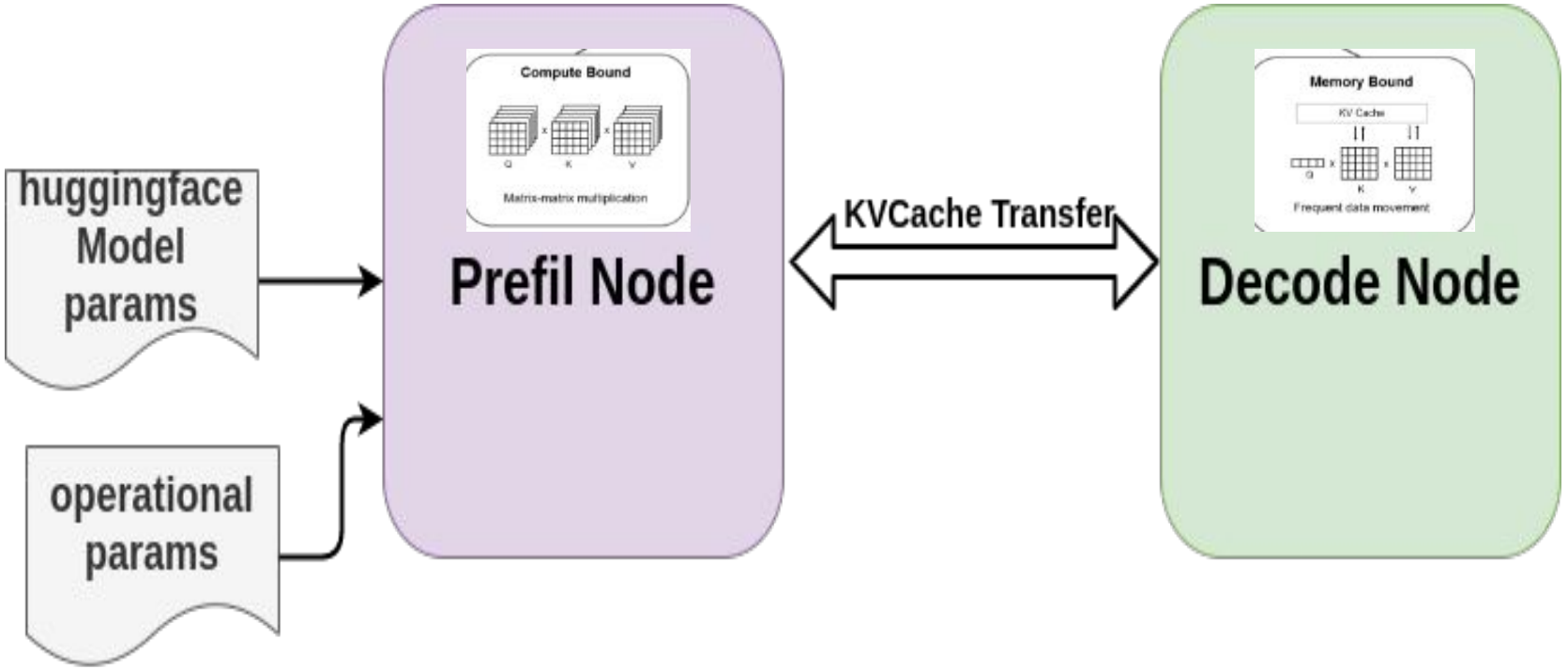
- Experiment and innovate new networking technologies without need to keep up with newer GPUs and worrying about different model techniques/sizes
  - Assumptions is we are dealing with memory requirements

How?

Use arithmetic formulas

- GPU computation based on the model and GPU
- pick a workload (input and output tokens) and use formulas that derive traffic for collectives and peer2peer (p2p) per NIC

Design Decision: Implement Our Own Tooling



Formulating GPU Computation (Matrix Multiplication)

The total computation, in seconds, is estimated using the following formula:

$$\frac{2 \cdot P + 4 \cdot L \cdot S \cdot H}{F}$$

P: Parameter count / L: Number of layers / S: Input tokens length / H: Hidden size / F: GPU FLOPs

- The per layer computation is to simply divide the result by the number of layers
- The input tokens length (*S*) changes depending if it's prefill or decode:
  - On prefill it's the prompt size
  - On decode it's always 1 as the process is autoregressive
- *P* tends to dominate the entire equation which leads to:
  - The bigger the model the higher the FLOPs needed

*** Formula is an approximation based on https://www.youtube.com/watch?v=X8H1t6DPY94

Formulating KV-Cache Traffic

The per layer KV-Cache is estimated using the following formula:

$$2 \cdot N_{blocks} \cdot b \cdot h \cdot N_{kv} \cdot \phi$$

N_{blocks}: Number of blocks / b: Block size / h: Head dimension / N_{kv}: Number of KV Heads / ϕ: Data type in bytes

- The block size is usually set to 16 tokens by vLLM due to Paged Attention
- The number of blocks is determined by either:
  - Full attention layer: *ceil[prompt token length / block size]*
  - Sliding attention layer: *ceil[min(prompt, sliding window) / block size]*

Note: The input size is a huge influence on the KvCache (upper bound of context size)

Validating Formulae

In order to confirm our equations we profiled all 3 models using NVIDIA’s Nsight profiler to test the KVCache Formula

4 H100 GPUs interconnected with an NVLINK Fabric

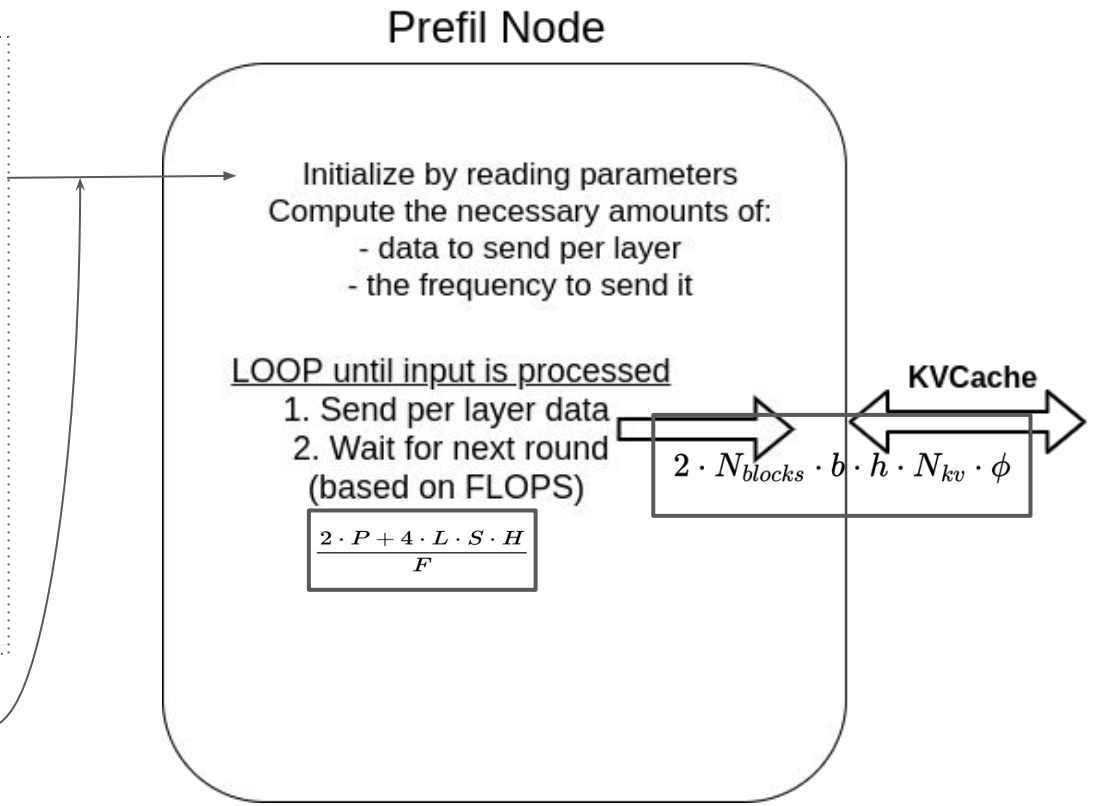
- 1-2 H100 used as prefill node(s) and the others used as decode

Architecture Overview: Prefil Node

Qwen3 32B FP8


```
{
  "architectures": [
    "Qwen3ForCausalLM"
  ],
  "attention_bias": false,
  "attention_dropout": 0.0,
  "bos_token_id": 151643,
  "eos_token_id": 151645,
  "head_dim": 128,
  "hidden_act": "silu",
  "hidden_size": 5120,
  "initializer_range": 0.02,
  "intermediate_size": 25600,
  "max_position_embeddings": 40960,
  "max_window_layers": 64,
  "model_type": "qwen3",
  "num_attention_heads": 64,
  "num_hidden_layers": 64,
  "num_key_value_heads": 8,
  "rms_norm_eps": 1e-06,
  "rope_scaling": null,
  "rope_theta": 1000000,
  "sliding_window": null,
  "tie_word_embeddings": false,
  "torch_dtype": "bfloat16",
  "transformers_version": "4.51.0",
  "use_cache": true,
  "use_sliding_window": false,
  "vocab_size": 151936,
  "quantization_config": {
    "activation_scheme": "dynamic",
    "fmt": "e4m3",
    "quant_method": "fp8",
    "weight_block_size": [
      128,
      128
    ]
  }
}
```


Workload:
Input size, GPU type



Why Not USe Existing Communication Libraries?

