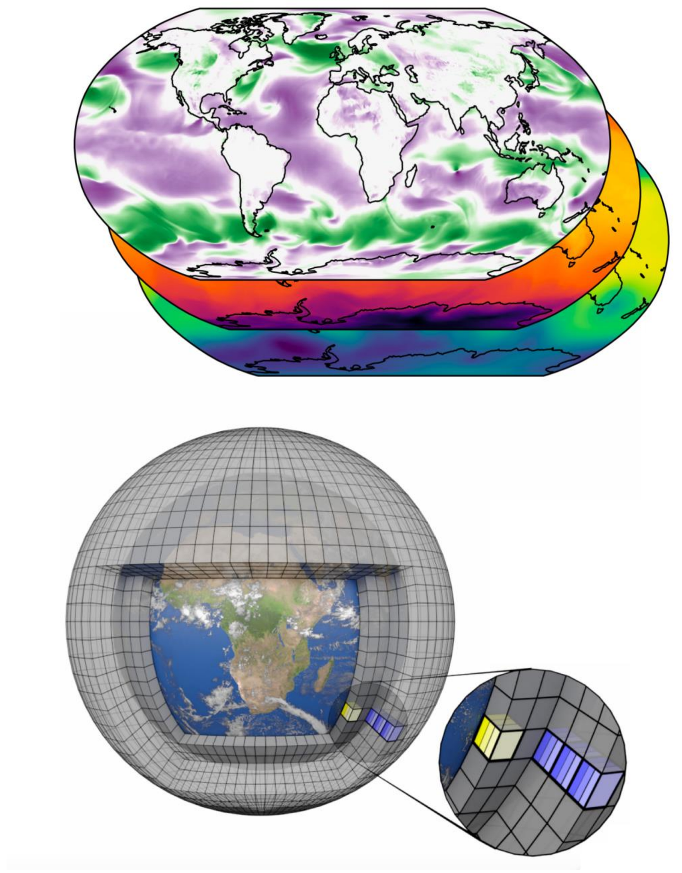
Optimizing Large-Scale Graph Neural Networks for the NVIDIA Grace Hopper Architecture

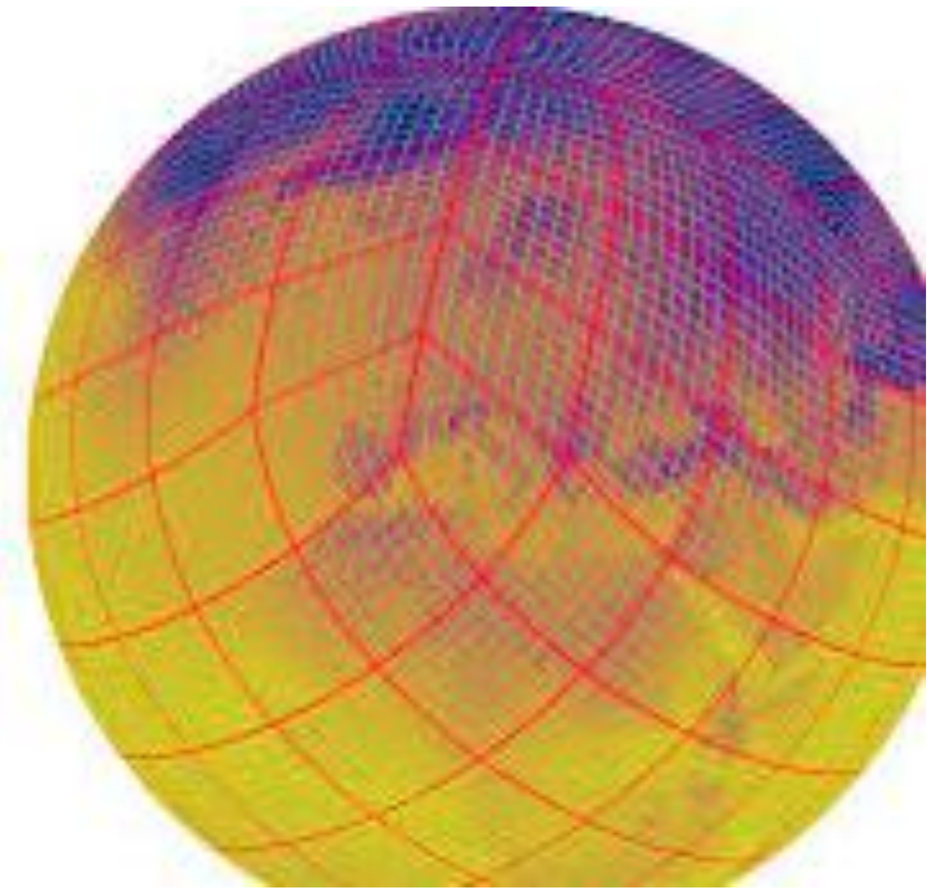
Maximilian Stadler | AI Developer Technology Engineer | 2025-09-17

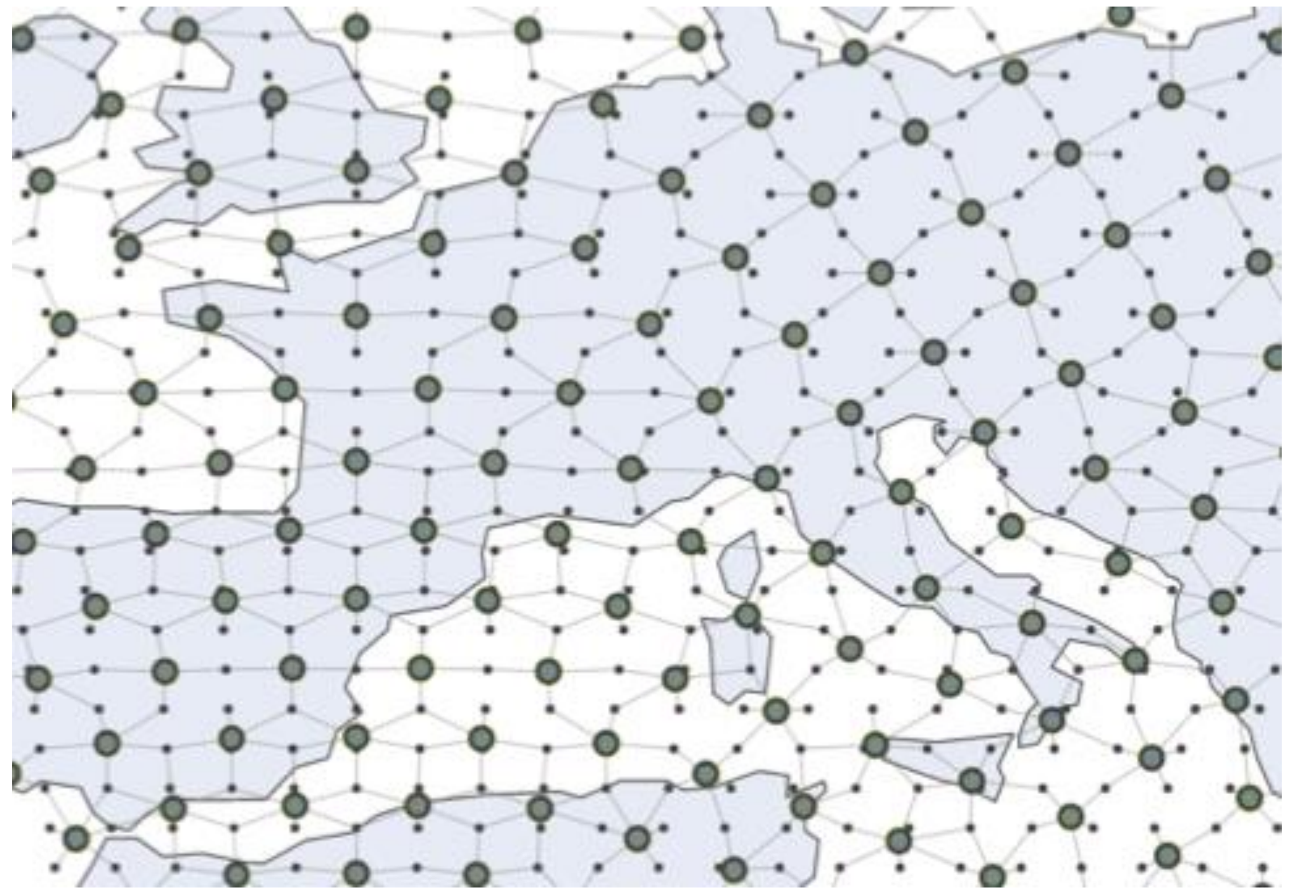
21st ECMWF Workshop on High Performance Computing in Metereology

