SpCCL: A Sparsity-Aware Collective Communication Library for GPU Platforms

Lannie **Dalton** Hough, Emir Gencer, Hoffmann Muki, Abhinav Bhatele

PARALLEL SOFTWARE AND SYSTEMS GROUP

UNIVERSITY OF MARYLAND

Motivation

- **Observation 1:** distributed sparse workloads are common
  - Gradient-pruned DDP (natural or induced), distributed GNN training, HPC apps
- **Observation 2:** standard collective libraries do not exploit sparsity

- **High-level goal:** produce an efficient sparsity-aware collective communication library

Challenges

- **Irregularity:** "sparse data" covers many different levels and patterns of sparsity
- **Dynamic sparsity:** reducing collectives densify data across collective steps
- **Conversion overheads:** (de)compression time cuts into savings from sending less data

```
Rank 0:  [3, 0, 1, 0]  (50% sparse)
+
Rank 1:  [0, 2, 3, 0]  (50% sparse)
=
Result:  [3, 2, 4, 0]  (25% sparse)
```

Format Selection

- **Index-based:** CSR, COO
- **Structured:** BSR, DIA
  - Overhead depends on structure, but can be substantial
- **Bit-Vector (BV):**
  - Implicitly encodes nonzero positions
  - 3.125% metadata overhead relative to dense

"ID" COO:

```
Dense   : [0, 7, 9, 0]
Indices : [1, 2]
Values  : [7, 9]
```

Bit-Vector:

```
Dense   : [0, 7, 9, 0]
BV      : 0110
```

Challenge for BV

- **Serial decompression:** scan bitvector from start and track *nnz* encountered
- **Parallel decompression:** threads start from arbitrary position in bitvector

```
Orig. Dense :  [0, 2, 0, 0, 3, 0, 1, 0]
- - - - - - - - - - - - - - - - - - - -
Bitvector   :  [0  1  0  0  1  0  1  0]
Nonzeros    :  [2, 3, 1]
- - - - - - - - - - - - - - - - - - - -
Decompressed:  [0, 0, 0, 0, 0, 0, 0, 0]
```



Thread 1 knows a nonzero goes in position 4, but not the corresponding position in nonzeros array

Solution: Auxiliary Indices

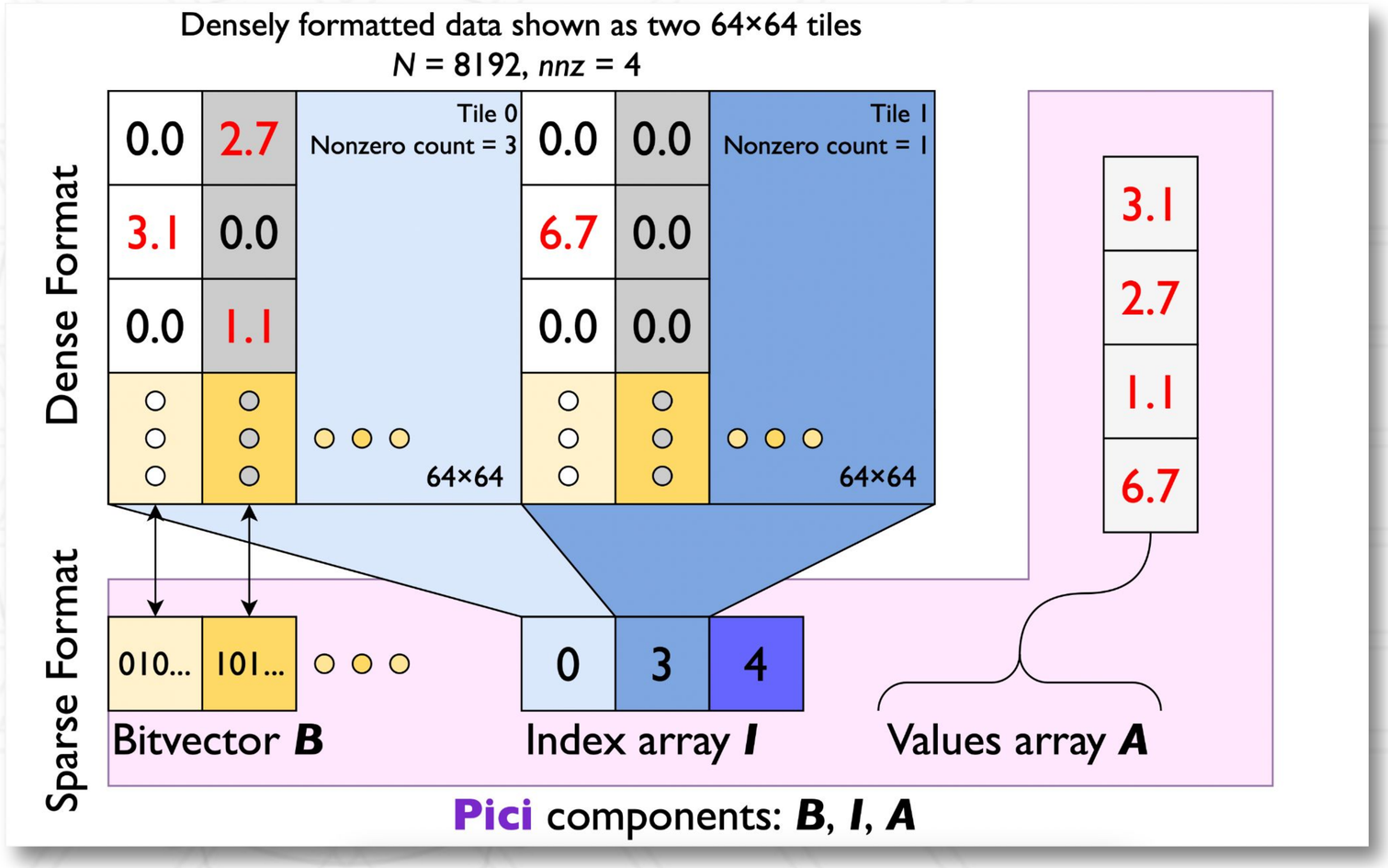
- Store index for each span of *k* elements, thread can process from start of any span
- Our format, **Pici**, does this for ***k* = 4096**, taking inspiration from field of succinct data strutures

example for *k* = 4

```
- - - - - - - - - - - - - - - - - -
Bitvector   : [0  1  0  0  1  0  1  0]
Nonzeros    : [2, 3, 1]
Indices     : [0,          1          ]
```