Library	Developer	Use Case	Plugins	HW Agnostic	RDMA Support	TCP Support	TCP Devmem Support	GPU Required
NCCL	NVIDIA	Collectives	Yes	NVIDIA GPUs Only	Yes	Yes	Yes	Yes
NIXL	NVIDIA	P2P	Yes	Yes	Yes	Yes	No	No*
UCCL	UC Berkeley	Collectives	No	Yes	Yes	No	via NCCL	No*

- NCCL is the only library that supports Devmem TCP thanks to the Google Plugin but requires GPU presence (io_uring is even harder)
- NIXL relies on UCX for network transports and even though UCX has support for plugins it does not currently support any form of TCP Zerocopy
  - NIXL itself also supports plugins but none with devmem support (non-trivial)
- UCCL supports Devmem TCP by wrapping NCCL (which requires GPUs)

Experiment Setup Details

Setup details

- Intel Xeon Platinum 8468 48C / DDR5 128GB
  - 105MiB L3 Cache
- Hyperthreading off
- Intel E810 2x100Gbps (Intel ICE)
  - Servers are connected back to back, single 100Gbps
- The servers are identical machines running the same kernel version and parameter tweaks
  - Net-next @ 6a4c4656b0d2d4056a1f0c35442db4e8a5cf8021
- Single Prefil and decode nodes (implies single flow)
- For devmem/iouring on ICE we are using the following patch set:
  - https://lore.kernel.org/all/20260224174618.2780516-1-aleksander.lobakin@intel.com/

ICE Setup details

- MTU Sizes:
  - TCP: 3698 / Devmem: 4148 / RDMA: 4200
  - TCP Header data split enabled for devmem/io_uring
- Device rings rx: 8160 / tx: 8160
  - To absorb high throughput and bursty traffic
- Adaptive RX/TX off with {rx/tx}-usecs 25
  - Buffers ~10% of the descriptor ring before firing an IRQ (about 305 KiB)
  - NOTE: Ice rounds down this number to 24
- In both machines we bind the single flow test to queue 1:
  - For RX we use flow steering into queue 1 / For TX we use XPS
  - Queue 1 is served by CPU 1
    - All TCP packets are served from this CPU

System Setup details

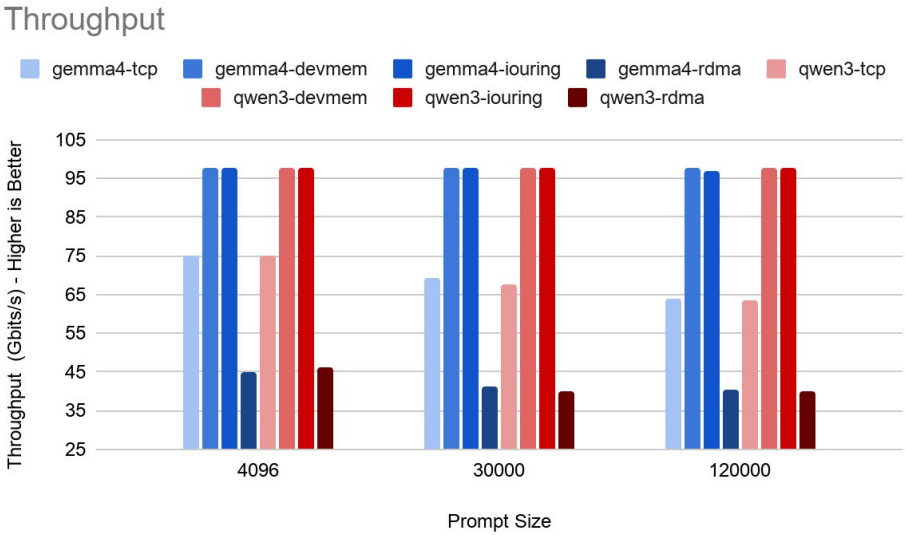
- tcp_{rmem/wmem}: "4096 87380 105000000"
  - Max limited by L3 Cache Size
- Steer uninteresting traffic(ARP etc) to CPU 0, bind network stack to CPU 1 and bind the application to CPU 3
  - Ensures high throughput with ample room for the network stack to process TCP
  - The tradeoff is a small latency penalty of accessing cache lines between CPUs 3 and 1
- Governor set to performance mode with idle states disabled
  - Ensures predictable tail latencies
- Kernel command line tweaks:
  - *init_on_alloc=0 hardened_usercopy=off mitigations=off intel_iommu=on iommu=pt isolcpus=0-7 nohz_full=0-7 rcu_nocbs=0-7 rcu_nocb_poll irqaffinity=8-15 kthread_cpus=8-15*
  - Minimize background jitter
  - Effectively we only use CPUs 1 and 3

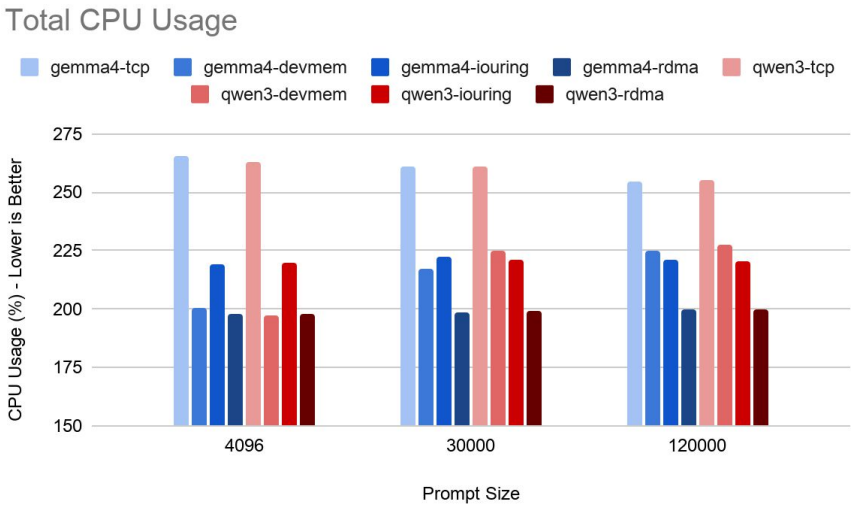
Transport Setup details

- Both RDMA and Devmem/iouring transports allocate a 1 GiB receive memory pool
  - Sizing the memory pool correctly is extremely important to achieve high performance
  - ~10% of the link speed buffering
- For devmem we batch release tokens for every receive system call in order to avoid calling multiple setsockopt() receive
- For RDMA we implement a credit flow control system using RDMA WRITEs for low latency (see back slides for details)
- Both Devmem and TCP async poll for readiness while RDMA busy polls completions
- Iouring follows the same logic of waiting for events by installing an unbounded recv
  - The kernel performs the equivalent of polling + recvmsg while userspace only processes CQEs
  - Note: We use io-uring devmem on RX and MSG_ZEROCOPY on TX
- Message sizes and pacing are calculated using the previous formulas
- Single Flow KV-Cache
- Ideal: ensure no drops or retransmits for all