Diversity of AI4Science

Data Modalities more than just Text or Images







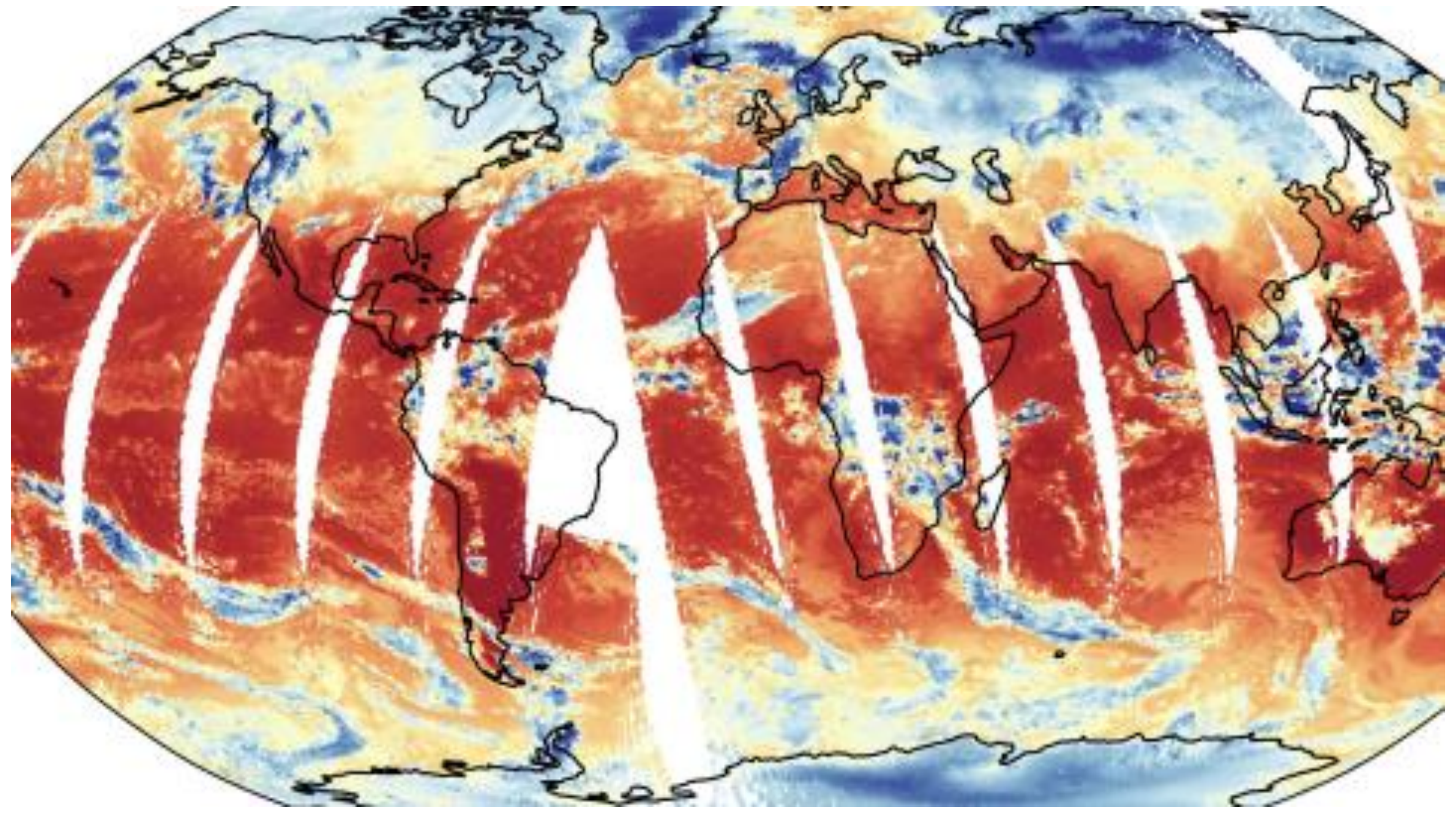


Image Credits: GraphCast | PEAR | AIFS | GraphDOP

Graph Neural Networks

Broad Term – Simple Idea

- any neural network operating on graph data

- Notion of Graphs
  - usually representing directed relations (`is_neighbor_of / is_category / ...`)
  - usually defined as bi-partite graph with source nodes SRC and destination nodes DST being connected by edges representing a relation $SRC \rightarrow DST$

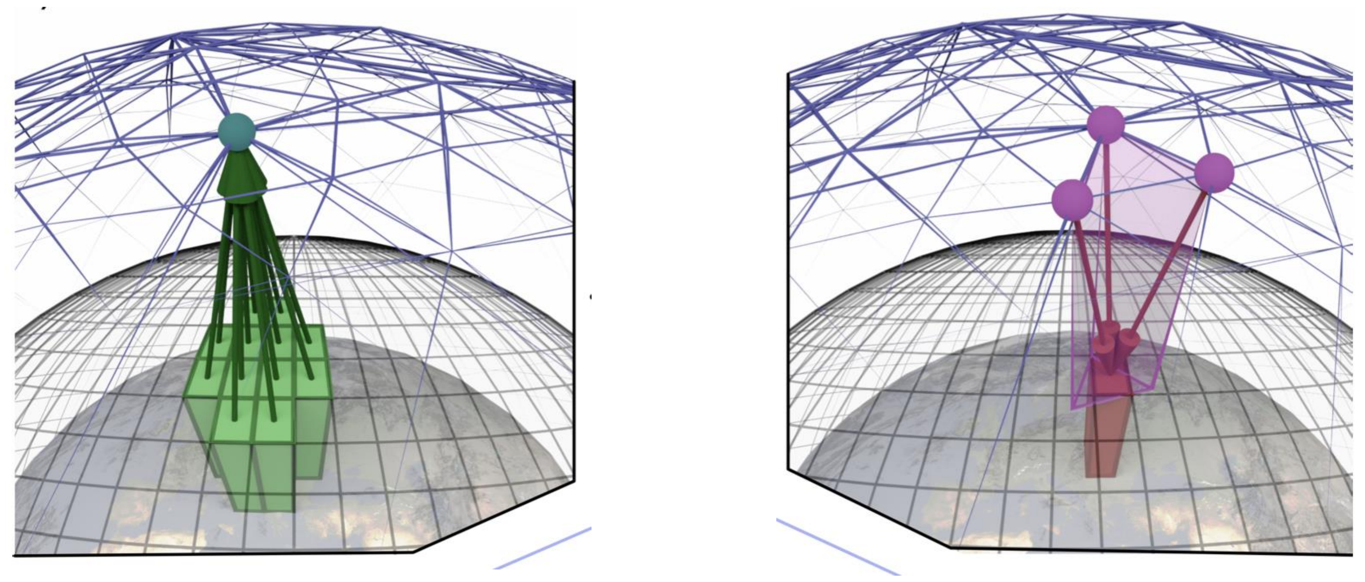


Image Credits: GraphCast

Graph Neural Networks

Broad Term – Simple Idea

- any neural network operating on graph data

- Notion of Graphs
  - usually representing directed relations (`is_neighbor_of / is_category / ...`)
  - usually defined as bi-partite graph with source nodes SRC and destination nodes DST being connected by edges representing a relation $SRC \rightarrow DST$

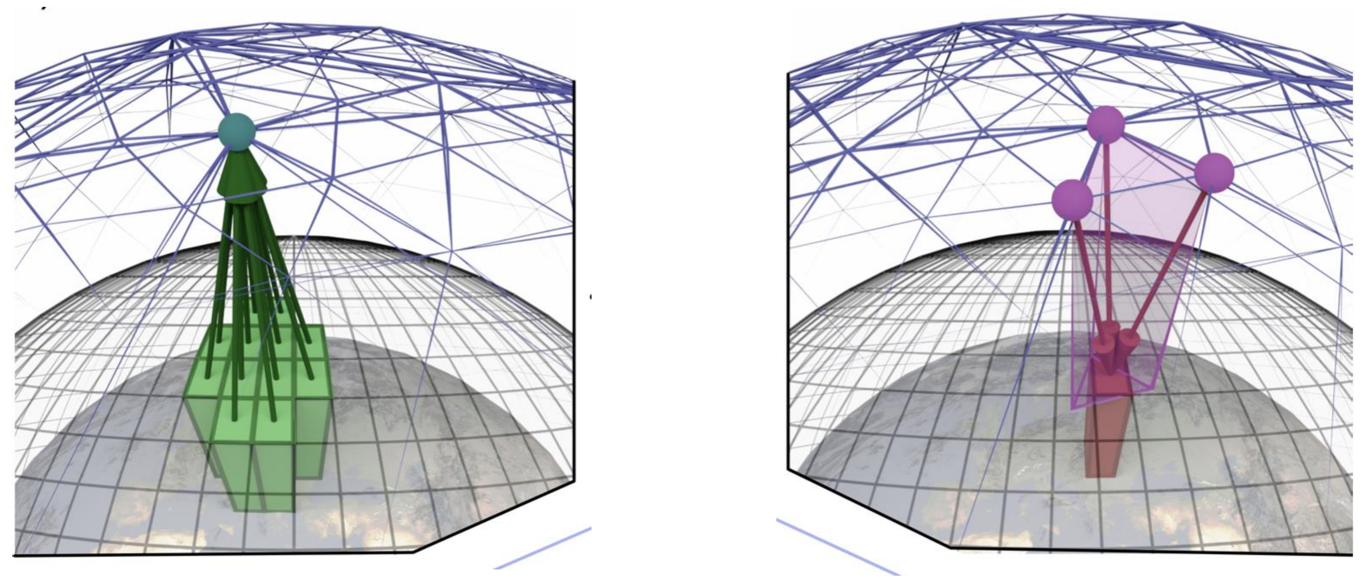


Image Credits: GraphCast

- Message Passing
  - transforming node embeddings leveraging graph structure
  - creation of „messages“ $m(x_{src}, x_{dst}, x_{edge})$
  - aggregation of messages in neighborhood $h_{dst} = AGG_{src \in N(dst)}(m(x_{src}, x_{dst}, x_{edge}))$
  - Examples
    - Simple ”Convolution”: $m(x_{src}, x_{dst}, x_{edge}) = Linear\ (x_{src} | \ x_{dst} \ | \ x_{edge})\ \ with\ \ AGG_{src \in N(dst)} = \sum$
    - GraphTransformer: $x_{src} \rightarrow key\ \&\ value\ , x_{dst} \rightarrow query$, AGG local attention within neighborhood

Graph Neural Networks

Is optimizing then any different?

- General Model Optimizations
  - DataLoader (ZARR, custom pipelines, NVIDIA DALI, NVIDIA PhysicsNeMo, …)
  - Optimizer (PyTorch, NVIDIA APEX, …)
  - Kernel Fusion (torch.compile, NVIDIA TransformerEngine, NVIDIA APEX, …)
  - Custom Kernels (FlashAttention, cuDNN, Triton Kernels for Fused GEMM + ACT kernels, etc..)

Graph Neural Networks

Is optimizing then any different?

- General Model Optimizations
  - DataLoader (ZARR, custom pipelines, NVIDIA DALI, NVIDIA PhysicsNeMo, …)
  - Optimizer (PyTorch, NVIDIA APEX, …)
  - Kernel Fusion (torch.compile, NVIDIA TransformerEngine, NVIDIA APEX, …)
  - Custom Kernels (FlashAttention, cuDNN, Triton Kernels, etc..)

- Specifically for Message-Passing
  - **representation of graph structure can be a major factor**
  - **edge list** (PyTorch Geometric): materializes messages if done naively, requires atomic additions, slow indexing backward

```
src, dst = edge_index
x_src = x[src]  # [E, D]
out = torch.empty((N_DST, D), ...)
for e_idx, d in enumerate(dst):
    out[d] += x_src[e_idx]
```

Graph Neural Networks

Is optimizing then any different?

- General Model Optimizations
  - DataLoader (ZARR, custom pipelines, NVIDIA DALI, NVIDIA PhysicsNeMo, …)
  - Optimizer (PyTorch, NVIDIA APEX, …)
  - Kernel Fusion (torch.compile, NVIDIA TransformerEngine, NVIDIA APEX, …)
  - Custom Kernels (FlashAttention, cuDNN, Triton Kernels, etc..)