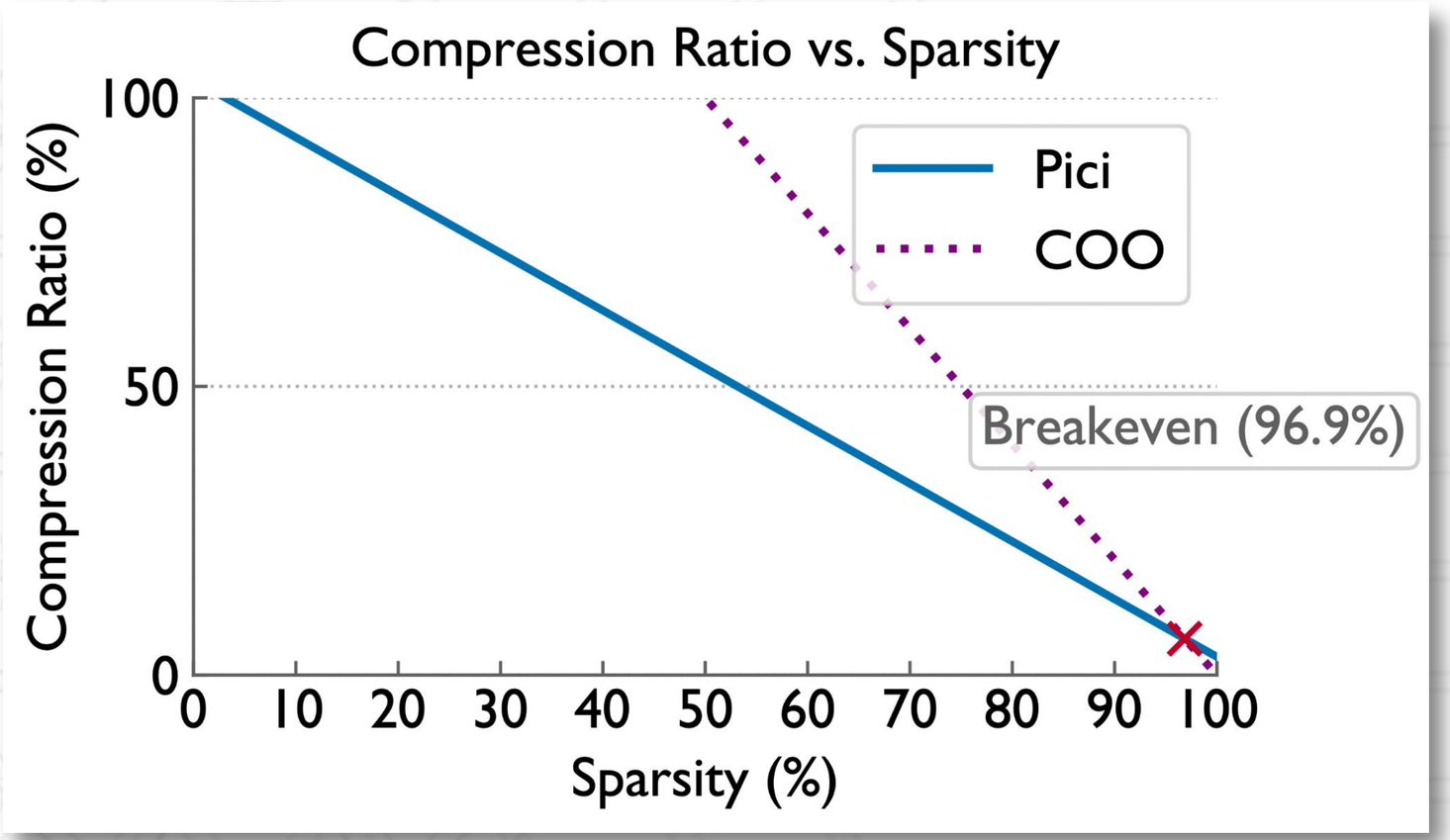
Solution: Pici

- On-the-fly prefix sum of column counts obtained via **popcount** allows threads to start at any column
- Column-wise tile layout gives coalesced reads when scanning for nonzeros



Solution: Pici

- Index overhead ~0.78% relative to bitvector, Pici overhead ~3.15% relative to dense data
- Bitvector alone is 3.125% relative to dense data



Compression

- Each 64x64 tile is handed by a warp, a warp may handle multiple tiles
- Each thread in a warp handles two columns
- Collective compresses small subset of data at once, so no cross-block coordination

- Compression has three phases:
  - **Phase 1:** scan for nonzeros, populate bitvector and partial index array
    - Warp-shuffle reduction to get nonzero count in tile
  - **Phase 2:** sync, prefix sum nonzero counts, sync
  - **Phase 3:** compact nonzeros into values array
    - Warp-shuffle prefix sum on popcounts to get per-column nonzero count
    - `__ffsll` intrinsic to avoid scanning every row in column