Experimental Results

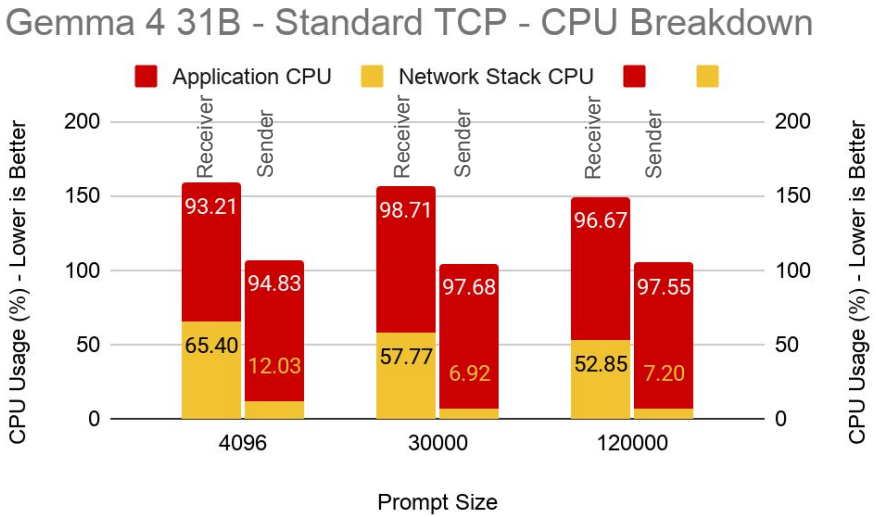
Performance Summary

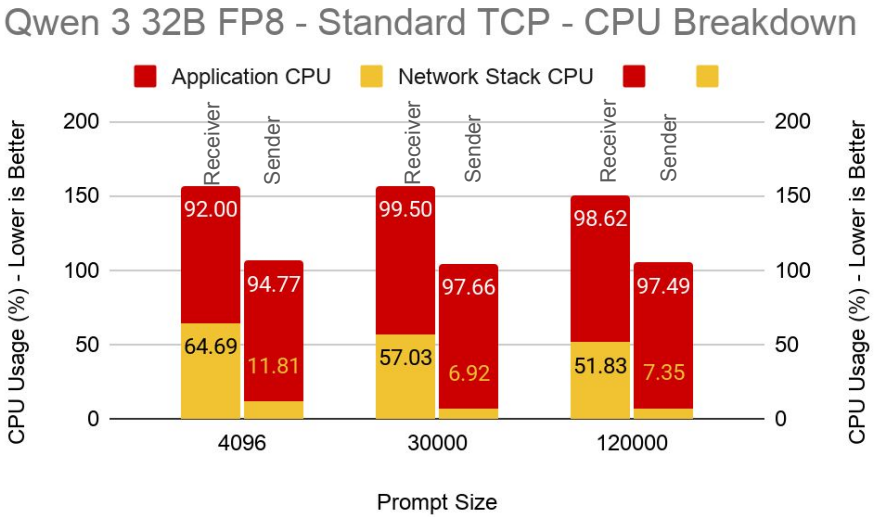




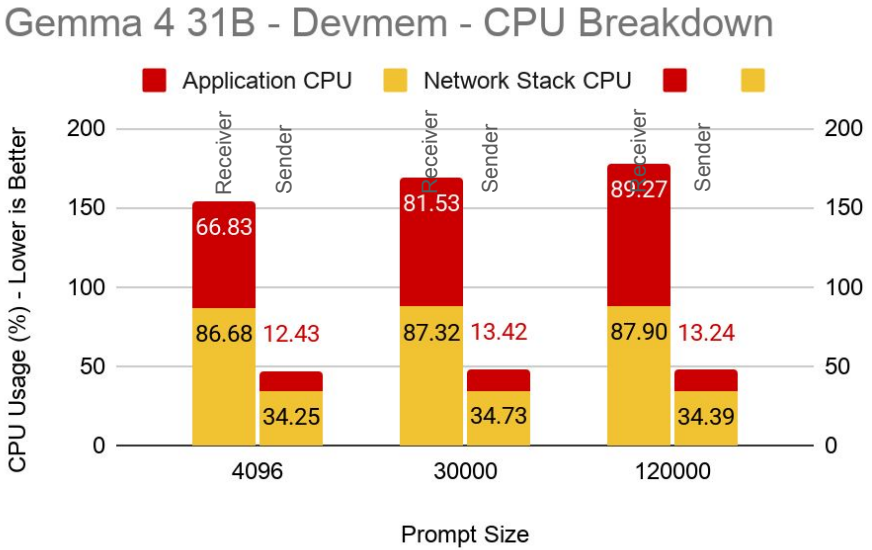
Gemma 4 31B / Qwen 3 32B FP8

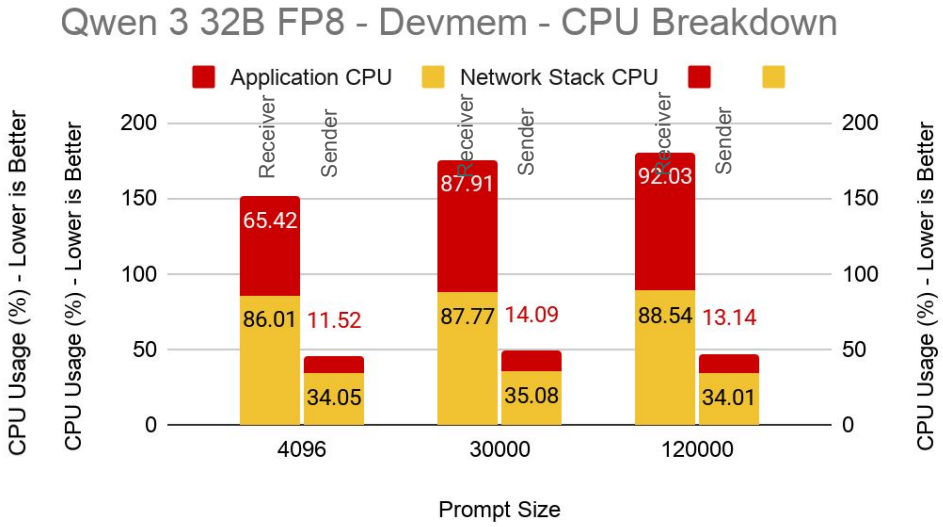
Sender/Receiver CPU Breakdown: Standard TCP



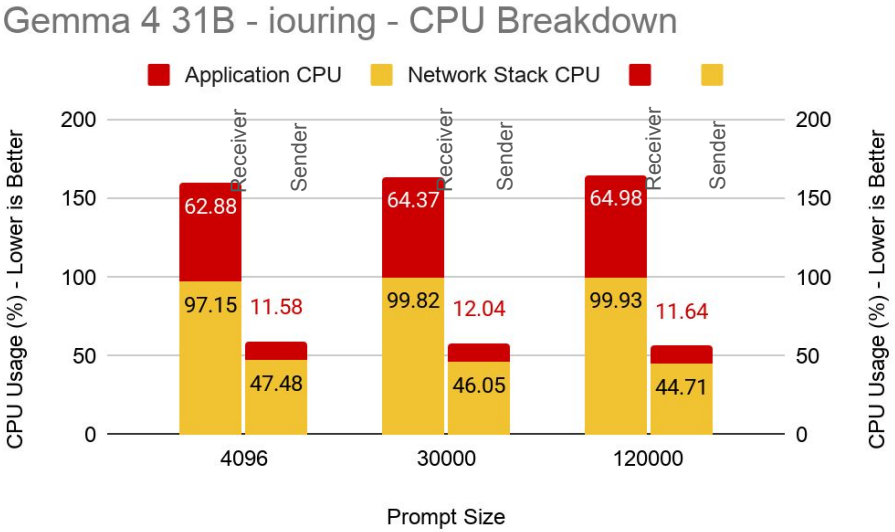


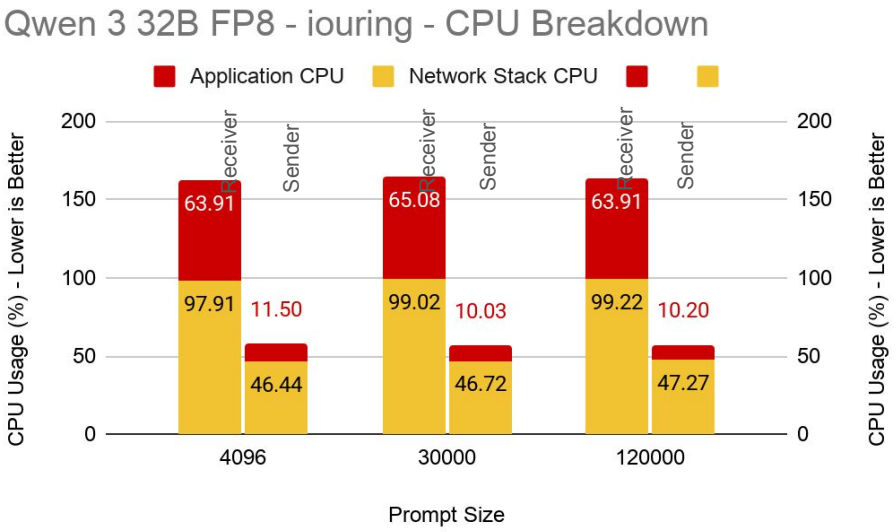
CPU Breakdown: Devmem



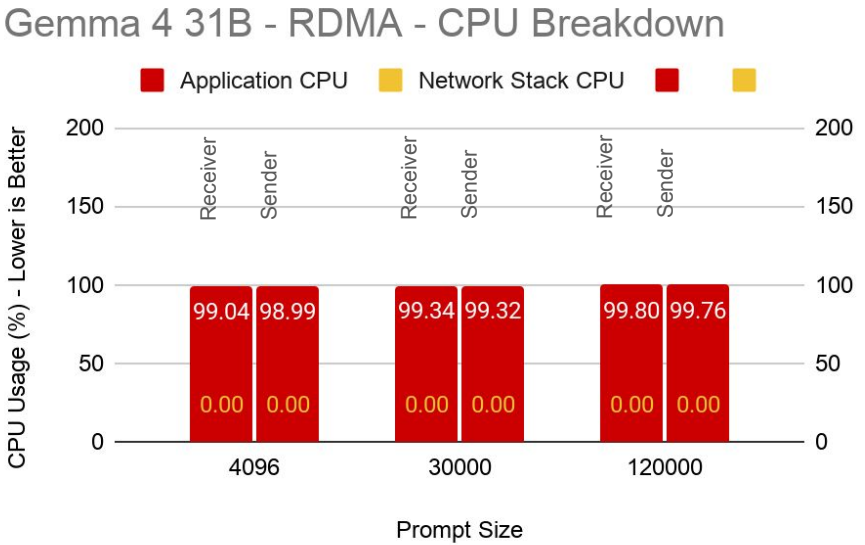


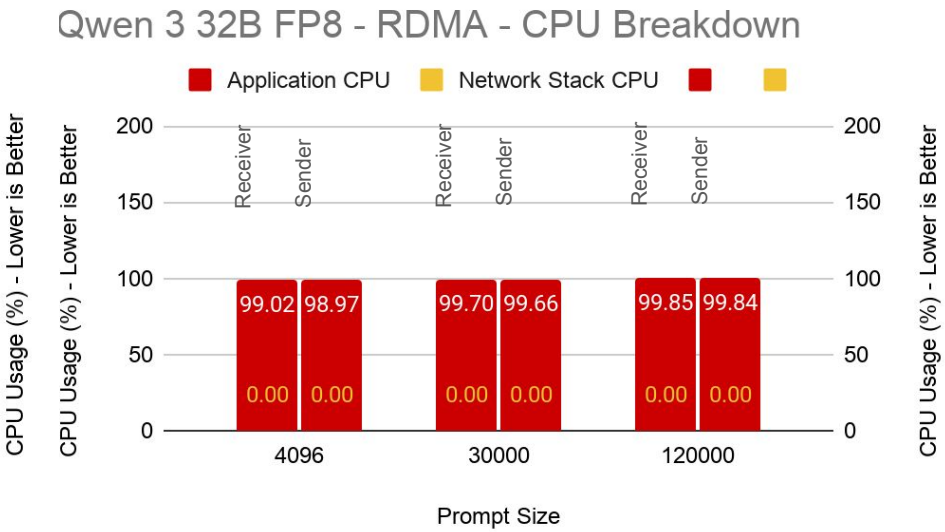
CPU Breakdown: iouring





CPU Breakdown: RoceV2 RDMA





Observation Summary

- Non-ZC TCP as expected is the least performant and deteriorates with increase in prompt size
- On ZC: both iouring and devmem TCP achieve the wire rate(100gbps) however iouring is observed to consume more CPU at lower prompt size (upto 20% at prompt size of 4096). We investigate this..
- In all the TCP cases, the receiver side is the bottleneck
- RDMA peaks at 45Gbps but throughput deteriorates at higher prompt size