- Specifically for Message-Passing
  - **representation of graph structure can be a major factor**
  - **edge list** (PyTorch Geometric): materializes messages if done naively, requires atomic additions, slow indexing backward

```
src, dst = edge_index
x_src = x[src]  # [E, D]
out = torch.empty((N_DST, D), ...)
for e_idx, d in enumerate(dst):
    out[d] += x_src[e_idx]
```

  - **compressed format:** allows for easier explicit kernel fusion, in-register accumulation, and avoids materialization

```
out = torch.empty((N_DST, D), ...)
for d in range(N_DST):
    off_start, off_end = offsets[d], offsets[d+1]
    acc = 0.0
    for e_idx in range(off_start, off_end):
        src = indices[e_idx]
        acc += out[src, :]
    out[d, :] = acc
```

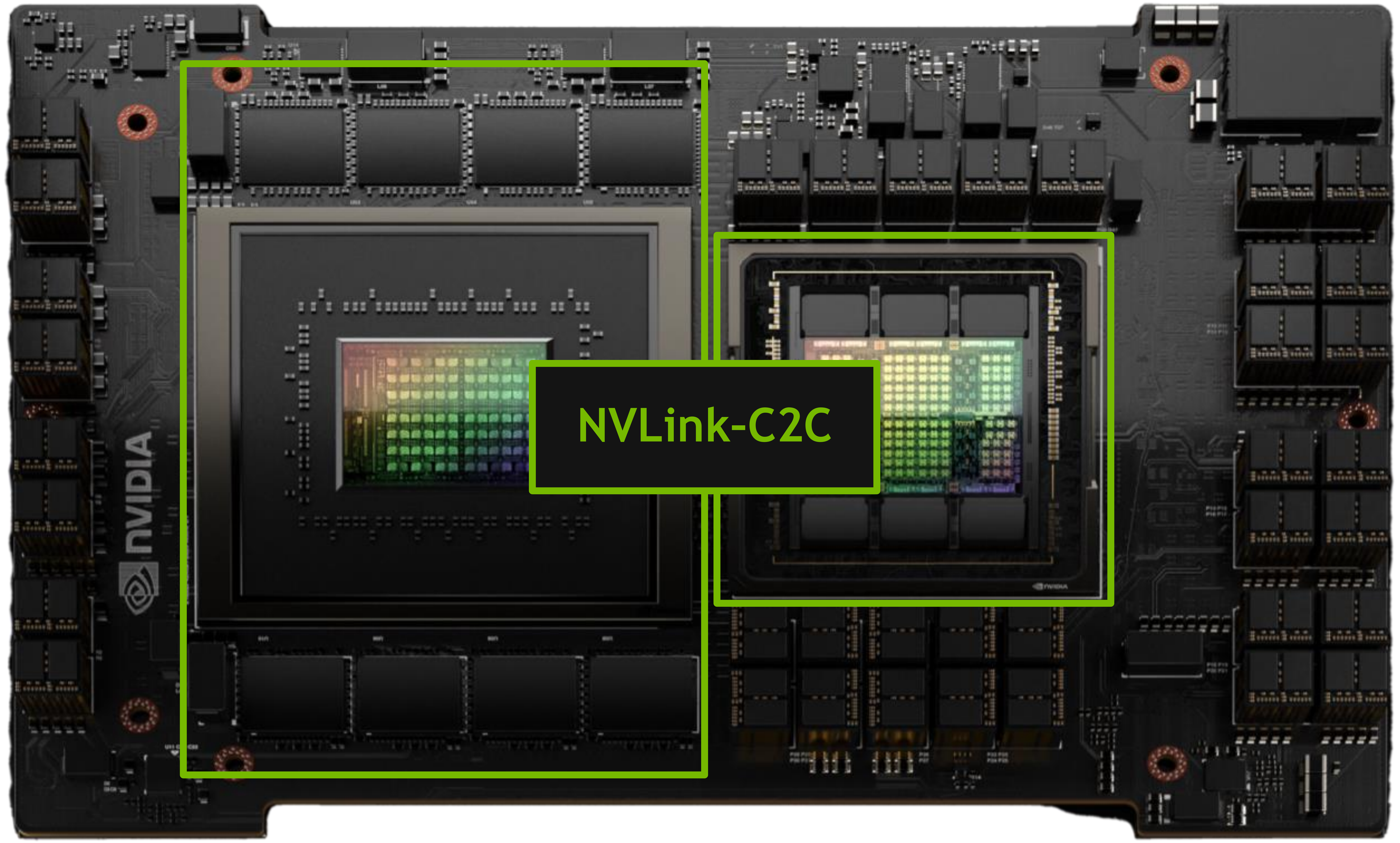
Grace Hopper Superchip

JUPITER | ALPS | ISAMBARD | HELIOS | …
40,000+ GH200 GPUs in European HPC Clusters

