
UCCL: AN EXTENSIBLE SOFTWARE TRANSPORT LAYER FOR GPU NETWORKING
By: Lekhit Nitin Borole

CREDITS:

Check-out: https://github.com/uccl-project/uccl

Yang Zhou

Zhongjie Chen

Ziming Mao

ChonLam Lao

Shuo Yang

Pravein Govindan Kannan

Jiaqi Gao

Yilong Zhao

Yongji Wu

Kaichao You

Fengyuan Ren

Zhiying Xu

Costin Raiciu

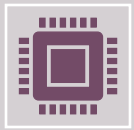
Ion Stoica

Gemini: for clearing my doubts

Microsoft power point for amazing design suggestions

QUESTIONS TO POUNDER UPON

"Control Coalescing," where decisions are made on 32KB chunks to save CPU cycles. While this improves efficiency, how does this coarser granularity affect the system's ability to react to very rapid, sub-chunk-level network congestion?



Design relies on the host CPU for the control plane. As NIC speeds increase to 800 Gbps or 1.6 Tbps, how does the CPU requirement for UCCL scale? Is there a risk that the host CPU could become the new bottleneck in the system?

Highlight that UCCL can solve the incast problem for workloads like Mixture-of-Experts (MoE) using receiver-driven congestion control. Could you elaborate on how UCCL's software approach provides a more effective or tunable solution for this compared to hardware-based congestion control mechanisms?

BACK GROUND: TYPES OF CONNECTIONS

Reliable Connection (RC): Too rigid to evolve and hard to control with software methods.

Unreliable Connection (UC): Just the right amount of flexibility. The hardware handles the data transfer part. Where as software is free to explore control plane.

Unreliable Datagram (UD): Best to avoid unless UC is not available. (comes with a lot of complications!!)

THE CORE IDEA?

- You know your application requirements better than ‘one-size-fit’ solution that NIC offers.
- Hardware evolves slowly compared to software.

KNOWN LIMITATIONS: MOTIVATIONS!!

RECEIVER DRIVEN CC FOR INCAST FLOW

Deepseek MOE and average exceeding 10X standard workload.

APPLICATION TRANSPORT DESIGN

Add reliability to things that actually need it!!

INEFFICIENT LOSS RECOVERY

The problem with go-back-N (Limits of SRAM)

HETEROGENEOUS NIC

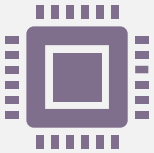
Reduces bandwidth by up to 33X says (alibaba ref: 2.2)