Latency Measurements

Latency While Idle Setup

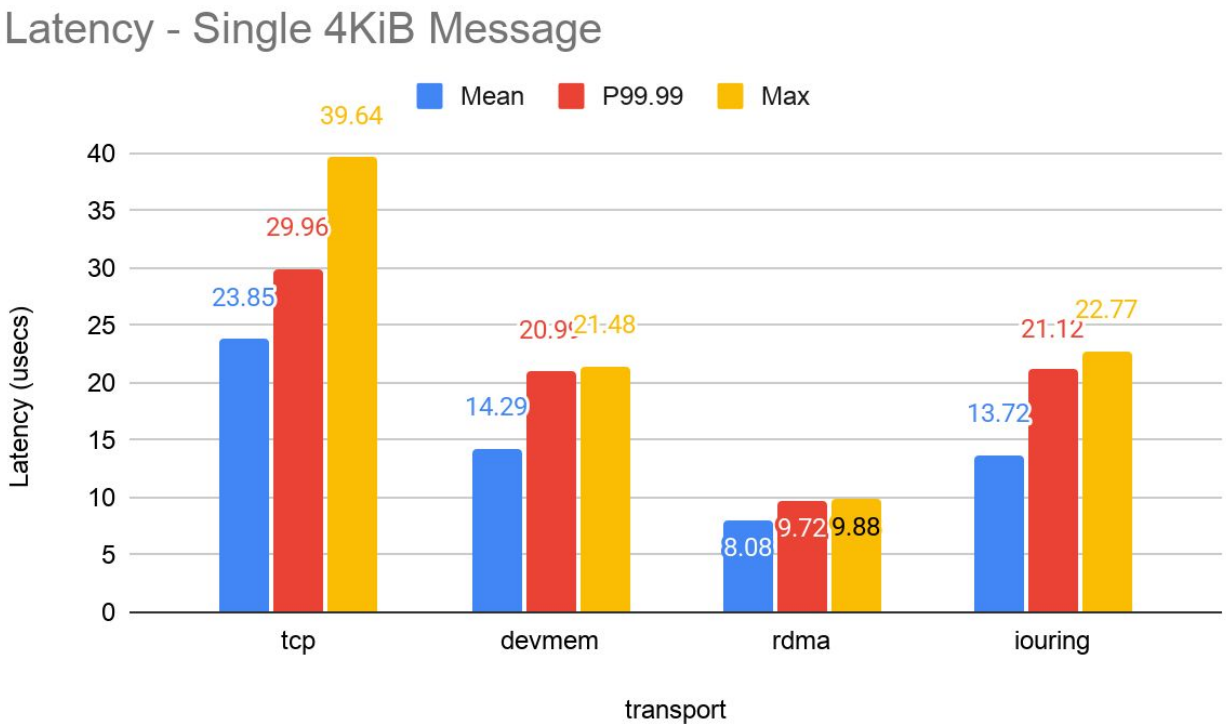
Single page transfer to measure how long it takes for the transport to transfer and receive a page visible to the server application

In order to precisely measure the latency, both sides have their system clocks synchronized with PTP

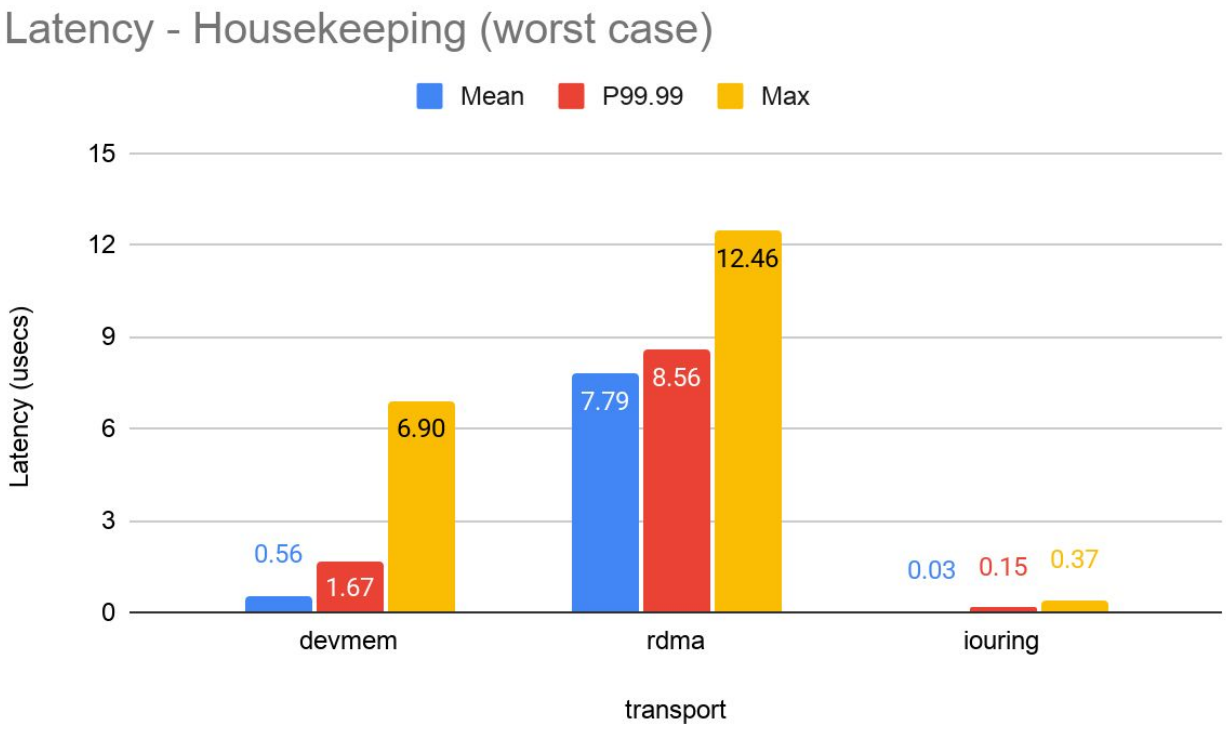
- The client writes a timestamp to the message and then the server takes out the difference

We run 21000 iterations of each test with 1000 iterations of warmup

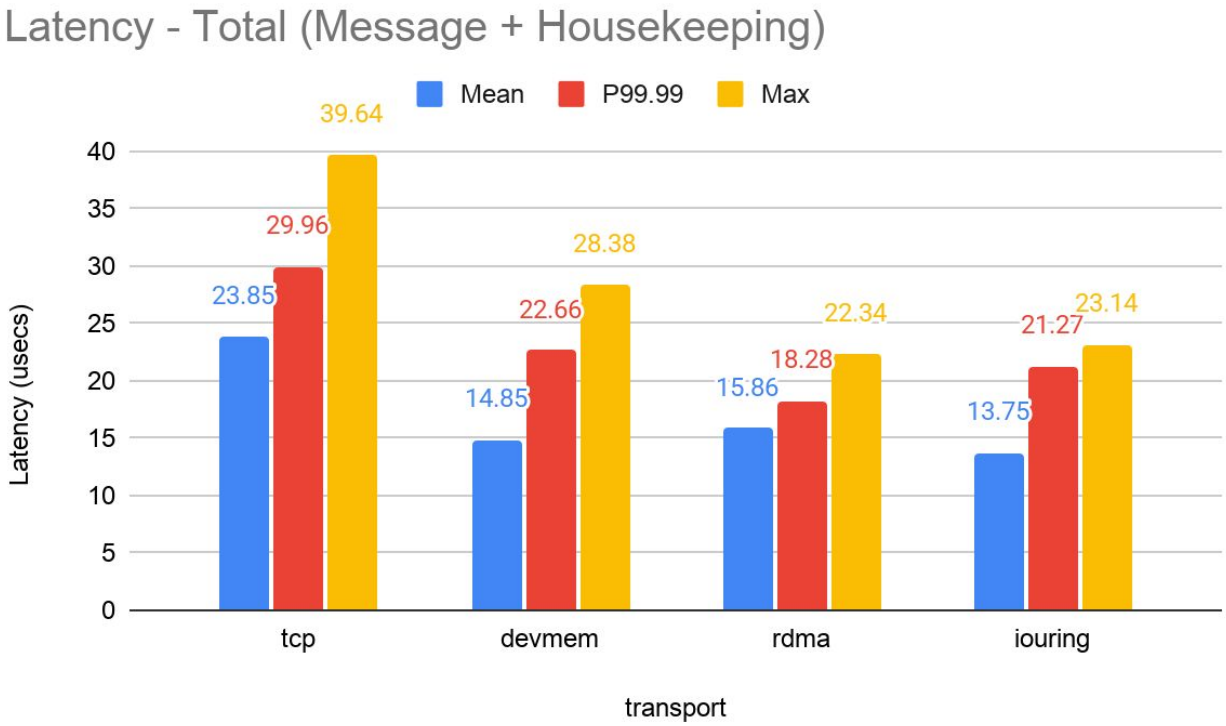
Latency While Idle



Latency While Idle Housekeeping Cost



Latency While Idle Including Housekeeping Cost



Discussion On Latency

As mentioned earlier: KVCache transfer involves a unidirectional “very large” bulk data transfer

- In a production systems with multi-user and batching involved the link will be occupied most of the time at full capacity
- Housekeeping overhead will be amortized

IOW:

Latency contribution on the results is minimal**

The bottleneck will be how fast the transport can get data to the other side

** for the workload under test (where the prefill GPU is constantly busy)

Recommendations Based On Results

From the experimental results we would recommend TCP Devmem

However, there are still open questions on (discussed in following slides..):

- iouring TCP with the strange CPU behavior
- Rocev2 with the low throughput and deterioration

https://www.facebook.com/groups/2616981278627207/posts/4118190835172903/

Challenges: Code Issues

There Were Dragons…

We are using the v4 of the following patch set:
(https://lore.kernel.org/all/20260224174618.2780516-1-aleksander.lobakin@intel.com/)