Grace CPU

72 Arm Neoverse V2 cores
LPDDR5X Memory

- Capacity: 120 GB
- Bandwidth: up to 500 GB/s



Hopper GPU

HBM3 Memory

- **Capacity: 96 GB**
- **Bandwidth: up to 4 TB/s**

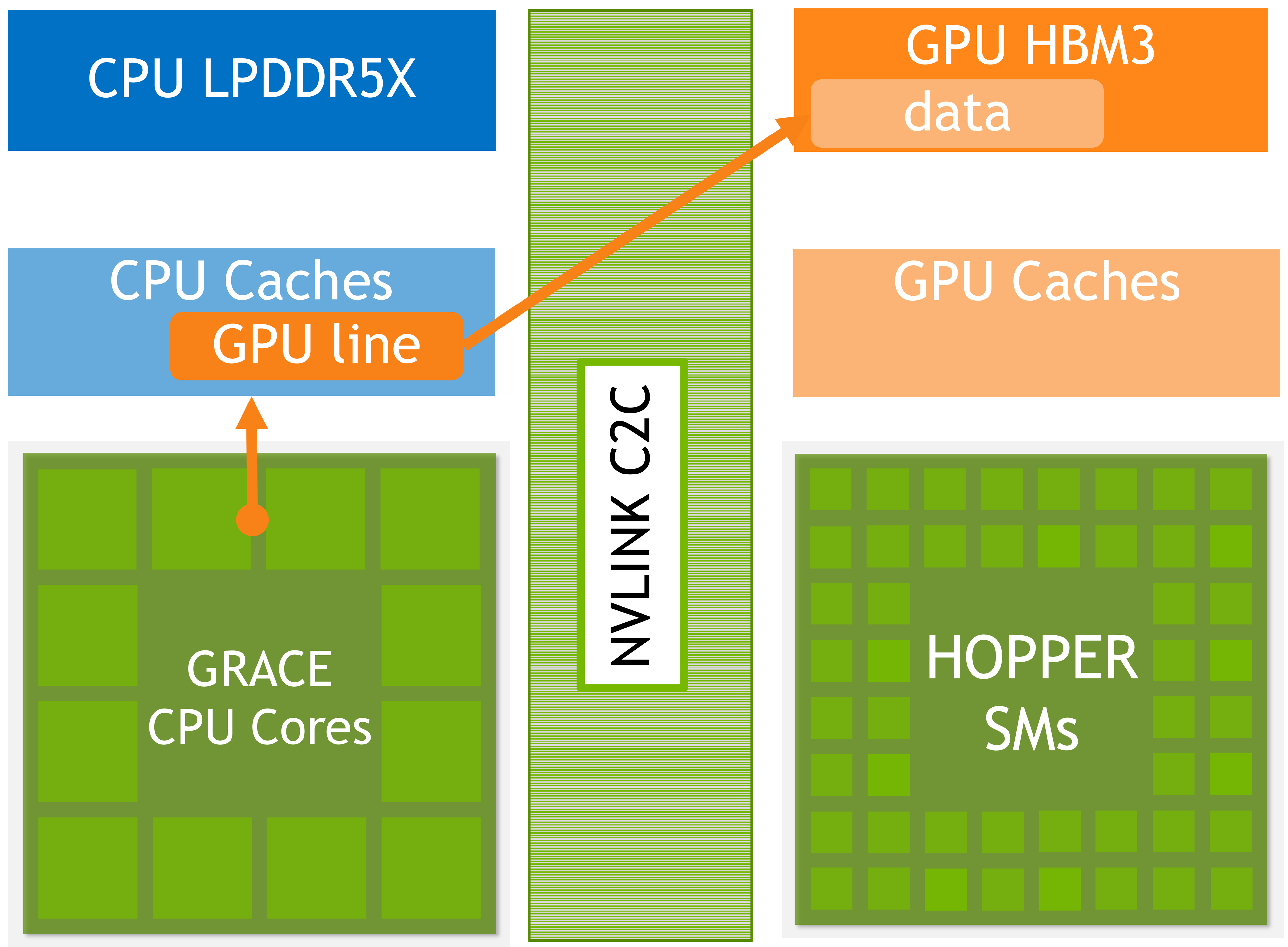
132 SMs

- FP32: 67 TF/s
- FP64: 34 TF/s
- **TF32-TC: 494 TF/s**
- **BF16-TC: 989 TF/s**

900 GB/s bidirectional bandwidth
(450 GB/s per direction)

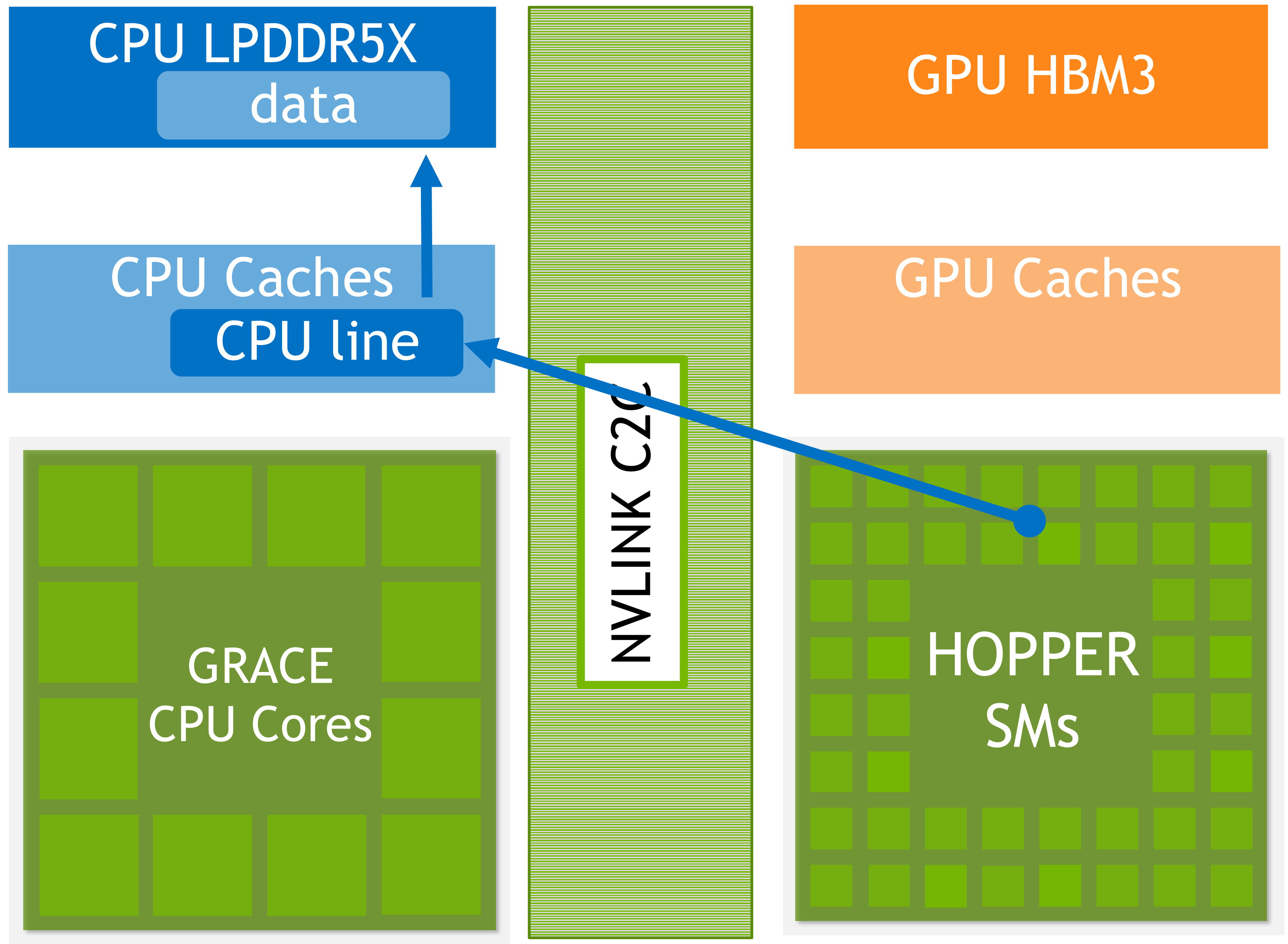
Global Access to All Data

Cache-coherent access via NVLink C2C from either processor to either physical memory



Grace directly reading Hopper's memory

CPU fetches GPU data into CPU L3 cache
Cache remains coherent with GPU memory
Changes to GPU memory evict cache line



Hopper directly reading Grace's memory

GPU loads CPU data via CPU L3 cache
CPU and GPU can both hit on cached data
Changes to CPU memory update cache line

Access Paths

Connecting Grace Hopper Superchips with Memory Consistent NVLink

Parenthesis

Arithmetic Intensity

- **Arithmetic Intensity**
  - basic idea behind roofline models of primitives or models
  - Ratio of FLOPS executed by BYTES read or written
  - Critical Arithmetic Intensity (AI*) is ratio of compute-throughput and memory bandwidth → indicator of limiter

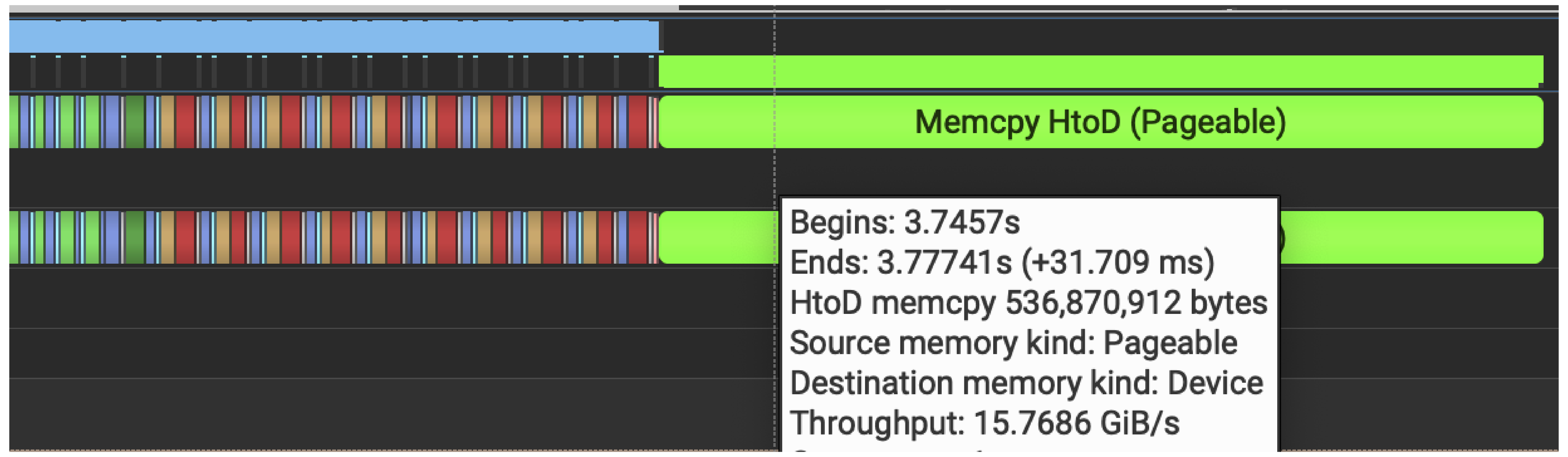
- **Tall GEMM in BF16**: $[N, K] \times [K, K] \rightarrow [N, K]$
  - FLOPS: $2 \times N \times K \times K$
  - BYTES: $\approx 4 \times N \times K$
  - → **AI: $\boldsymbol{K/2}$**

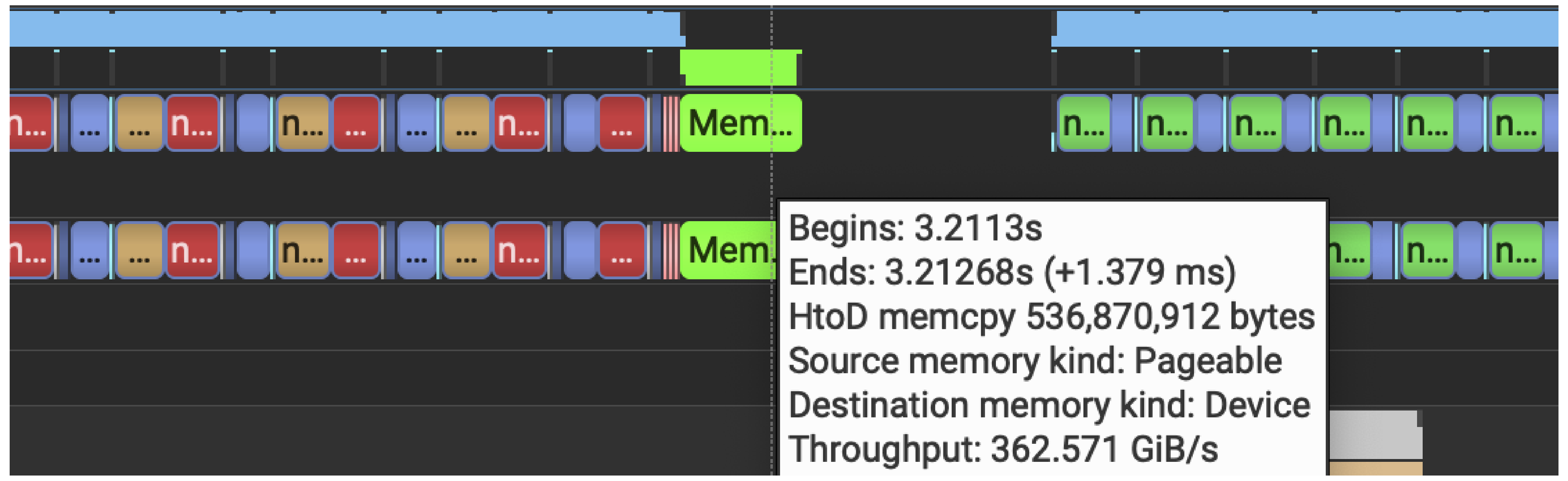
- **Hardware Characteristics**
  - AI*-HBM: $\frac{989\,TF/s}{4\,TB/s} \approx 250\,\frac{FLOPS}{B} \Rightarrow K^* > 500$
  - AI*-PCIe: $\frac{989\,TF/s}{64\,GB/s} \approx 15{,}000\,\frac{FLOPS}{B} \Rightarrow K^* > 30{,}000$
  - AI*-C2C: $\frac{989\,TF/s}{450\,GB/s} \approx 2{,}200\,\frac{\mathrm{FLOPS}}{\mathrm{B}} \Rightarrow K^* > 4{,}400$