PRIOR WORK

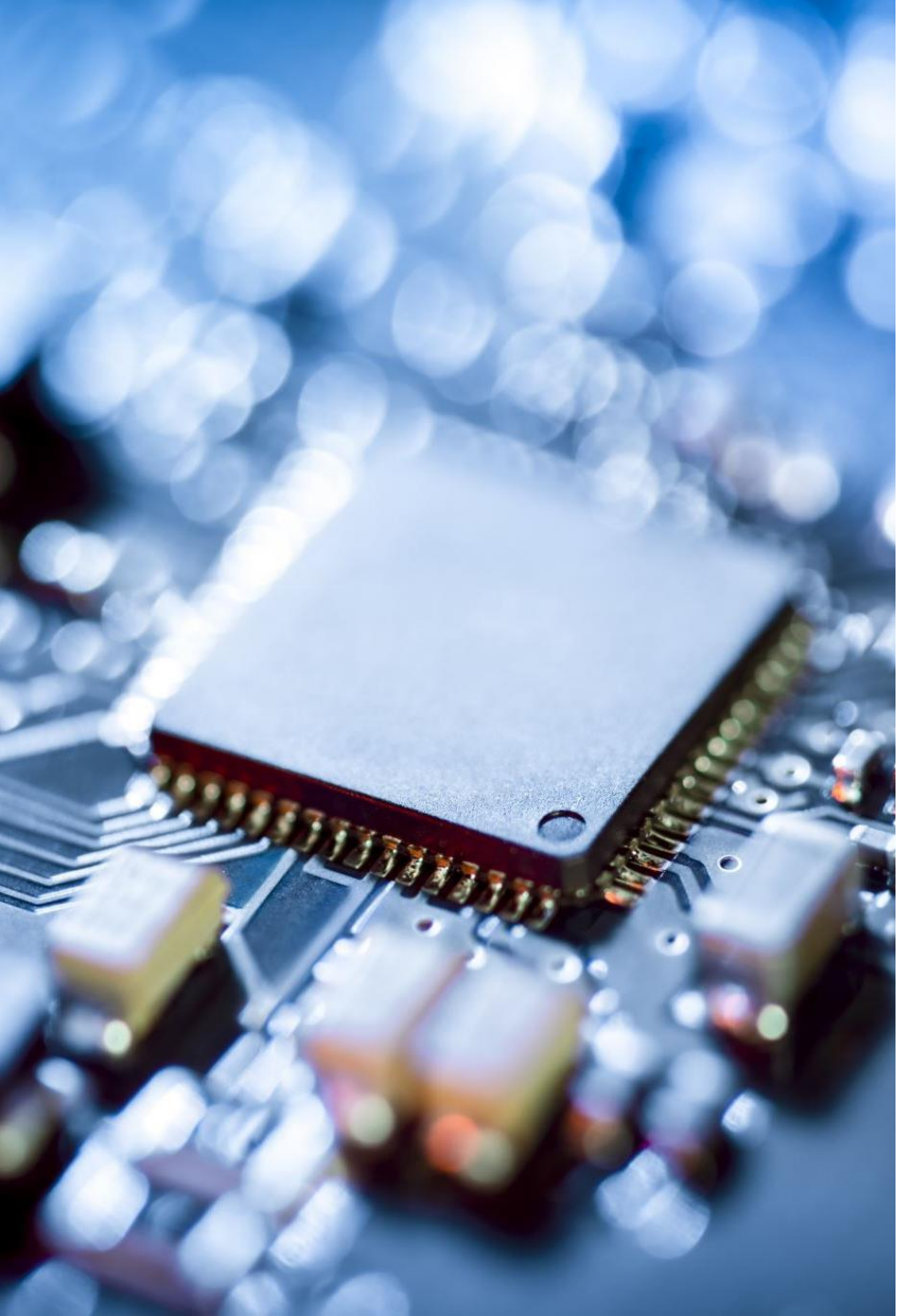
SMART NICS

Supports programming capability

Remember: U know your application data flow better!!



Similar to UCCL: Though UCCL leverages CPU (better usage of commodity hardware)



CPU-S

- Flor tried, but for 100GB flows, we are dealing with 3.2TBs
- UCCL added support for multipathing