Issue 1: Page pool leak

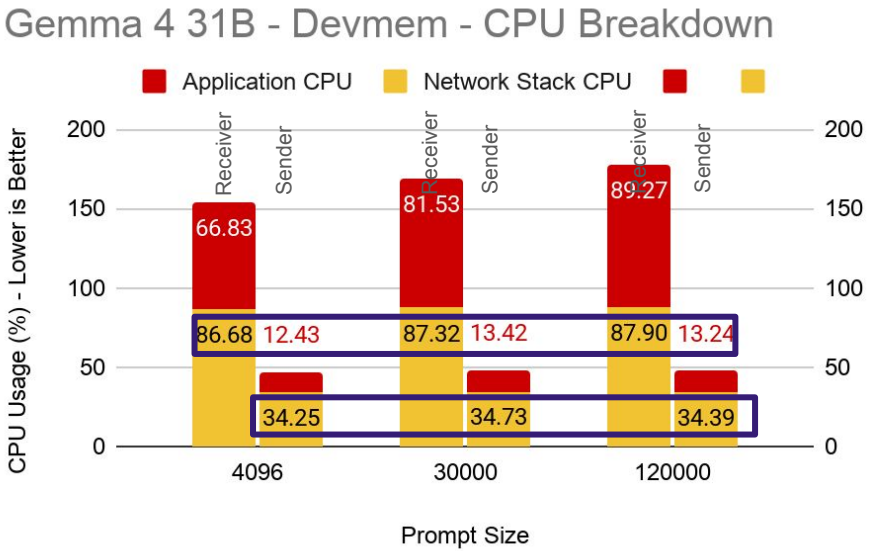
- fixed with a one liner

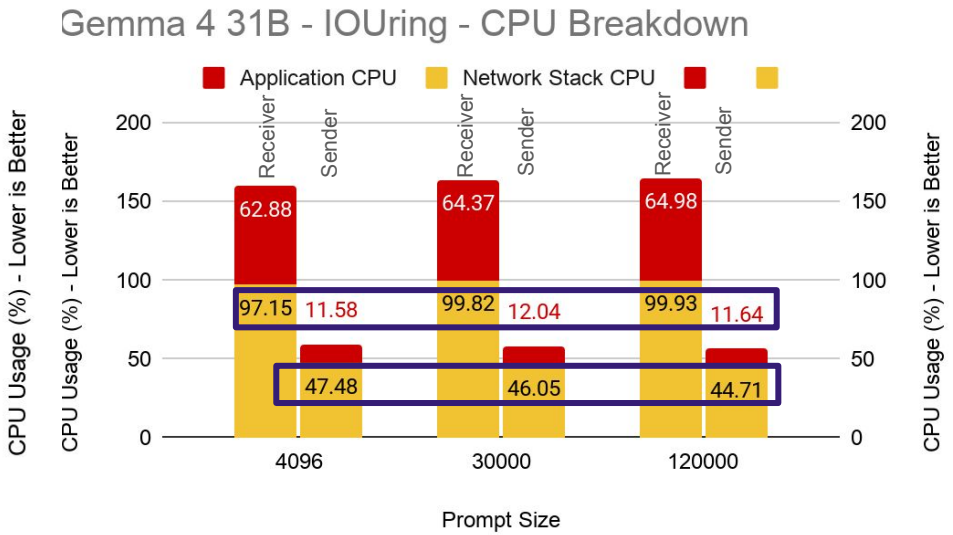
Issue 2: BUG Splat when registered area is less than RX descriptors

- not fixed, adapted user space code instead

Challenges: High SoftIRQ In IOUring

Mystery: Lower RX/TX Softirq for Devmem vs IOuring…





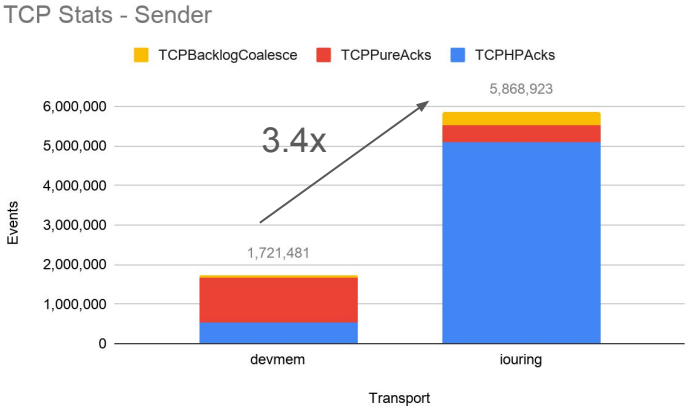
Recall: Achieved BW is the same at ~100Gbps....

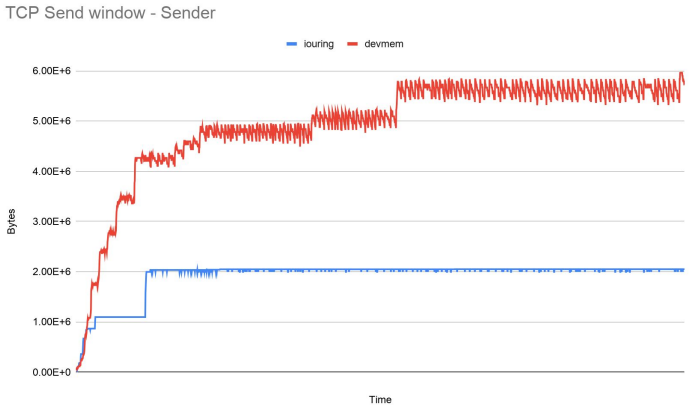
Zoning in on iouring recvzc…

Noticed that the number of ACKs generated by iouring was 3.4x higher than devmem and that the advertised window size stayed mostly flat

- Majority of iouring ACKs are ‘HPAcks’, which don’t carry any window updates
- Majority for devmem are ‘PureAcks’ carrying window updates (sawtooth pattern)

This higher ACK clocking on iouring is visible in the softirq CPU usage…





The “culprit”: DEFER_TASK_RUN

Feature helps us avoid the kernel interrupts of our program (after we noticed non-consistent latency) without it. Gave us:

- Explicit Control: Task work is no longer processed automatically on every kernel transition. Instead, it is deferred and only executed when app explicitly calls `io_uring_enter()` with the `IORING_ENTER_GETEVENTS` flag.
- Better Batching: You gain full control over when to process completions, making your processing pipeline much more efficient

IORING_SETUP_SINGLE_ISSUER
A hint to the kernel that only a single task (or thread) will submit requests, which is used for internal optimisations. The submission task is either the task that created the ring, or if **IORING_SETUP_R_DISABLED** is specified then it is the task that enables the ring through io_uring_register(2). The kernel enforces this rule, failing requests with **-EEXIST** if the restriction is violated. Note that when **IORING_SETUP_SQPOLL** is set it is considered that the polling task is doing all submissions on behalf of the userspace and so it always complies with the rule disregarding how many userspace tasks do io_uring_enter(2). Available since 6.0.

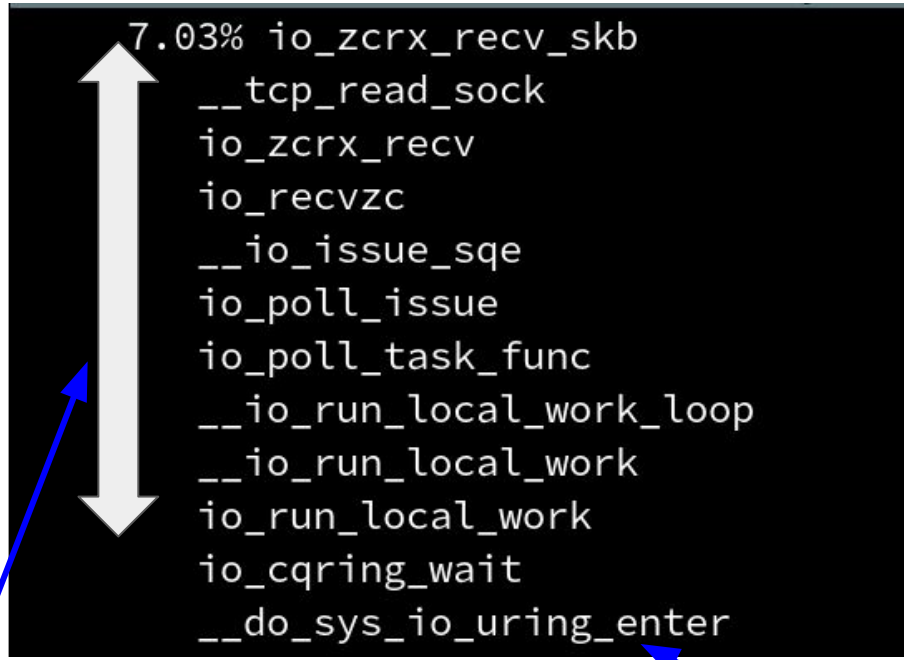
IORING_SETUP_DEFER_TASKRUN
By default, io_uring will process all outstanding work at the end of any system call or thread interrupt. This can delay the application from making other progress. Setting this flag will hint to io_uring that it should defer work until an io_uring_enter(2) call with the **IORING_ENTER_GETEVENTS** flag set. This allows the application to request work to run just before it wants to process completions. This flag requires the **IORING_SETUP_SINGLE_ISSUER** flag to be set, and also enforces that the call to io_uring_enter(2) is called from the same thread that submitted requests. Note that if this flag is set then it is the application's responsibility to periodically trigger work (for example via any of the CQE waiting functions) or else completions may not be delivered. Available since 6.1.

iouring recvzc DEFER_TASK_RUN

Observed that whenever the application waits for a cqes, it drains the socket receive queue via *tcp_read_sock*

- IOW, The application itself drives the generation of the notifications by reading from the socket

You could argue that you could do the same “batching/amortization” with devmem by deferring reads, but you dont get the non-interuptability effect



Processes TCP data immediately when syscall issued

Enter syscall

`io_uring_enter()`

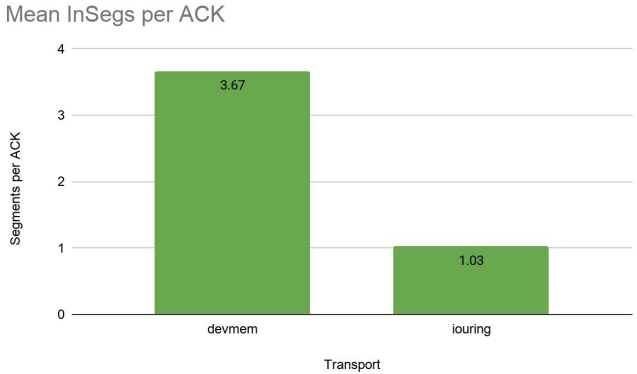
IOuring DEFER_TASK_RUN effect

We observe delayed acks in use…

- When app reads from the socket receive queue kernel checks if an ACK is pending to be sent and whether window update needed via *tcp_cleanup_rbuf*

GRO is in use

- Essentially on every iouring syscall entry we do a ~GRO sized read….
  - Immediately triggers the (deferred)ACKs
- devmem app issues the read (kicking the ack) after accumulating more data: every ~3.5 GRO-sized read…



iouring cleanup (bytes):

```
[4K, 8K)             523 |                                                    |
[8K, 16K)           1998 |                                                    |
[16K, 32K)         24795 |                                                    |
[32K, 64K)       5665595 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[64K, 128K)       183124 |@                                                   |
[128K, 256K)       15608 |                                                    |
[256K, 512K)           0 |                                                    |
[512K, 1M)             1 |                                                    |
```

devmem cleanup (bytes):

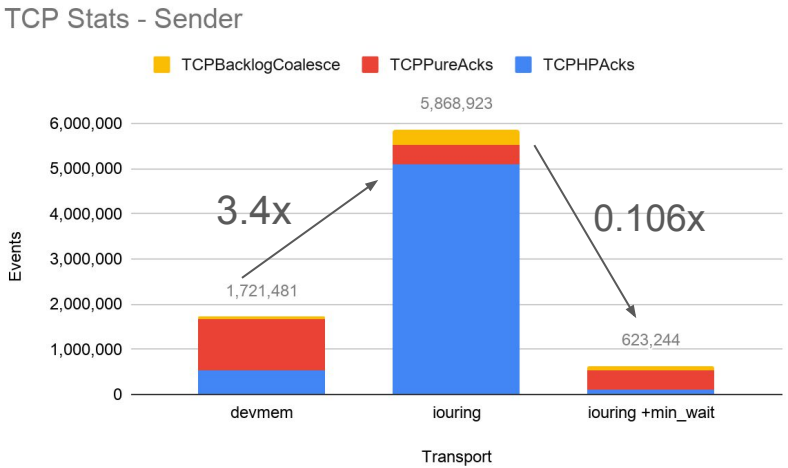
```
[4K, 8K)            5197 |                                                    |
[8K, 16K)            342 |                                                    |
[16K, 32K)          2206 |                                                    |
[32K, 64K)         30573 |@@                                                  |
[64K, 128K)       213761 |@@@@@@@@@@@@@@@@                                    |
[128K, 256K)      722140 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[256K, 512K)      545250 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@              |
[512K, 1M)          1889 |                                                    |
[1M, 2M)              12 |                                                    |
```

iouring: Can we delay but achieve same throughput?