Optimizing Graph Neural Networks

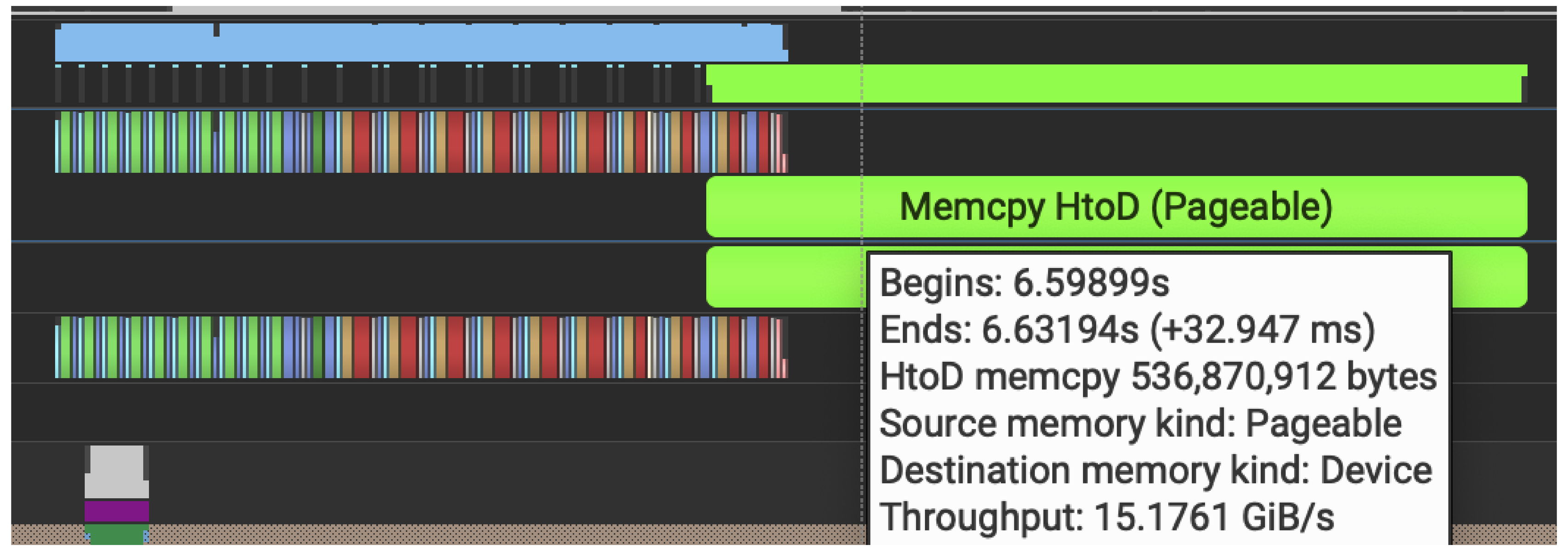
Problem 1: DataLoader and H2D Transfers

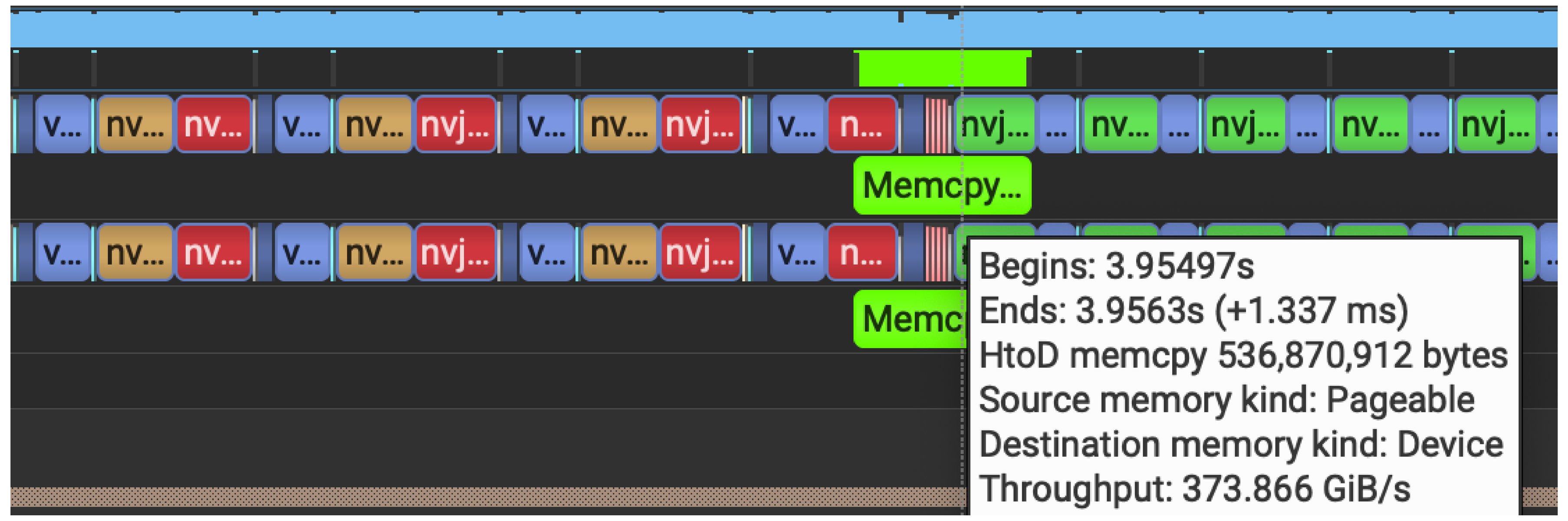
- AI*-PCIe: $\frac{989\,TF/s}{64\,GB/s} \approx 15{,}000\,\frac{FLOPS}{B} \Rightarrow K^* > 30{,}000\ (900M\ param)$ vs AI*-C2C: $\frac{989\,TF/s}{450\,GB/s} \approx 2{,}200\,\frac{\text{FLOPS}}{\text{B}} \Rightarrow K^* > 4{,}400\ (20M\ param)$
- Profiles of Dummy MLP Workload: H200 (left) vs. GH200 (right)





- With Asynchronous Prefetching





- Grace Hopper with C2C
  - makes copying data to GPU easier, does not require pinning memory
  - asynchronous prefetching still beneficial and possible to overlap at smaller model sizes

Optimizing Graph Neural Networks

Problem 2: Memory Footprint

- unlike Large Language Models
  - parameter count usually “small”: O(1M) – O(1B)
  - activation footprint, however, large -> O(1M) – O(1B) rows of node or edge embeddings
  - parenthesis, example GEMM + ACT

GEMM	ACT	…	ACT	GEMM	GEMM

    - $O = \sigma(X @ W) = \sigma(Y)$ hence $grad_Y = grad_O \cdot \sigma'(Y)$ and $grad_X = grad_Y @ W.T$ and $grad_W = X.T @ grad_Y$
    - computing gradients for X and W requires X, W, and output of GEMM
  - → most tensors are just stale until backward pass
  - → Can we optimize that?

Optimizing Graph Neural Networks

Problem 2: Memory Footprint

- unlike Large Language Models
  - parameter count usually “small”: O(1M) – O(1B)
  - activation footprint, however, large -> O(1M) – O(1B) rows of node or edge embeddings
  - parenthesis, example GEMM + ACT

    GEMM | ACT | … | ACT | GEMM | GEMM

    - $O = \sigma(X @ W) = \sigma(Y)$ hence $grad_Y = grad_O \cdot \sigma'(Y)$ and $grad_X = grad_Y @ W.T$ and $grad_W = X.T @ grad_Y$
    - computing gradients for X and W requires X, W, and output of GEMM
  - → most tensors are just stale until backward pass
  - → Can we optimize that?

- Recomputing Activation
  - manually define autograd functions for combinations of primitives with manual control over what is stored for backward pass
  - example here: recompute Y in backward pass, only store X and W

- Checkpointing: `torch.utils.checkpoint.checkpoint(func, *args)`
  - automatic “creation” of checkpoints: divide model into chunks and only store tensors at each end
  - recompute forward passes in between when encountered in backward pass

    GEMM | ACT | … | GEMM | ACT | ACT | GEMM | GEMM

  - **Trade-Off:** number of checkpoints reduces size of required intermediate allocations but also increases number of checkpointed tensors

Offloading

Simple Tensor Hooks

- another option: simply move stored tensors to CPU until needed in the backward pass
- ideally: prefetch them in advance

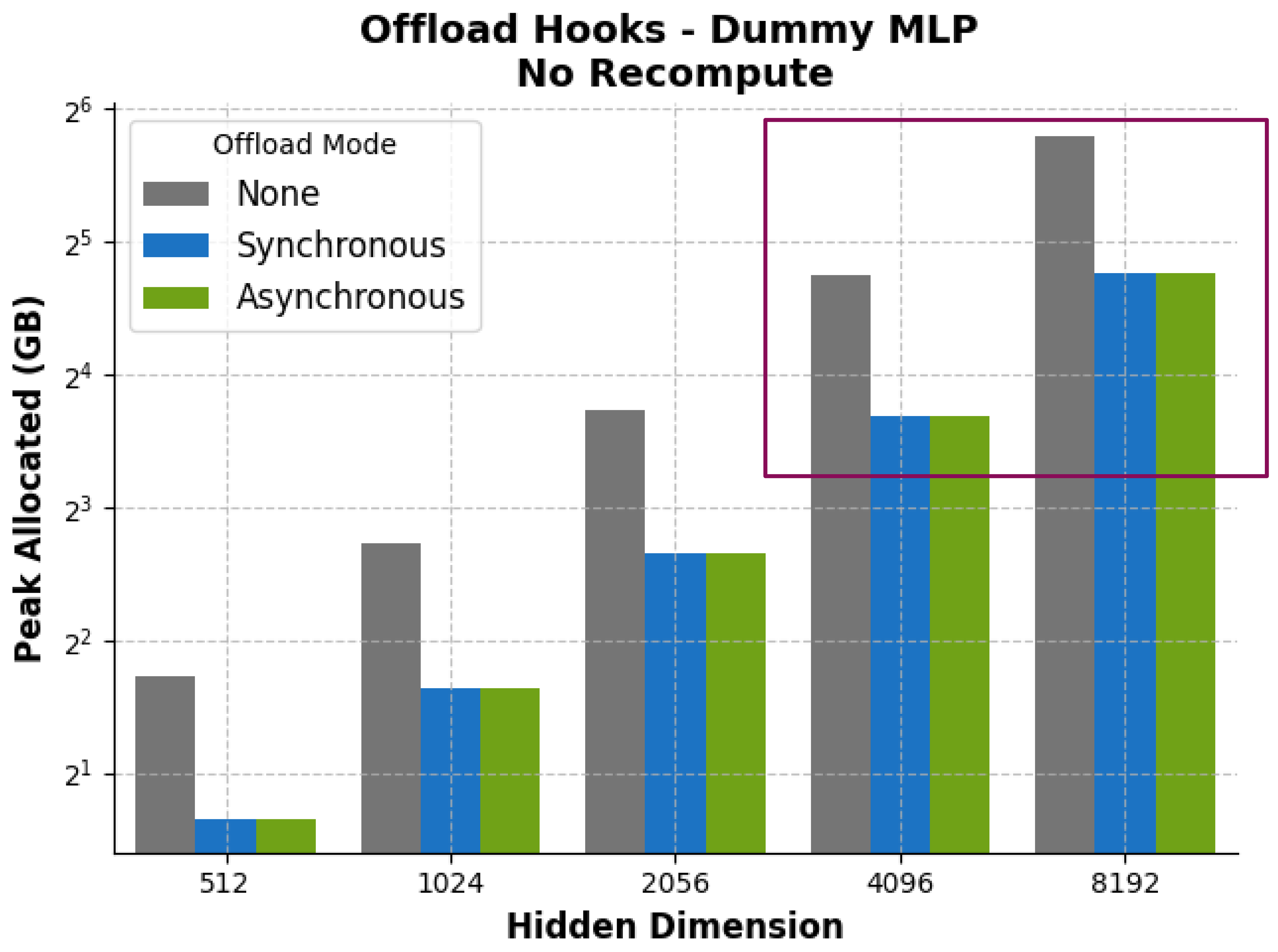
- PyTorch: exposes saved_tensors_hooks
  - either apply modification when “packing” tensors
  - or when “unpacking” tensors