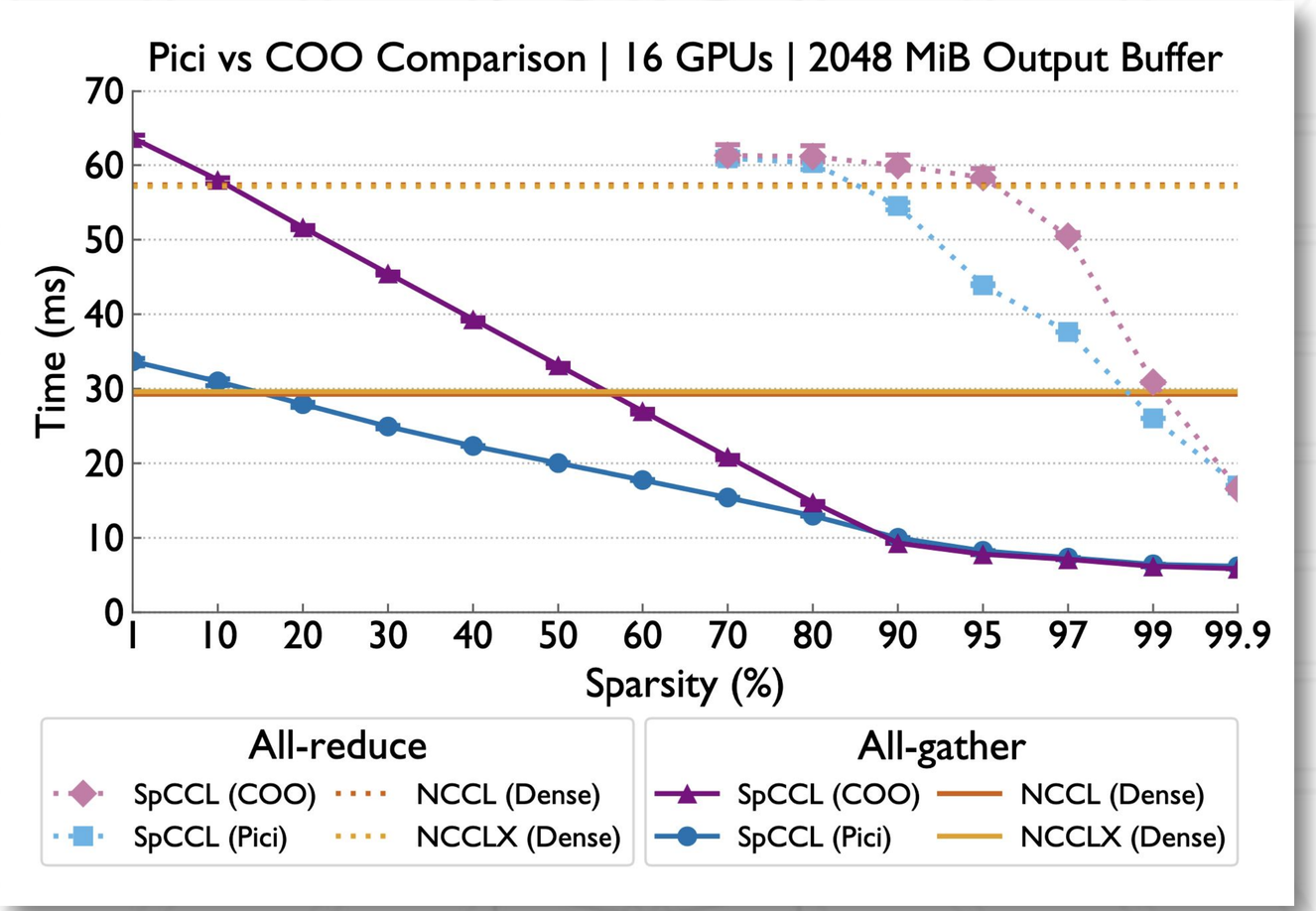
- Decompression is similar to phase 3 of compression and implements scatter-add variant

SpCCL

- Implemented as extension to **NCCLX** **(really just NCCL)**
- Supports all-gather, reduce-scatter, all-reduce
- Extends NCCLX Ring algorithm
  - All-reduce is implemented as reduce-scatter + all-gather

- **All-gather:** Compression can occur once per rank
- **Reduce-scatter, all-reduce:** Must accumulate into dense buffer at each step