Yes! *io_uring_submit_and_wait_min_timeout*

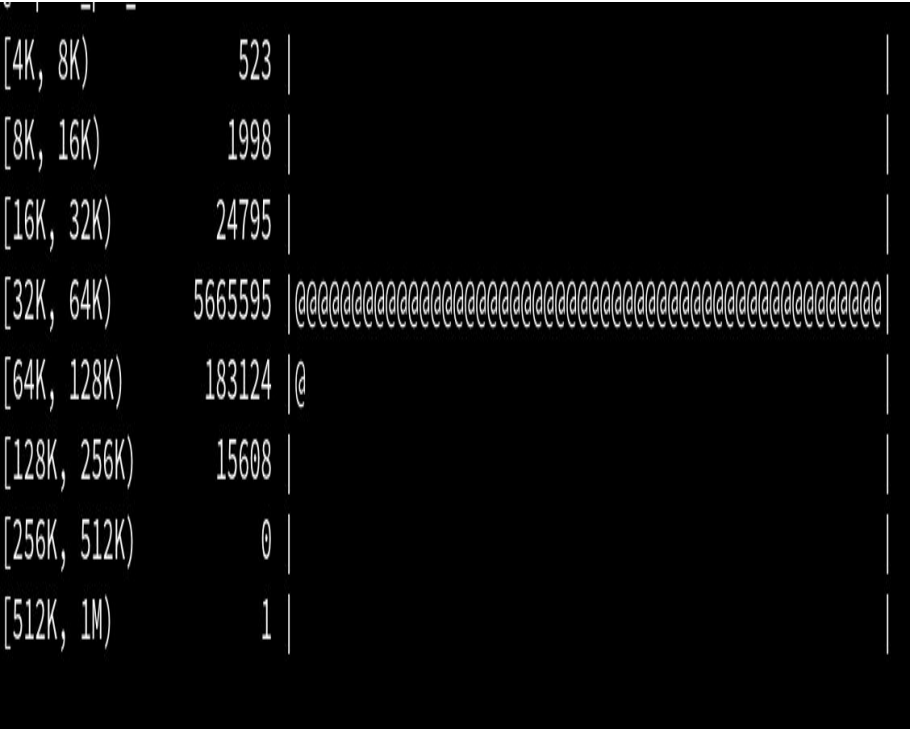
specify “wait for at upto 300 CQEs or 100 usecs”

Observation: after 100 usecs and <300 events, waits further 10 usecs for any new events…

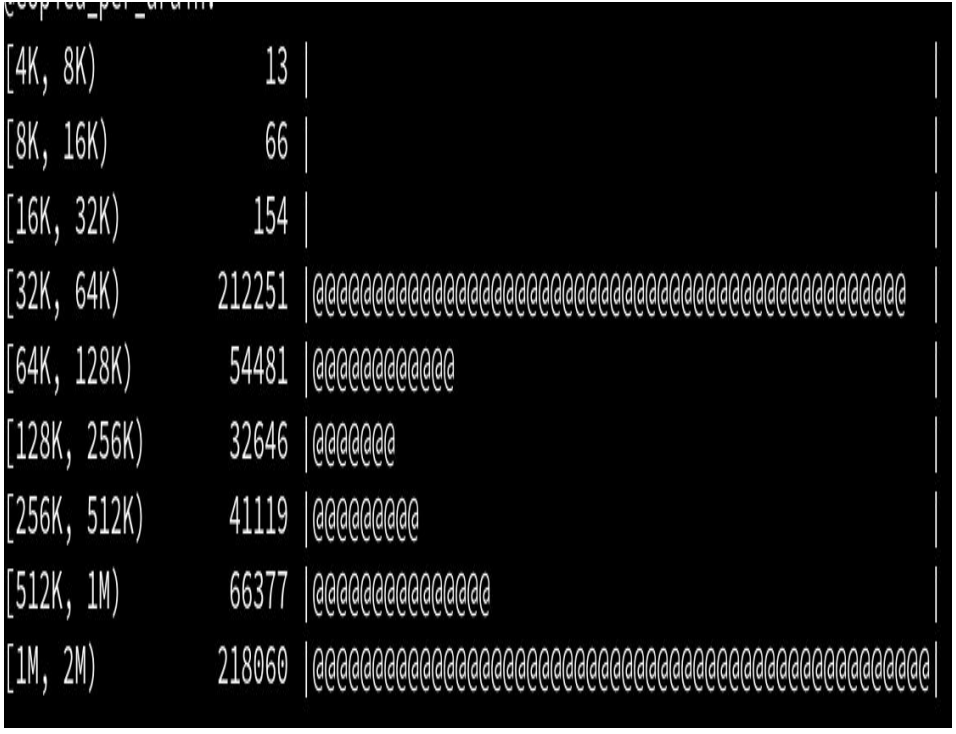


TCP: iouring recvzc side effect

Before

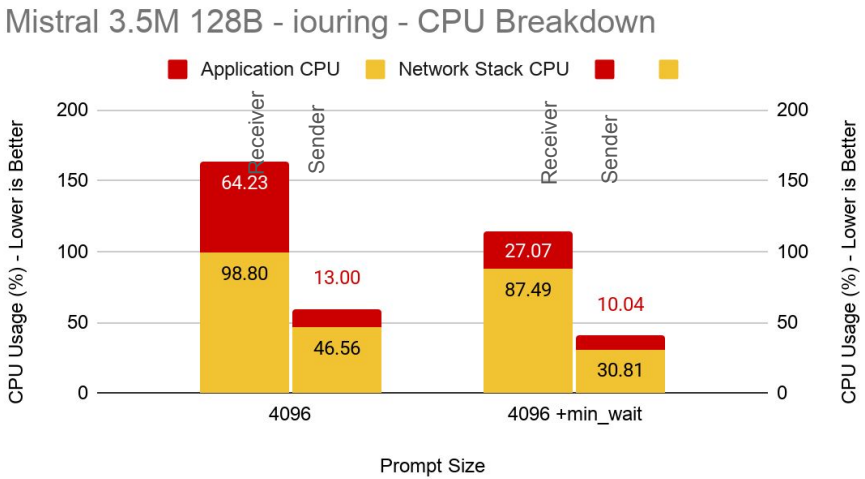


After



iouring recvzc side improvement

- Not only the client network CPU improved by 15%, the server application CPU decreased by half as TCP itself become lighter by not generating as many ACKs



App Before:

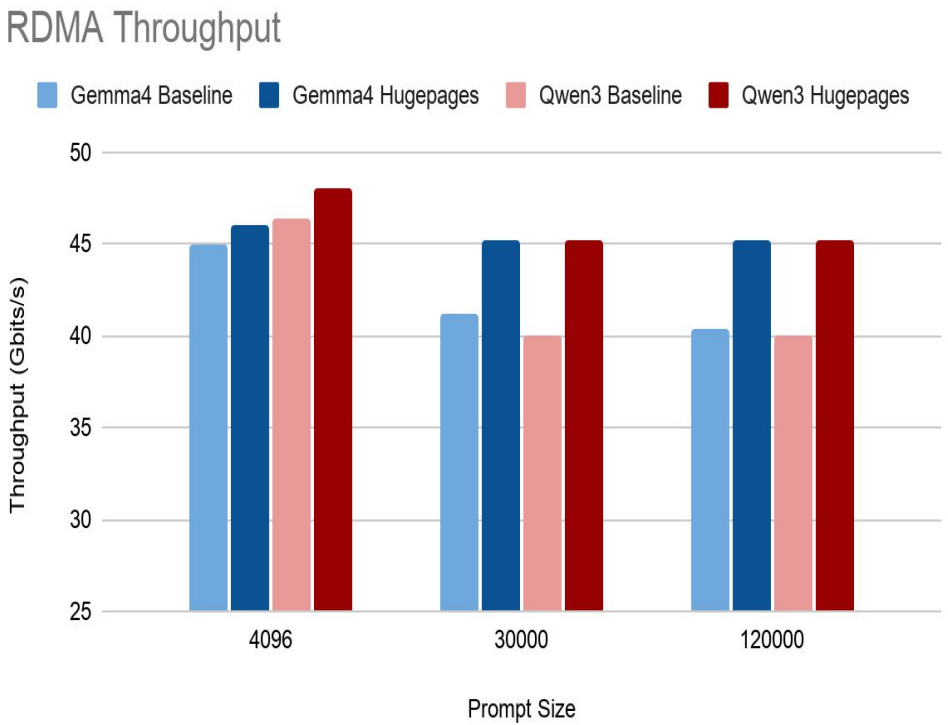
```
- 26.39% __do_sys_io_uring_enter
   - 26.24% io_cqring_wait
      - 23.73% io_run_local_work
         - 23.64% __io_run_local_work
            - 23.18% __io_run_local_work_loop
               - io_poll_task_func
                  - 22.94% io_poll_issue
                       __io_issue_sqe
                     - io_recvzc
                        - 22.88% io_zcrx_recv
                           - 20.29% __tcp_read_sock
                              + 9.25% __tcp_transmit_skb
                              + 8.79% io_zcrx_recv_skb
                              + 0.65% __tcp_send_ack.part.0
                                0.56% __sk_mem_reduce_allocated
```