UCCL DESIGN

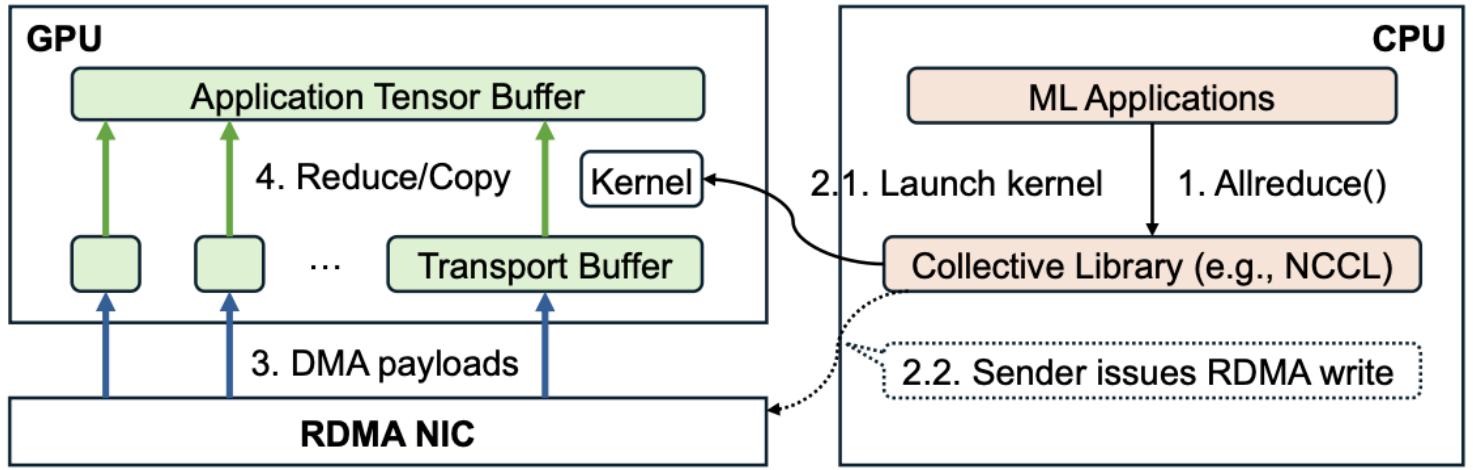
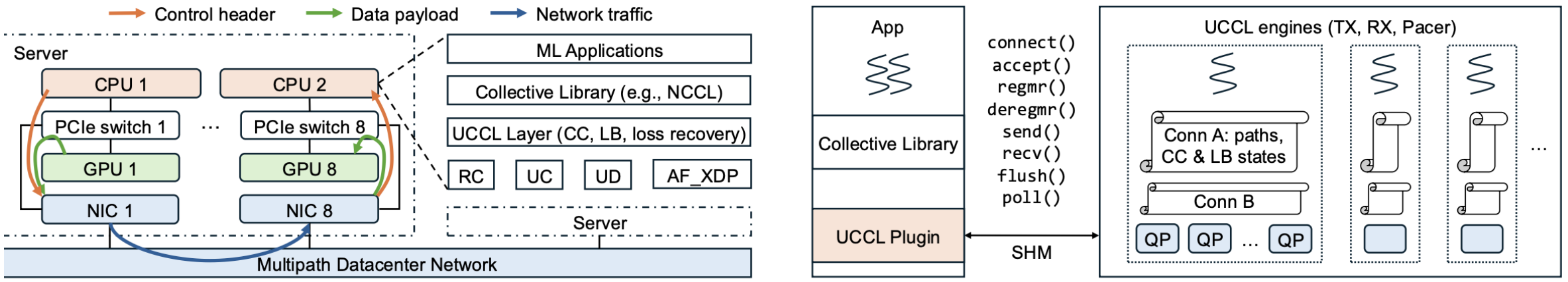


Figure 1. Collective communication over RDMA (receiver side). The receiver directly receives data into the GPU memory (e.g., GPUDirect [27]); the sender side works similarly except that it will additionally issue an RDMA write in step 2.



(a) UCCL architecture.

(b) UCCL threading model.

Figure 2. Overview of UCCL extensible transport for GPU networking. We assume a common intra-server topology for GPU servers [22] where individual PCIe switches directly connect a few GPUs, NICs, and CPU, providing high-bandwidth data transfer among them.