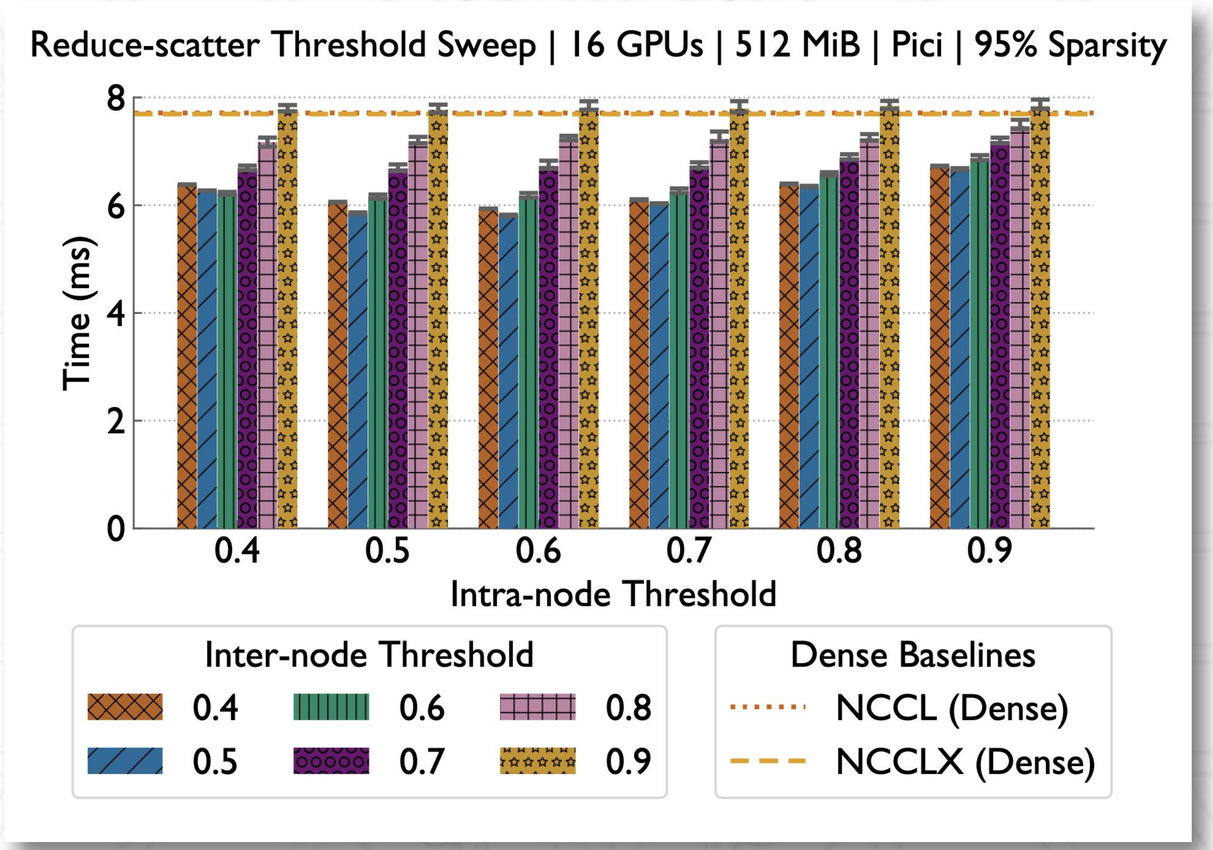
SpCCL – Pici v COO



SpCCL – Adaptive Sparsity

- **Adaptive sparsity:** thresholds control sending dense data when overhead exceeds savings
- **Link-aware adaptivity:** Threshold varies intra- vs inter-node
- **Phase-aware adaptivity:** For all-reduce, threshold varies by RS/AG phase

SpCCL – Adaptive Sparsity

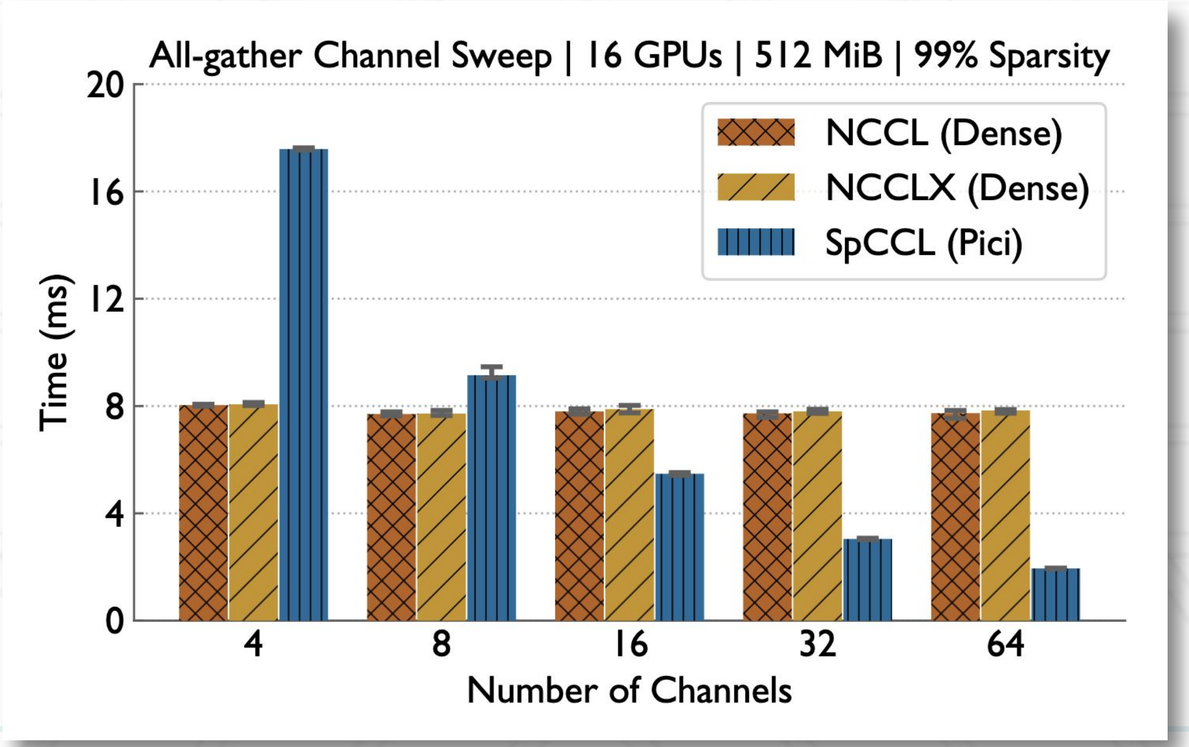


NCCL Background

- **Protocols:** Simple, LL, LL128
  - SpCCL uses **Simple**
- **Channels:** NCCL communication divided across channels
- **Slices:** Granularity of individual network transmissions
  - Pipelined using send/recv FIFO buffers