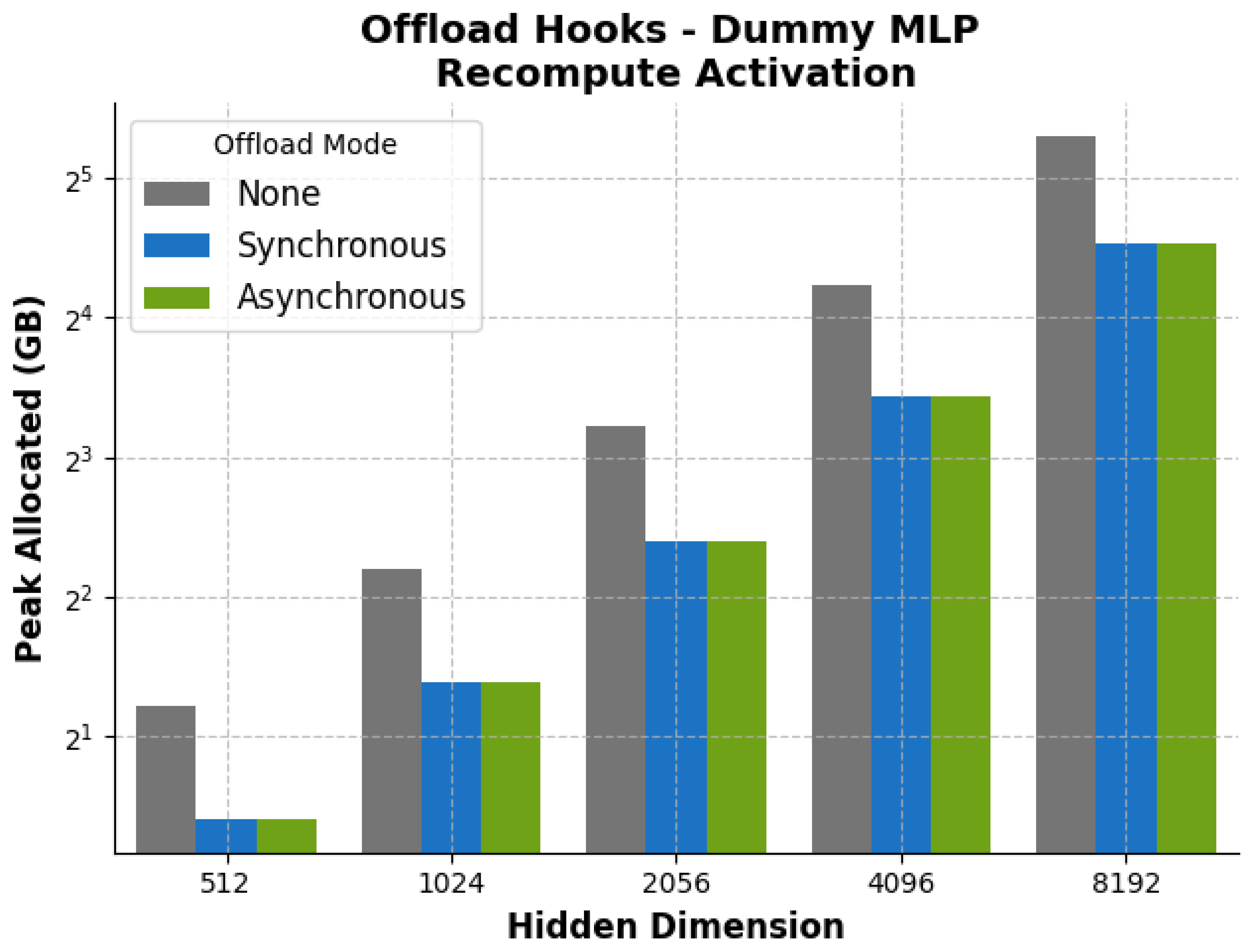
- Asynchronous by
  - leveraging asynchronous copy on dedicated stream
  - non-blocking transfer on host which allows it to run ahead

```python
torch.autograd.graph.saved_tensors_hooks(
    pack_hook, unpack_hook)


def synchronous_pack_hook(tensor):
    return (tensor.cpu(), tensor.device)

def synchronous_unpack_hook(packed):
    return packed[0].to(device=packed[1])


def asynchronous_pack_hook(tensor):
    with torch.cuda.stream(copy_stream):
        packed = torch.empty(
                    tensor.size(),
                    dtype=tensor.dtype,
                    layout=tensor.layout,
                    pin_memory=True,
                    device="cpu",
                )
         packed.copy_(tensor, non_blocking=True)
         tensor.record_stream(copy_stream)

    return (packed, tensor.device)
```

Offloading with Tensor Hooks

Is it that simple?



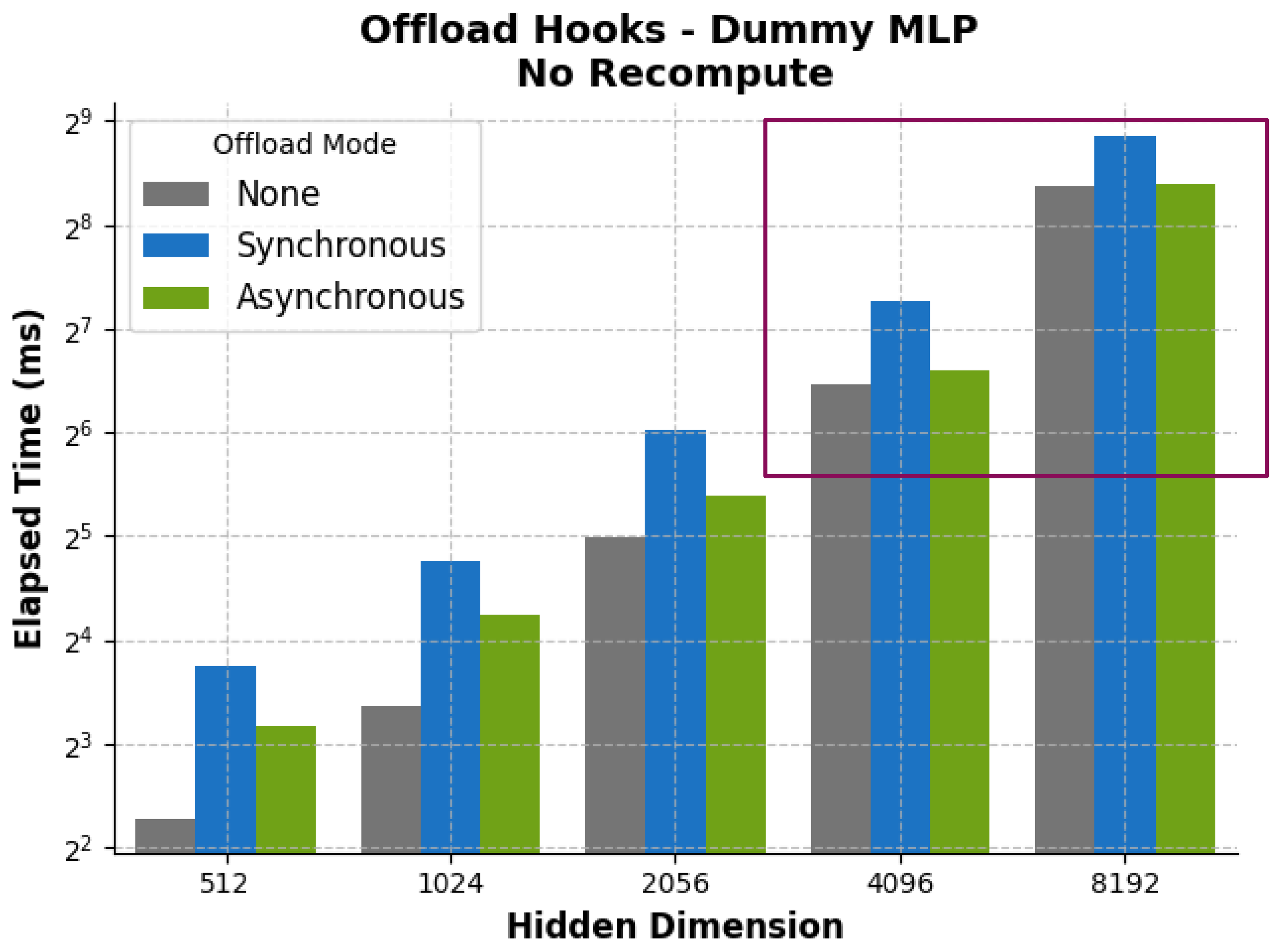


offloading can reduce memory footprint

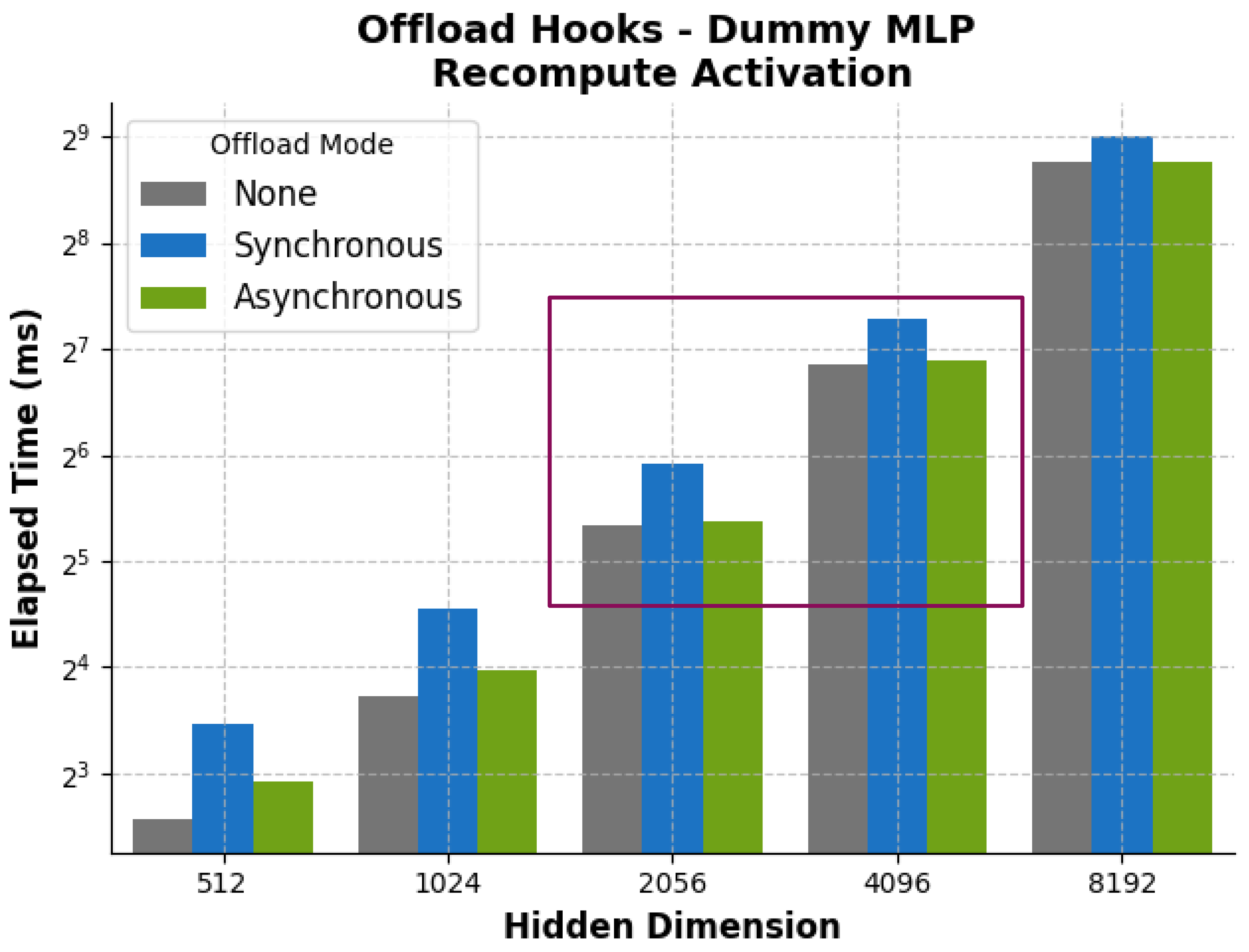
Dummy MLP Workload (Linear, SiLU, LayerNorm) with N=131072 and dtype=bfloat16