IDEAS

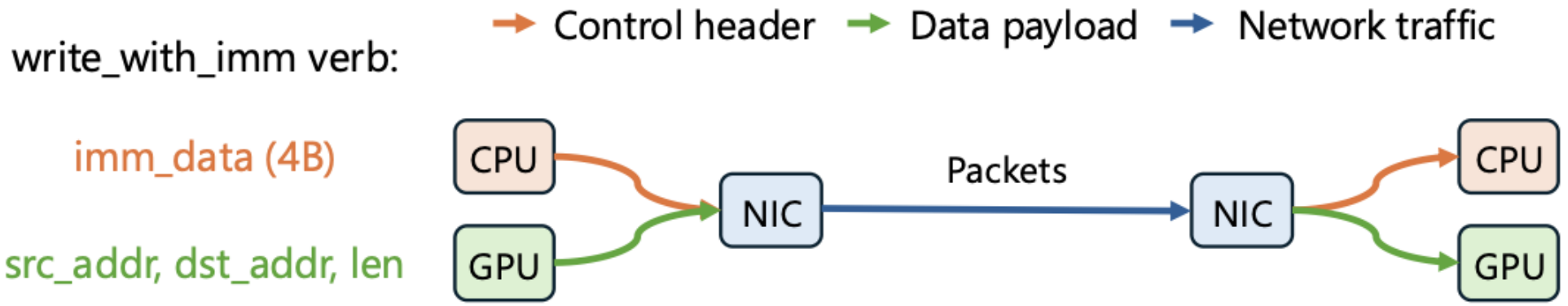


Figure 3. Leveraging RDMA `write_with_imm` to separate control header and data payload for UC/RC.

UC → LIKE POINTED OUT, PREFERRED!! WHY?

- It takes care of data transfer
- Faster than MMIO --> GPUDirect
- Splits packages and reassemble without CPU cycles

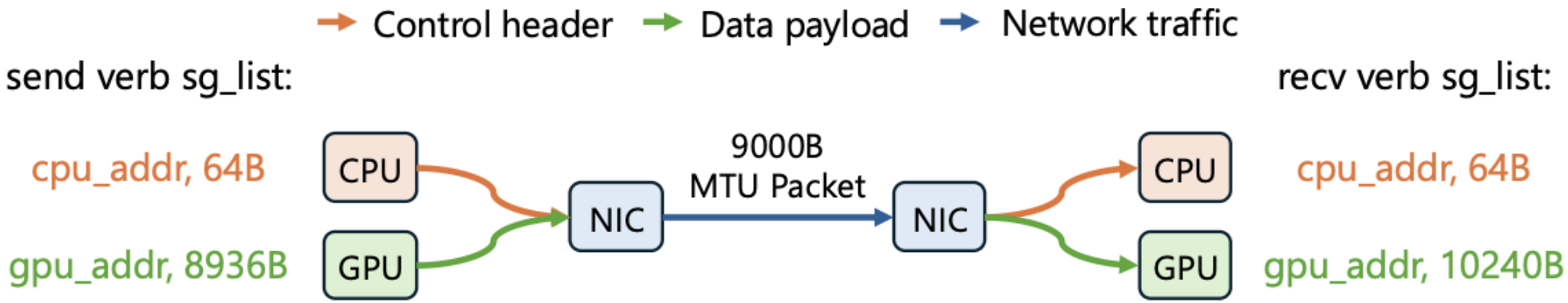


Figure 4. Leveraging RDMA send/recv scatter-gather to separate control header and data payload for UD.

UD → WHEN UC IS NOW AVAILABLE WHY?

MTU (memory transfer unit) limits size of data

Segmentation of data managed by application (u!!)

Consumes CPU cycles for package reassembly

HARNESSING MULTIPATH → ECMP (EQUAL COST MULTI-PATH)

- UC & RC → Default 256 QP
- UD → 16
  - Comes with caching problems
  - Supports one to many connections
  - Handling out of order packets