App After:

```
- 6.43% io_cqring_wait
   - 6.32% io_run_local_work
      - 6.31% __io_run_local_work
           __io_run_local_work_loop
         - io_poll_task_func
            - 6.29% io_poll_issue
                 __io_issue_sqe
                 io_recvzc
               - io_zcrx_recv
                  - 5.73% __tcp_read_sock
                     - 4.32% io_zcrx_recv_skb
                          0.51% io_zcrx_queue_cqe
                       0.50% release_sock
```

Challenges: Rocev2 TLB Trashing

RoceV2: Issue Translating Memory Pages



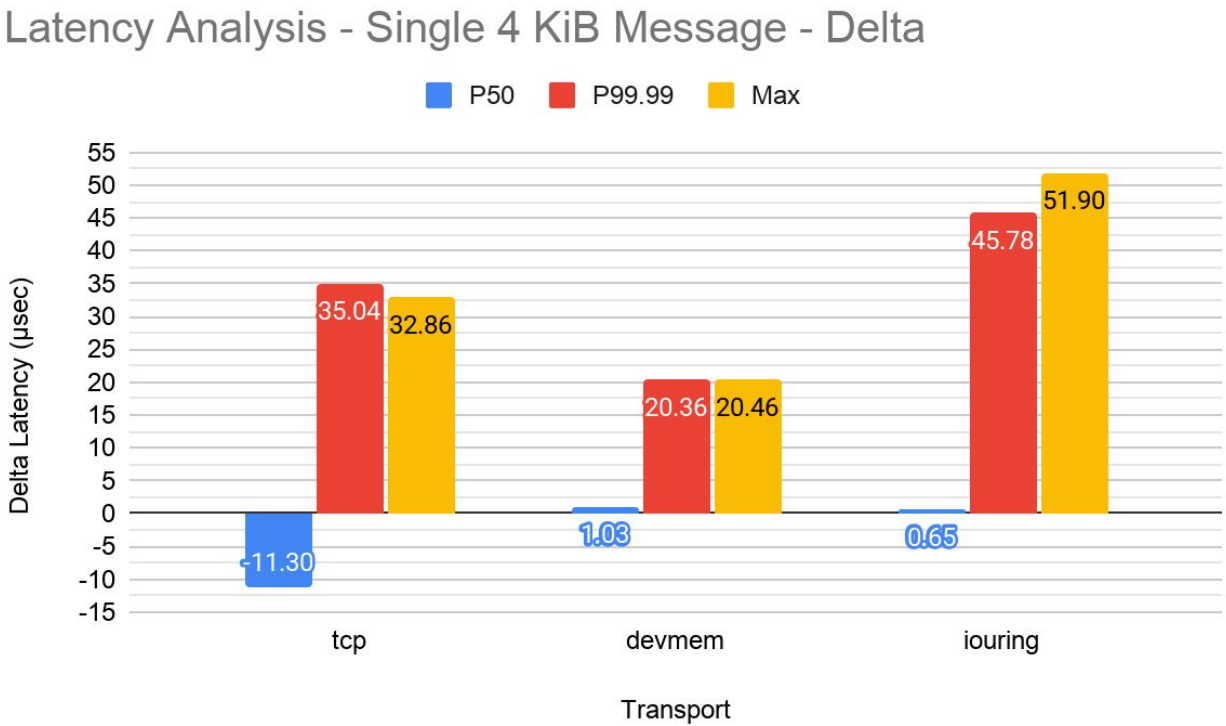
- Throughput deteriorating as the prompt size increased
- We discovered that our ICE hw achieves a better overall throughput if memory regions are using huge pages
- This indicates that the DMA engine in the ICE hardware performs much better when dealing with fewer pages, suggesting that we were in a TLB thrashing scenario

Challenges: Coalescing

Adaptive Coalescing considered harmful

- High throughput and predictable latency is essential for AI workloads
- In our experimentation we noticed how adaptive coalescing naively breaks these two requirements
  - First it makes the TCP transport more prone to drops in bursts, and therefore retransmissions, hurting predictability, throughput and latency
  - Second it makes tail latency unpredictable
- In this section we show results that led us to disable adaptive coalescing for the TCP transports

Adaptive Coalescing considered harmful



Conclusions And Recommendations

Discussions On KVCache Transport

Given the nature of KVCache transfer being a very large bulk unidirectional data transfer

TCP zc is well equipped to handle kvcache transfers

There are challenges in production:

1. TCP Devmem/iouring has worse tail latency compared to RoceV2 per message
    - Does it matter?
2. Writing to GPU memory will need some GPU-kernel maintenance effort (to “sort”)
    - This is where some of the efforts like Nvidia’s rocev2 incarnation, MRC and UEC deal well with (writing buffers to known address location)
    - It may not matter given GPU cycles are a-plenty
    - Tweak to avoid drops (as we did)
3. Adopting “Spraying” for a single flow will cause packet re-ordering (Trickery with DDP may help)
    - Wont matter if we can fill the link QP with a single flow
4. iouring adjustability gives good opportunity to “tweakability” but iouring semantics will be harder to integrate into standard AI infra lile NCCL etc (because those have very “strong” Socket semantics)

Rocev2 challenges (at least for the this NIC):

- we need >1 QP to achieve better throughput
    - Essentially, you need to have multiple decode endpoints(flows) to maximize throughput
    - Could be a bug or design intent

KVCache Transport Conclusions And Recommendations

In General

We believe ZC TCP is good enough for KVCache transfer

Out of the box configuration for Devmem/iouring/TCP is not good enough, so a good amount of tweaking is needed

- The most important tweak for this type of workload is to minimize drops..
  - Reminder: RoCEv2 tries to guarantee a lossless network with PFC
- Tweak
  - the socket buffers,
  - disable adaptive coalescing,
  - split the network cpu and the application cpu,
  - full cpu isolation from kernel background tasks
  - set governor to performance mode

In case of iouring, also keep an eye on TCP ack metrics and use the new forced batching API

KVCache Transport Disclaimers

Our tests are based on:

- A single vendor NIC. Other NICs may behave differently
- 100Gbps wire speed (we do a projection later on in this talk..)
- Tweaking so congestion and packet drops are removed when possible

The behaviors may change for any of the tested transports approaches at higher speeds

Projections:
If We Only Had Higher Link Speeds...

Projecting On Higher Link speeds

Given we dont have access to NICs with > 100Gbps we project our results to predict what would happen if the speeds were higher.

Disclaimer: Things may take a non-obvious turn when we run real experiments..

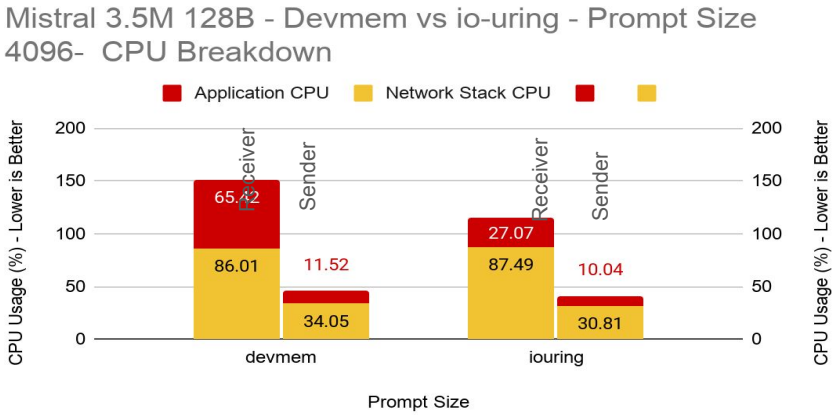
Projections on iouring and devmem

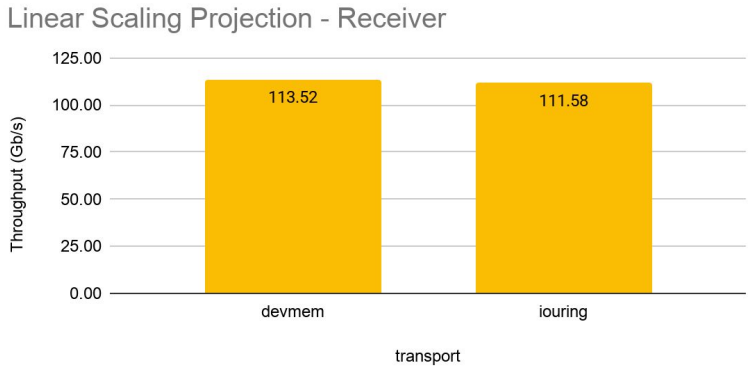
Receiver protocol processing is the bottleneck

Assumptions:

- 1 CPU for protocol and 1 application
- That we can linearly scale the cpu/bw
- And Ignore bottleneck between app and protocol processing
- 100% protocol cpu utilization

We hit about ~112 Gbps for iouring and ~114Gbps for devmem.
From experience HW GRO will improve this by another 25-30% -> 140Gbps per QP/flow.
Ideal case:
800Gbps achievable with RSS for 5 “flows”





Projections - iouring and devmem

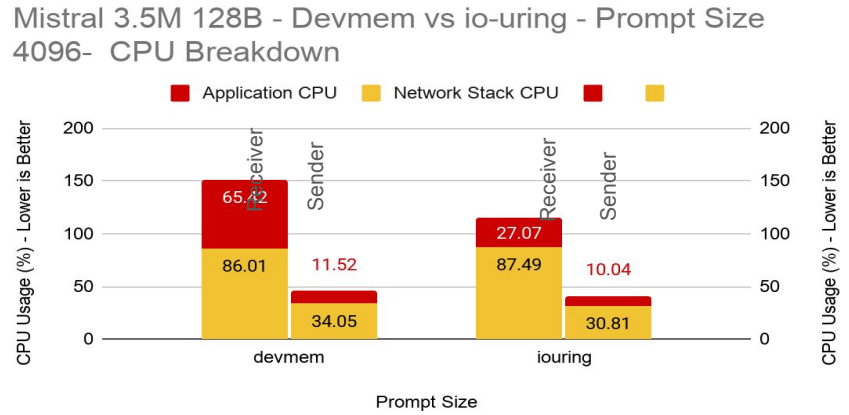
What if the protocol processing was ~zero?
(example totally/effectively offloaded)

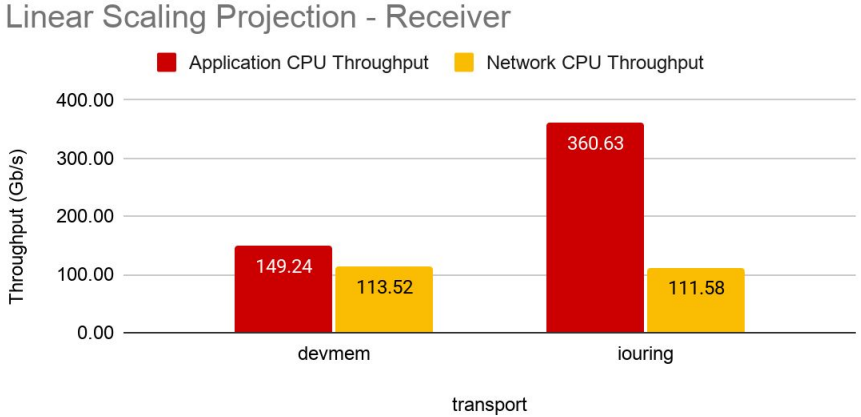
Second projection. Assumptions:

- We can linearly scale the app cpu/bw
- Ignore bottleneck between app and protocol

At 100% app processing:
149Gbps for devmem and 360Gbps for iouring
(assuming hw GRO)

Clearly io_uring scales better..