Offloading with Tensor Hooks

Is it that simple?



with all tensors offloaded,
offload hidden around D>=4096

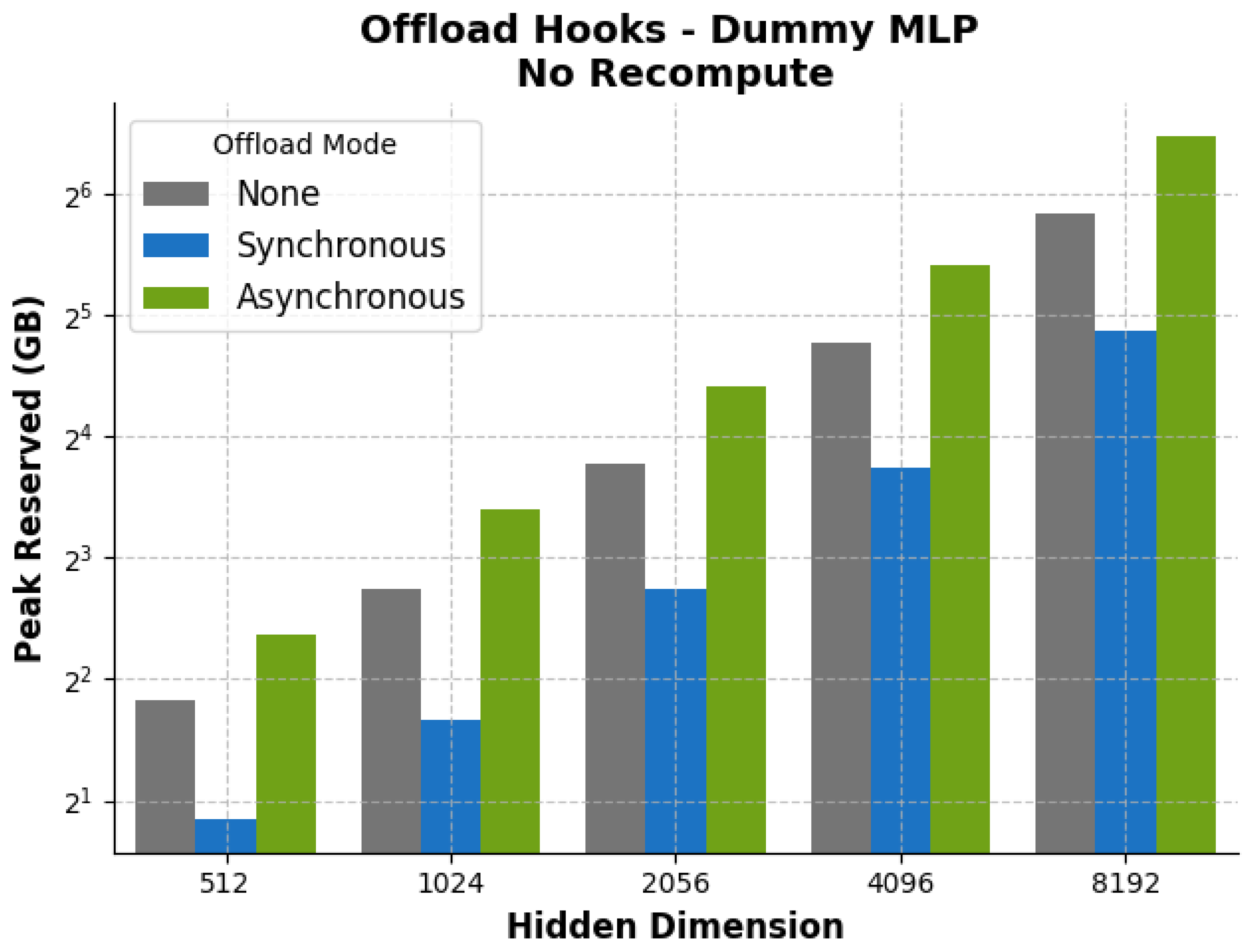


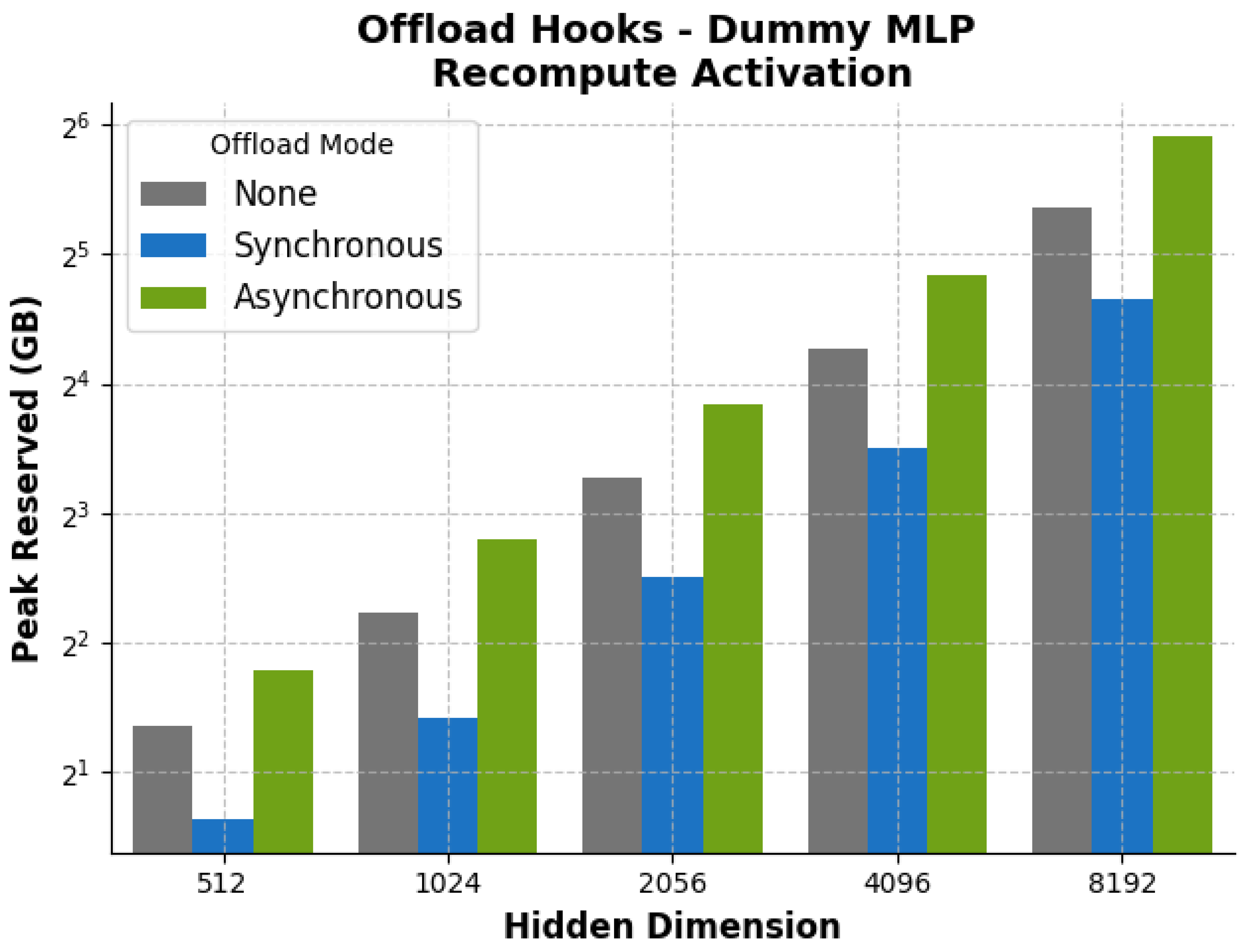
with recomputing activations (less to offload),
offload hidden around D>=2048

Dummy MLP Workload (Linear, SiLU, LayerNorm) with N=131072 and dtype=bfloat16

Offloading with Tensor Hooks

Is it that simple?



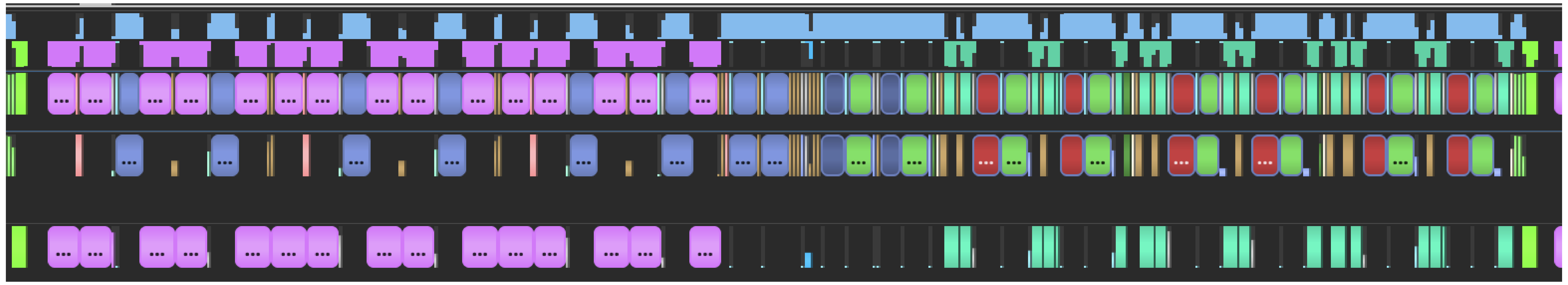


ISSUE: vanilla hooks don't play nicely with PyTorch allocator
→ overall, uses more memory!