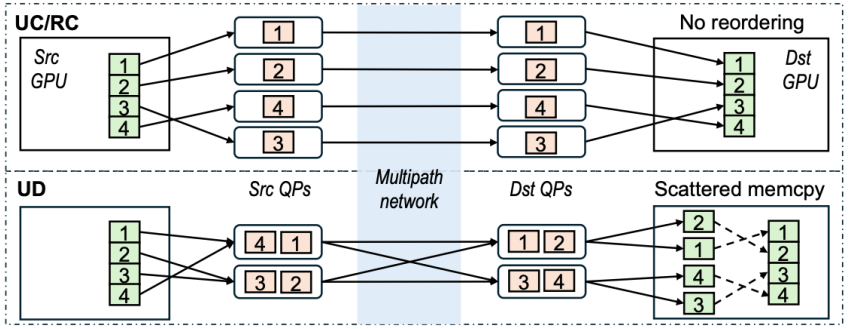
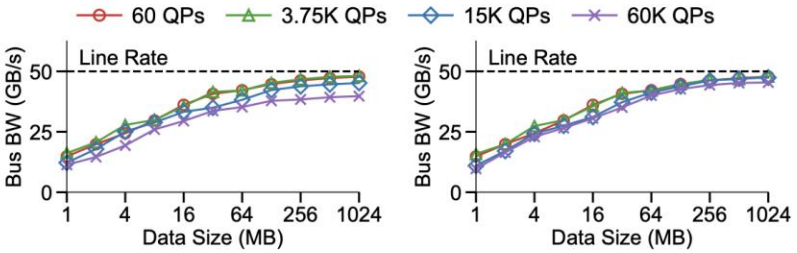


Figure 5. Multipathing and handling packet reordering in UCCL.



(a) For RC QPs. **(b)** For UC QPs.


TOWARDS
EFFICIENT
SOFTWARE
TRANSPORT

RUN TO COMPLETION EXECUTION

Deficit round robin → avoid starvation.

CONNECTION SPLITTING

Partition 256 QP (default) among thread + cores

CONTROL COALESCING

Chunking 32KB (default) so decision can be taken for larger chunks

Saving CPU cycles!! Reducing overhead

CHAINED POSTING → UD SPECIFIC

MMIO is involved so CPU cycles are needed

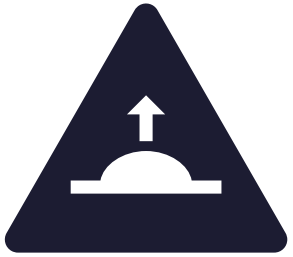
Smart way to avoid, make chain of 32 verbs!!

CONGESTION SIGNAL

Since we can’t use NIC level CC in control plane control. We need to rely on other methods.

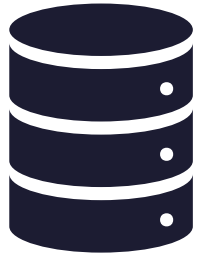
One way is to Use RTT (round trip time)

SOFTWARE CONTROL



Challenges??

Time in processing is high
Delayed response to congestion?



Solution??

Dedicated high priority QP for RTT
CPU polls its Completion queue.