SpCCL – Implementation Considerations

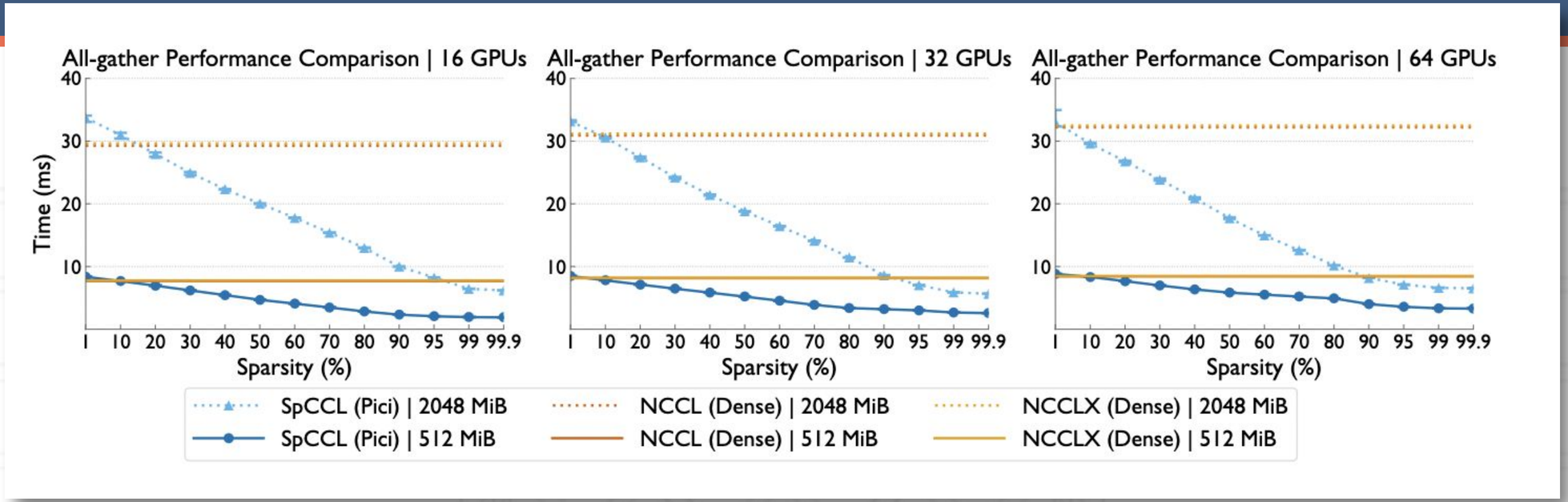
- **Compression:** at NCCL "slice" granularity, within `genericOp`
- **Optimization 1:** channel count is raised to saturate NIC
- **Optimization 2:** intra-node recvs go into staging buffer before scatter-add