Projecting RoceV2

Unfortunately we dont have access to another NIC with Rocev2 support, so this ~45gbps upper bound per-QP could a design intent that expects >1 QP to be used or a bug or an ASIC limitation. A protocol like UEC or MRC would be able to do fine with multiple QPs without worrying about re-ordering. IOW, Spraying would improve things..

Rocev2 has a big advantage: The 1 core CPU utilization we use for polling (for the 1QP) would be good enough for many more QPs.

Ignoring unforeseen overhead: our projection for 20 QPs for 800Gbps says we would not need more than 1 CPU core for this management on either side..

As opposed to iouring and devmem where TCP protocol is processed on the CPU

Conclusions on Projections

Based on projections:

Throughput requirements:

- Any of Rocev2, TCP devmem and iouring will be able to handle the high throughput requirements (800gbps) for KvCache transfer

CPU utilization as a decision:

- Rocev2 due to its protocol offload will consume significantly less CPU
- Iouring is a better choice than devmem

Based on our observations: the toss up then is between iouring vs Rocev2.

- Depending on the NIC(whether Rocev2 is supported or not) and available CPU cycles, pick your poison

Backup Slides

Disaggregated serving

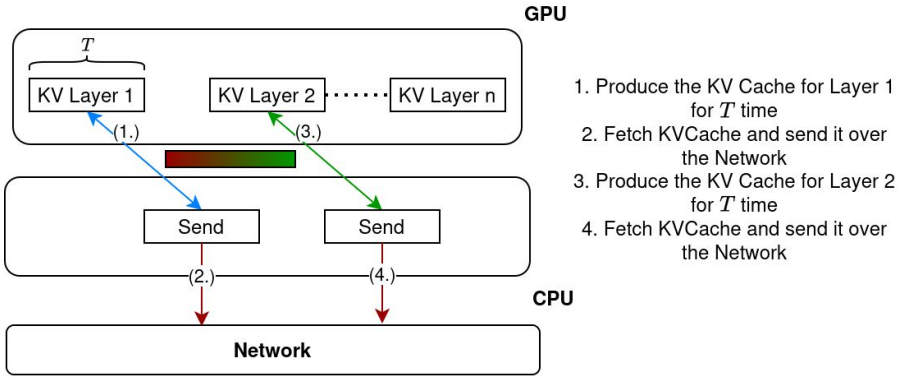
Transferring between prefill and decode can be done in 3 modes:

- ‘PUT’: Where the producer directly writes (blocking) the produced KV-Cache in the consumer instance
- ‘GET’: Where the consumer directly fetches (blocking) the produced KV-Cache from the producer’s storage
- ‘PUT-ASYNC’: Same as PUT however writes are non blocking, in this case the GPU is freed to produce the next KV Cache layer while transferring the previous layer

Transfer Technique

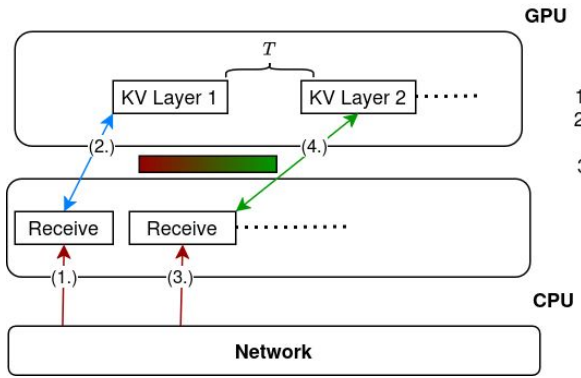
The simulation implements the ‘PUT-ASYNC’ transfer method overlapping the network transfer with the simulated computation whenever possible

Prefill side:



1. Produce the KV Cache for Layer 1 for T time
2. Fetch KVCache and send it over the Network
3. Produce the KV Cache for Layer 2 for T time
4. Fetch KVCache and send it over the Network

Decode side:



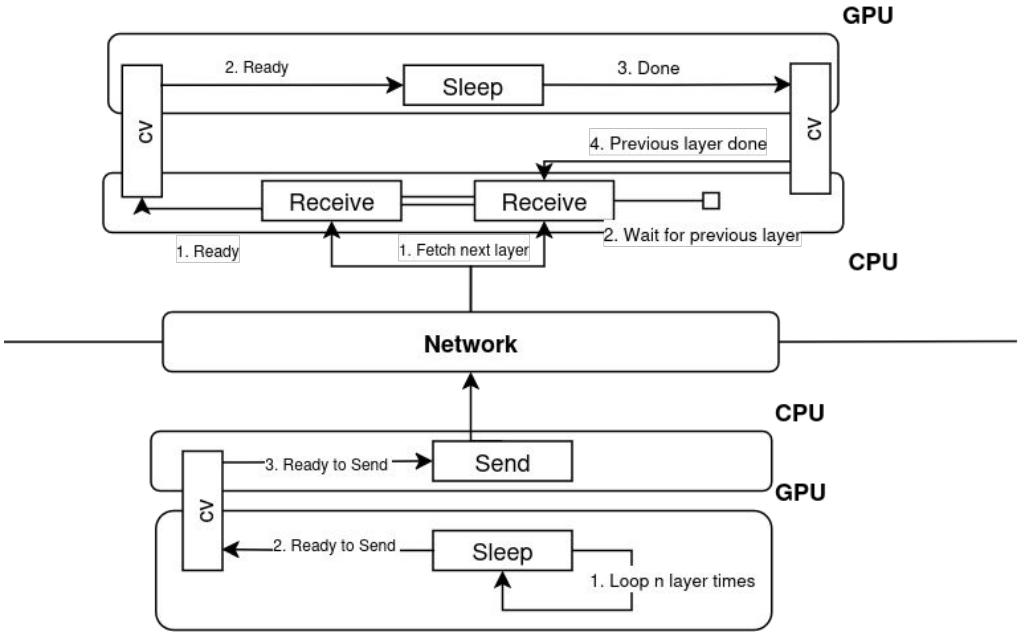
1. Receive the KV Cache for Layer 1
2. Fetch KVCache and prepare for T time
3. Receive the KV Cache for Layer 2
4. Fetch KVCache and prepare T time

Computation-Communication Overlap

For both cases, 2 and 3 happen simultaneously, however on the decode side we only pre-fetch up to the next layer while we let the prefill enqueue all layers as soon as possible to a background thread

Implementation Details

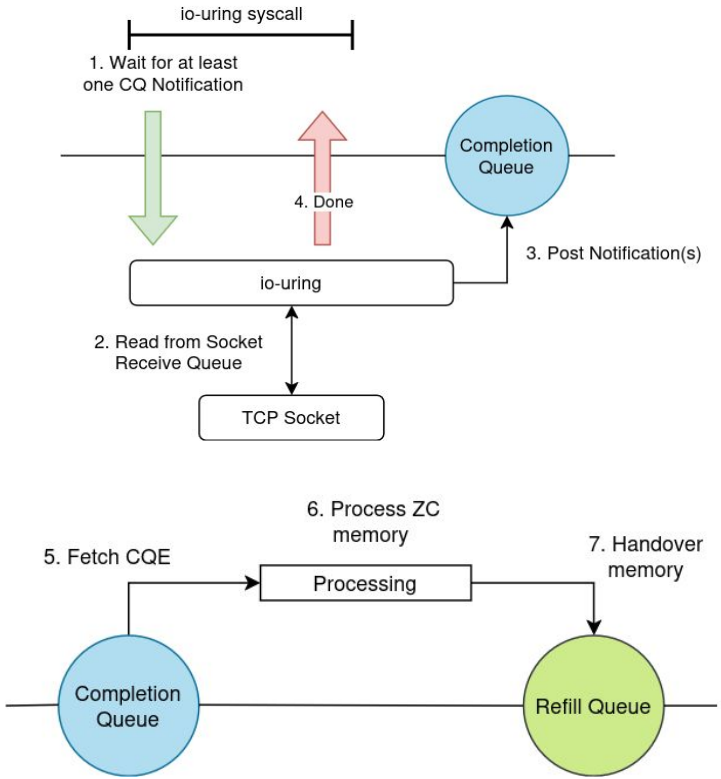
- In our implementation we simulate the CPU and GPU as two processes
- The GPU simulates computation by sleeping a N amount of time based on the model and target GPU passed as configurations
- The CPU handles the network communication and signals to the simulated GPU via a shared condition variable:
  - On send we immediately queue to the network the kv cache produced
  - On recv we have another queue to signal that the previous layer (n - 1) is done and a prefetch for the next layer (n + 1) can proceed



Recap on iouring..

App populates refill queue and submits a recv zerocopy operation to kernel

- App waits for a notification to be available (issues system call)
- Kernel interacts with the (internal) TCP socket
- Kernel populates CQ
- Kernel notifies app (completion of wait system call)
- App reads the CQ and processes the data
- App refills the refill queue



Our RDMA Implementation