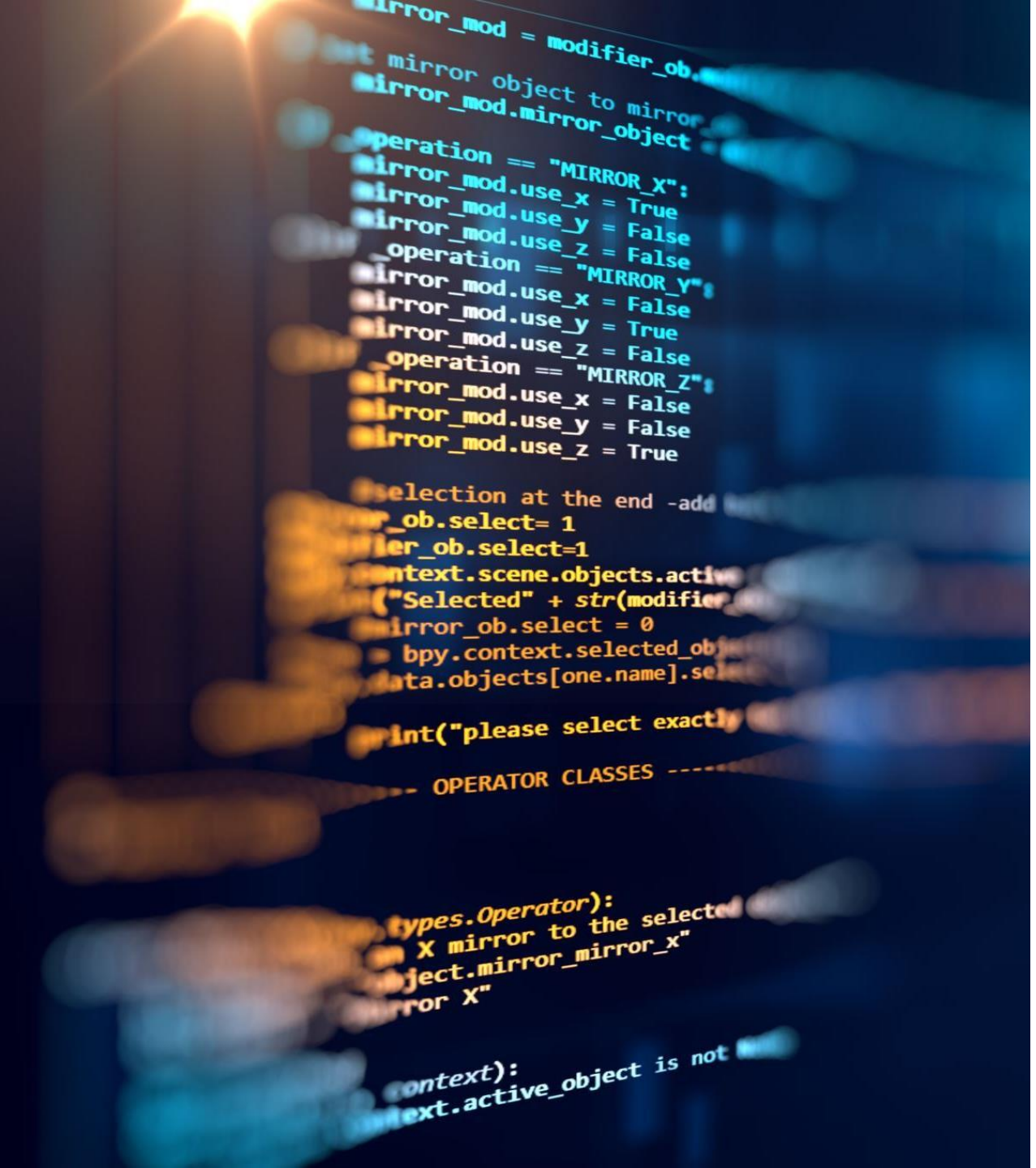
ADDRESSING PROBLEMS WE STARTED WITH: EXTENSIBILITY CASE STUDY

PACKET SPRAYING

- We need an effective way to address flow collisions in ML workloads
- How?
  - Using multiple paths effectively
  - Difficult for hardware, easily manageable for UCCL.

RECEIVER DRIVEN CONGESTION CONTROL

- Here the problem with MOE faced by Deepseek can be addressed.
- Where we can implement the receiver Driven Congestion control using UCCL



EFFICIENT LOSS RECOVERY

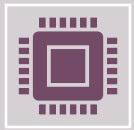
- Nic Default to Go back N
- Better loss recovery with software!!

THANK YOU!


QUESTIONS?

QUESTIONS!!

"Control Coalescing," where decisions are made on 32KB chunks to save CPU cycles. While this improves efficiency, how does this coarser granularity affect the system's ability to react to very rapid, sub-chunk-level network congestion?



Design relies on the host CPU for the control plane. As NIC speeds increase to 800 Gbps or 1.6 Tbps, how does the CPU requirement for UCCL scale? Is there a risk that the host CPU could become the new bottleneck in the system?

Highlight that UCCL can solve the incast problem for workloads like Mixture-of-Experts (MoE) using receiver-driven congestion control. Could you elaborate on how UCCL's software approach provides a more effective or tunable solution for this compared to hardware-based congestion control mechanisms?