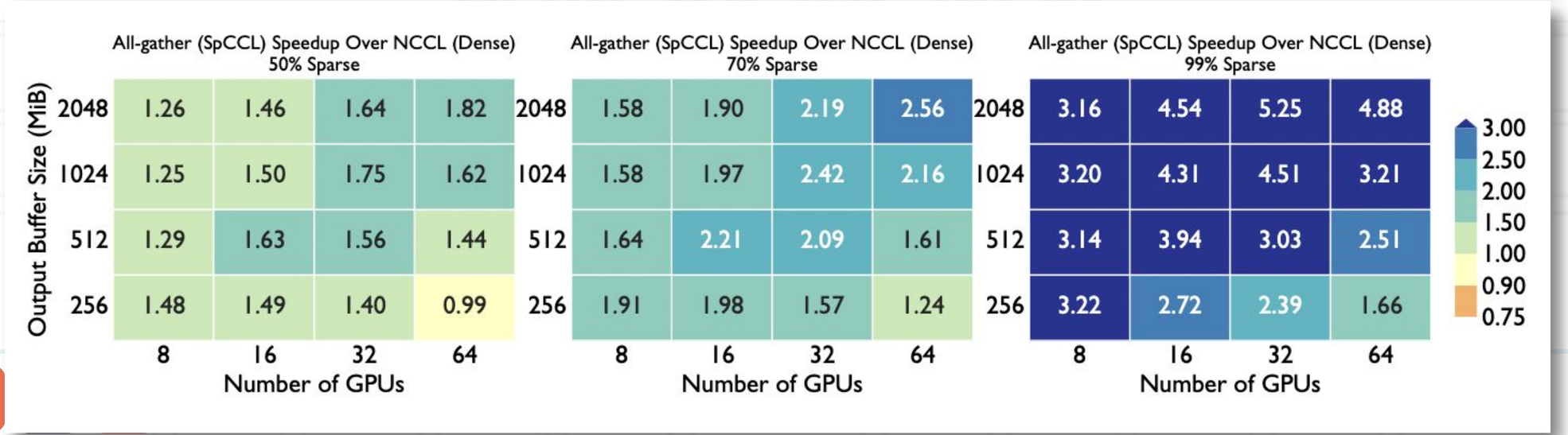


Evaluation – Microbenchmarks

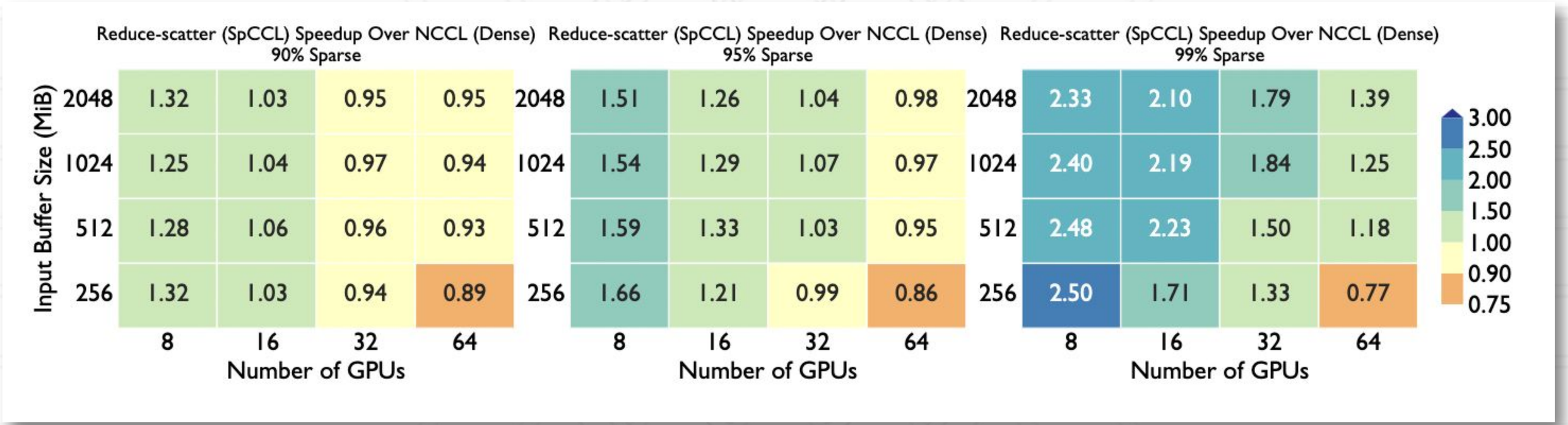
- All evaluations conducted on Perlmutter
  - 4x A100 per node
  - Slingshot inter-node, NVLink intra-node
- **Dense baselines:** NCCL, NCCLX
- **Sparse baselines (all-reduce):** SparCML
  - CPU-based, key-value/1D COO format
- **Scaling:** 8-64 GPUs
- **Problem sizes:** 256, 512, 1024, 2048 MiB

All-gather Microbenchmarks

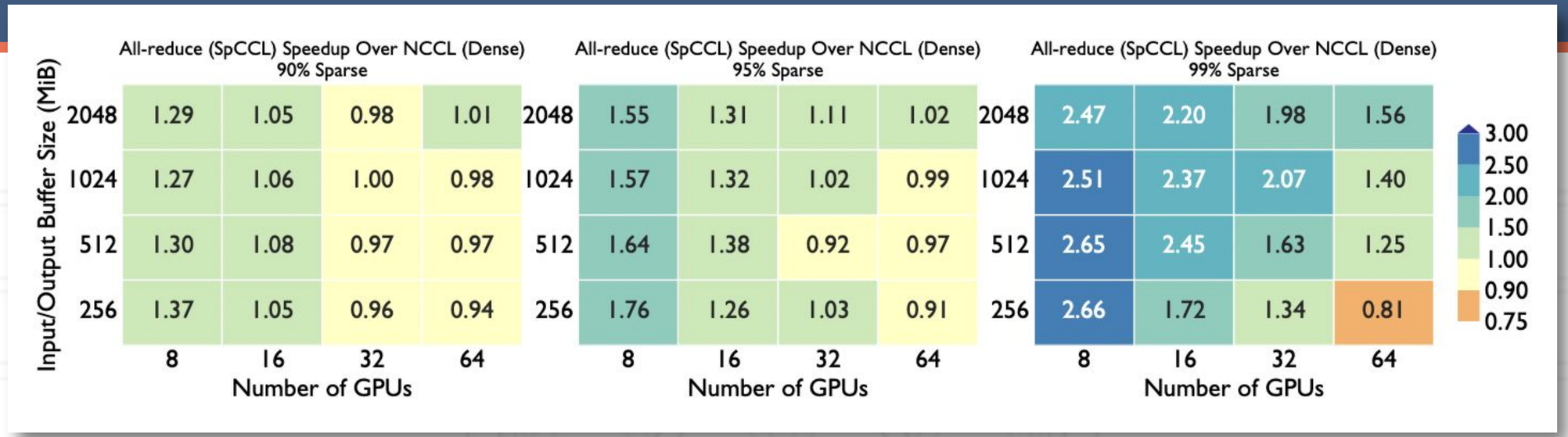


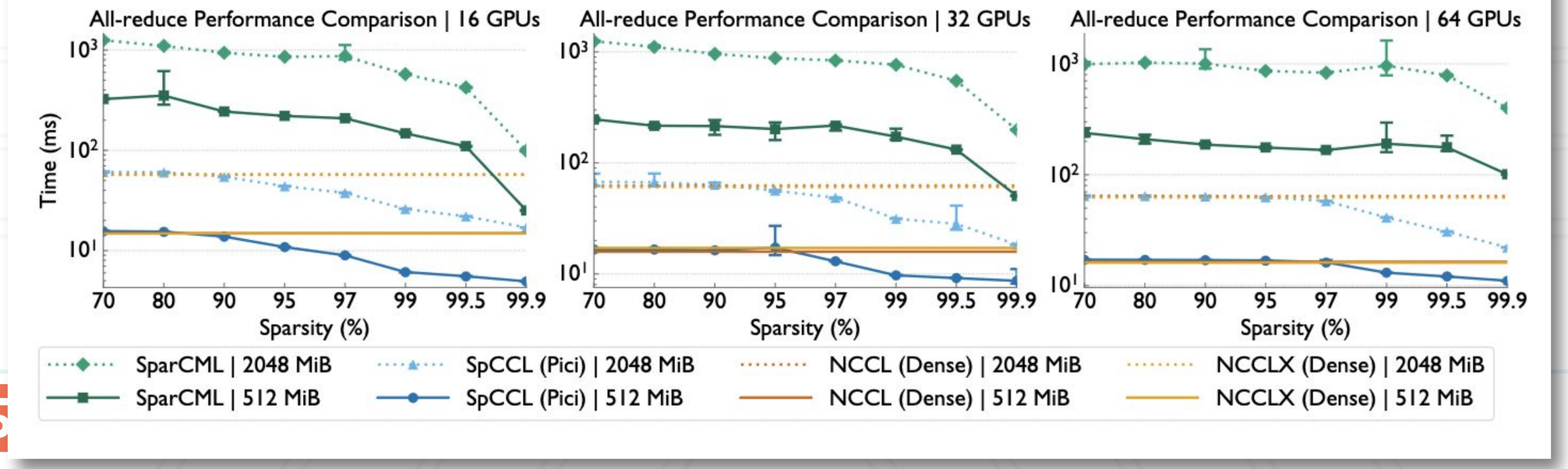


Reduce-scatter Microbenchmarks

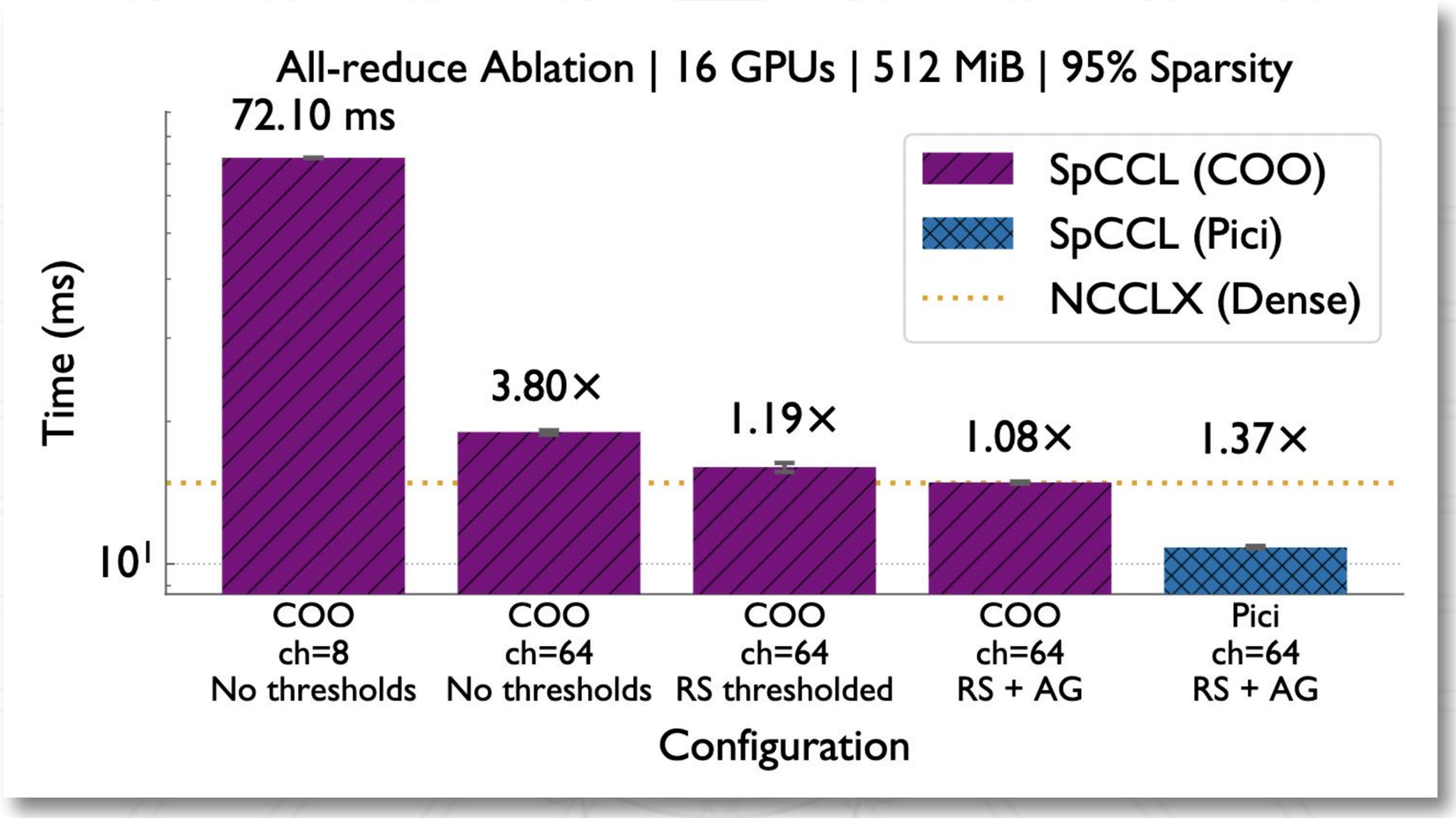


All-reduce Microbenchmarks



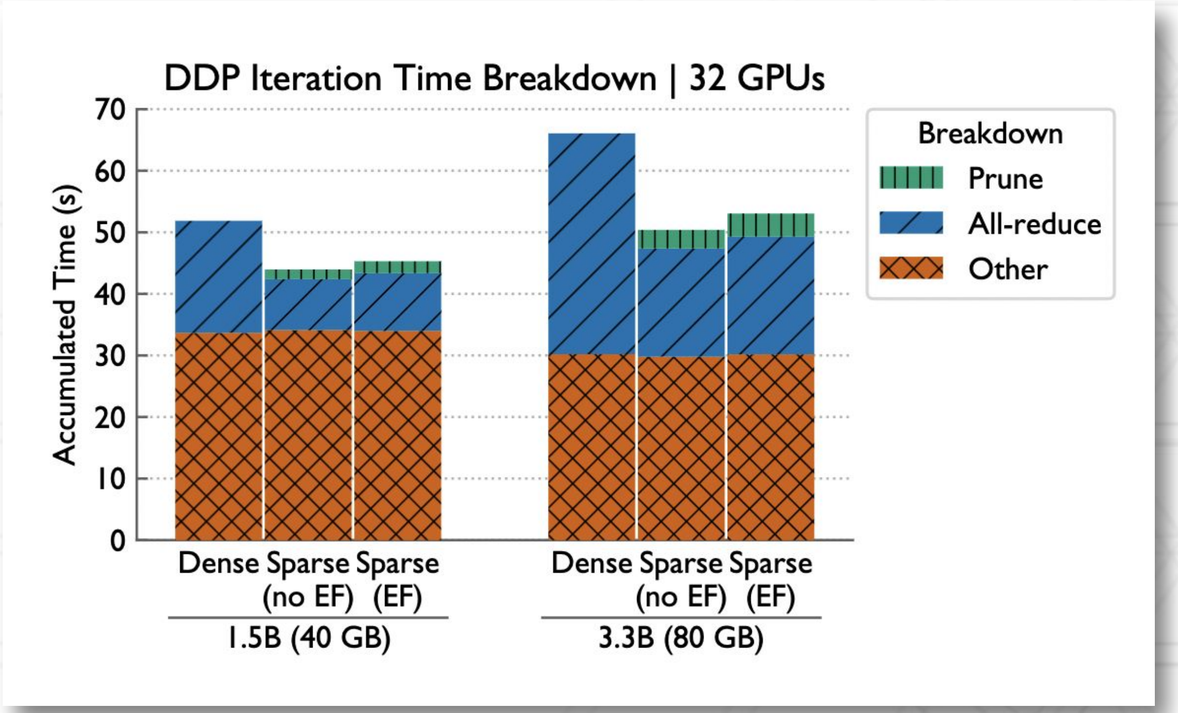


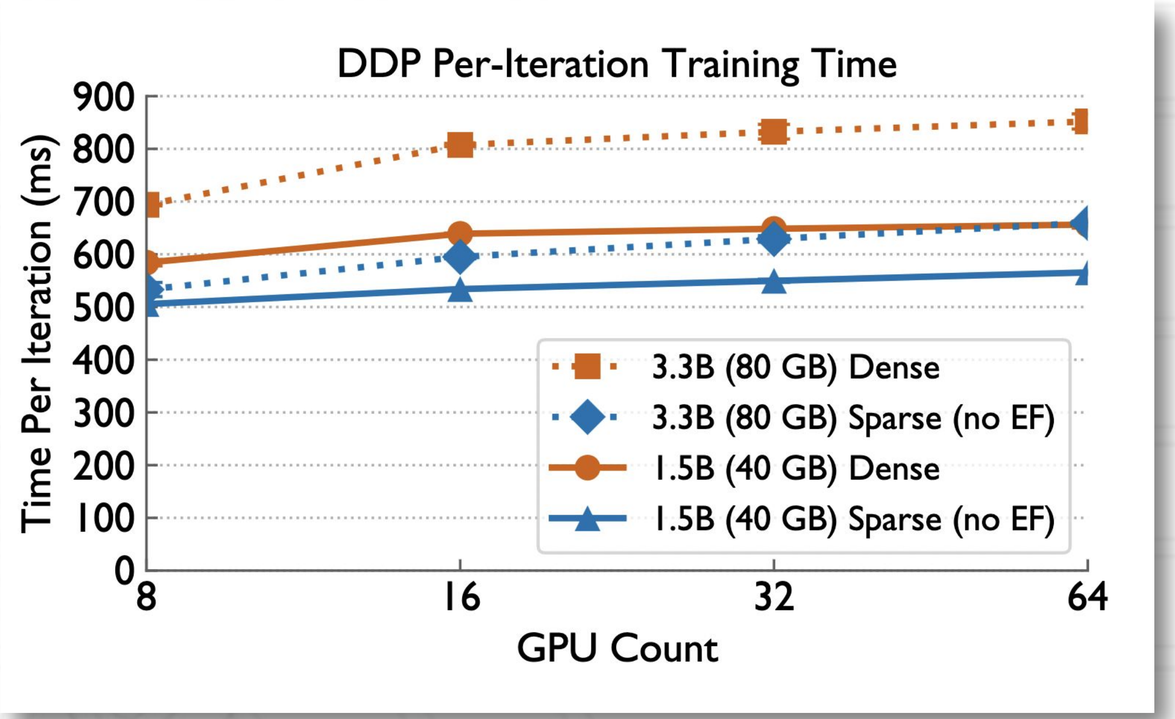
Ablations


All-reduce Ablation | 16 GPUs | 512 MiB | 95% Sparsity
72.10 ms
3.80×
1.19×
1.08×
1.37×
SpCCL (COO)
SpCCL (Pici)
NCCLX (Dense)
Time (ms)
10^1
COO ch=8 No thresholds
COO ch=64 No thresholds
COO ch=64 RS thresholded
COO ch=64 RS + AG
Pici ch=64 RS + AG
Configuration

LLM Training

- **Case study:** gradient-pruned LLM training





Limitations

- **Limitation 1:** Compute/communication overlap
- **Limitation 2:** Imbalanced sparsity across ranks
- **Limitation 3:** Non-Ring algorithms not evaluated
- **Limitation 4:** Some possible baselines unavailable on Perlmutter

Conclusions

- **SpCCL** outperforms dense & sparse baselines in many scenarios
- **Pici** outperforms COO for unstructured data at moderate sparsities, improving scalability

That is it – Thank you for attending!

Questions?