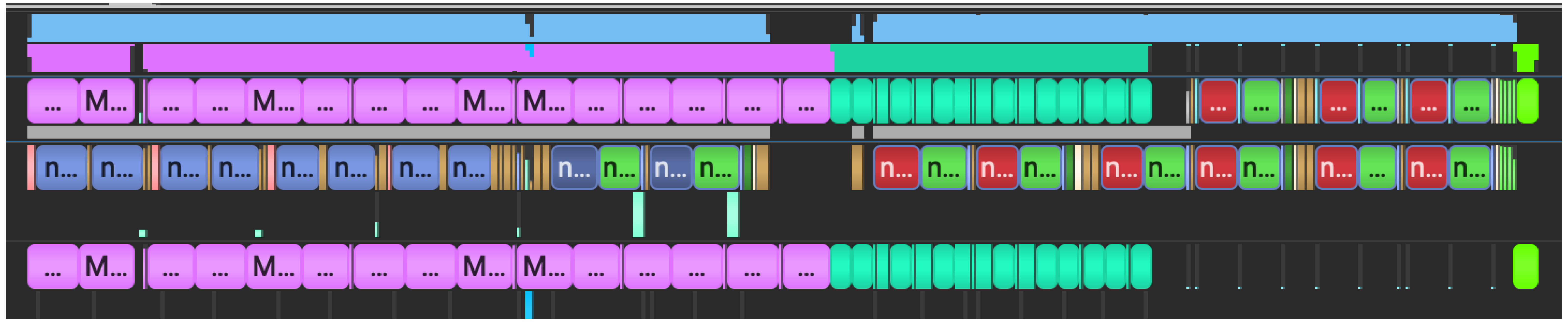
Dummy MLP Workload (Linear, SiLU, LayerNorm) with N=131072 and dtype=bfloat16

Offloading with Tensor Hooks

Is it that simple?



ISSUE: synchronous offloading gives good memory savings, but a lot of time then is spent in D2H (purple) / H2D (teal) for typical problem sizes



ISSUE: asymmetric D2H/H2D means D2H is limiting overlap opportunities
ISSUE: vanilla unpacking “too asynchronous” → to much tensors in memory too early

Dummy MLP Workload (Linear, SiLU, LayerNorm) with N=131072 and dtype=bfloat16

NVIDIA.

Offloading with Tensor Hooks

Practical Recommendations

- unfortunately: no silver bullet – unless hidden sizes are large enough

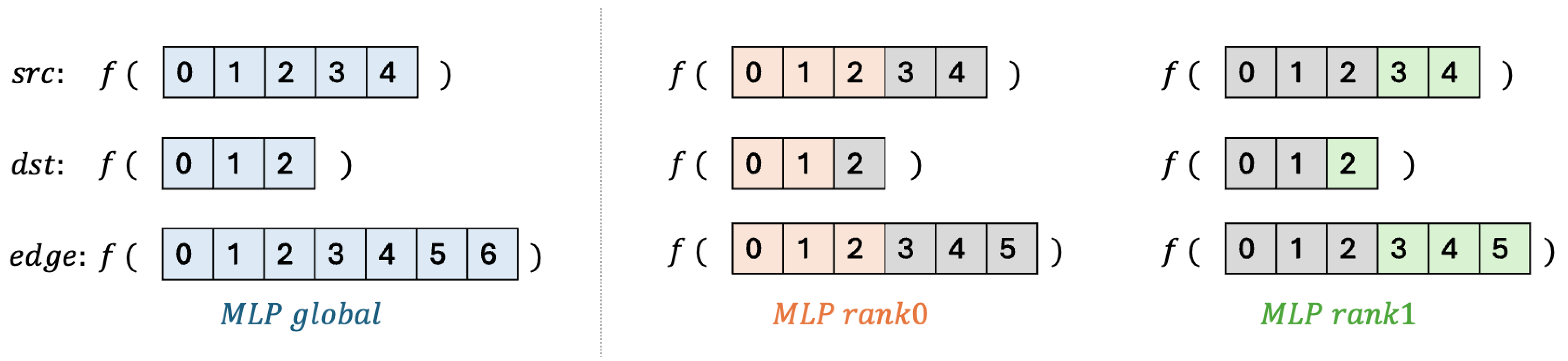
- in practice, requires “massaging” allocators of PyTorch
  - Tensor Hooks not aware of layer definitions
  - **model-specific assumptions**
    - family of model (Transformer, DiT, GNN)
    - layer structure
    - example: offloading for LLMs in NVIDIA TransformerEngine
  - **custom schedules**
    - offload queue
    - limit prefetching (not weights, not tensors smaller than threshold, not last layer, …)

- practical recommendation
  - **targeted offloading** (example only larger tensors in input mappers, only activations - not weights)
  - **combination** with other techniques
    - chunking
    - checkpointing
    - offloading for checkpoints

Distributing Graph Neural Networks

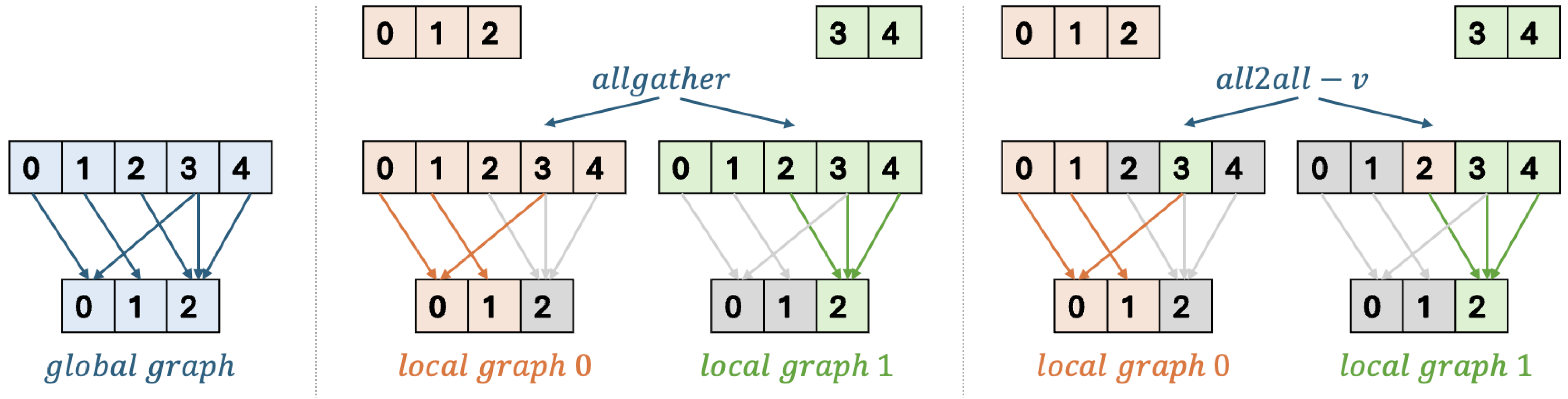
What about simply using more GPUs per model instance?

- orthogonal to single-GPU optimizations
- shard embeddings



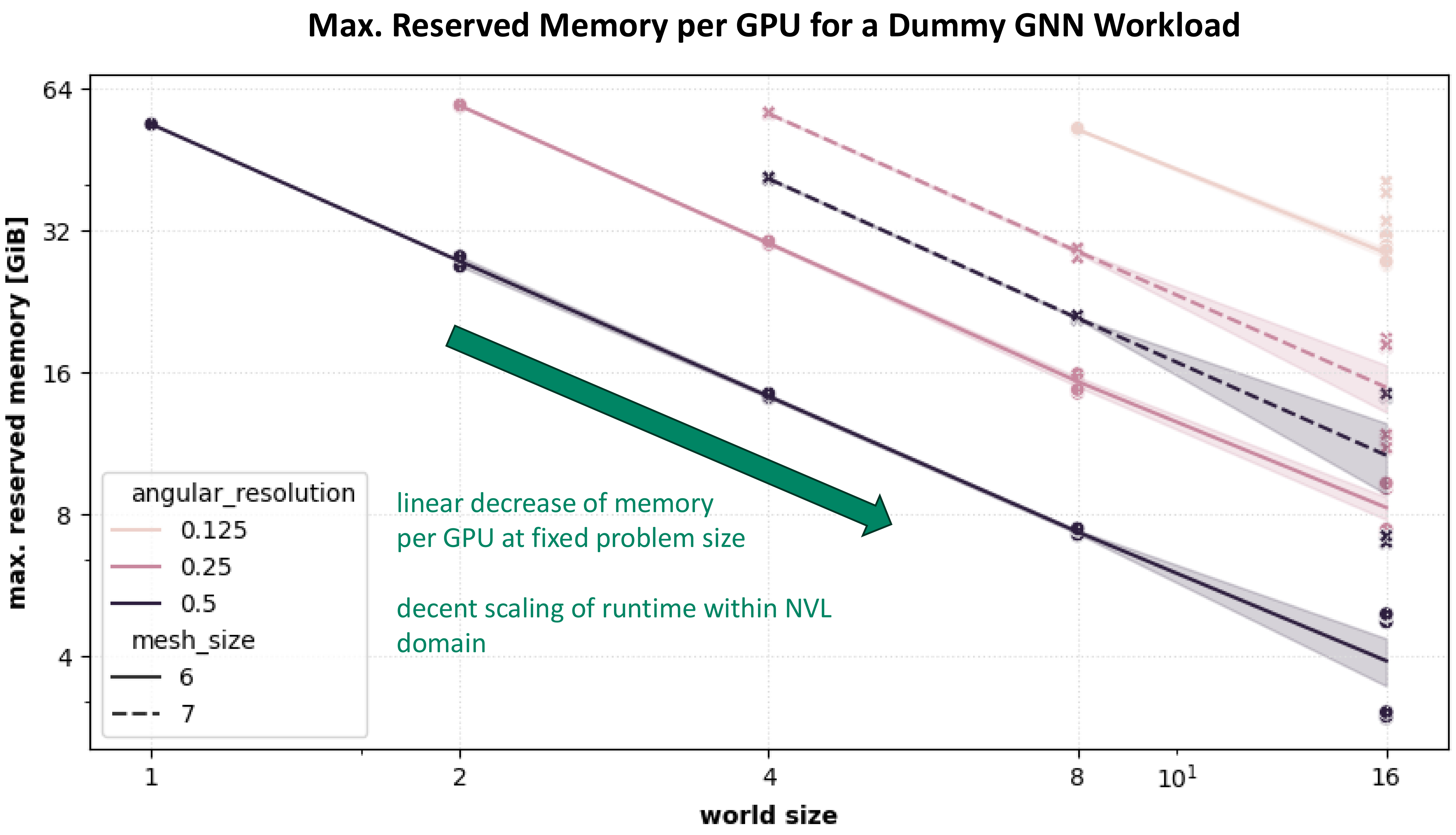
- distribute graph structure

edge-centric approach **node-centric approach**



Distributing Graph Neural Networks

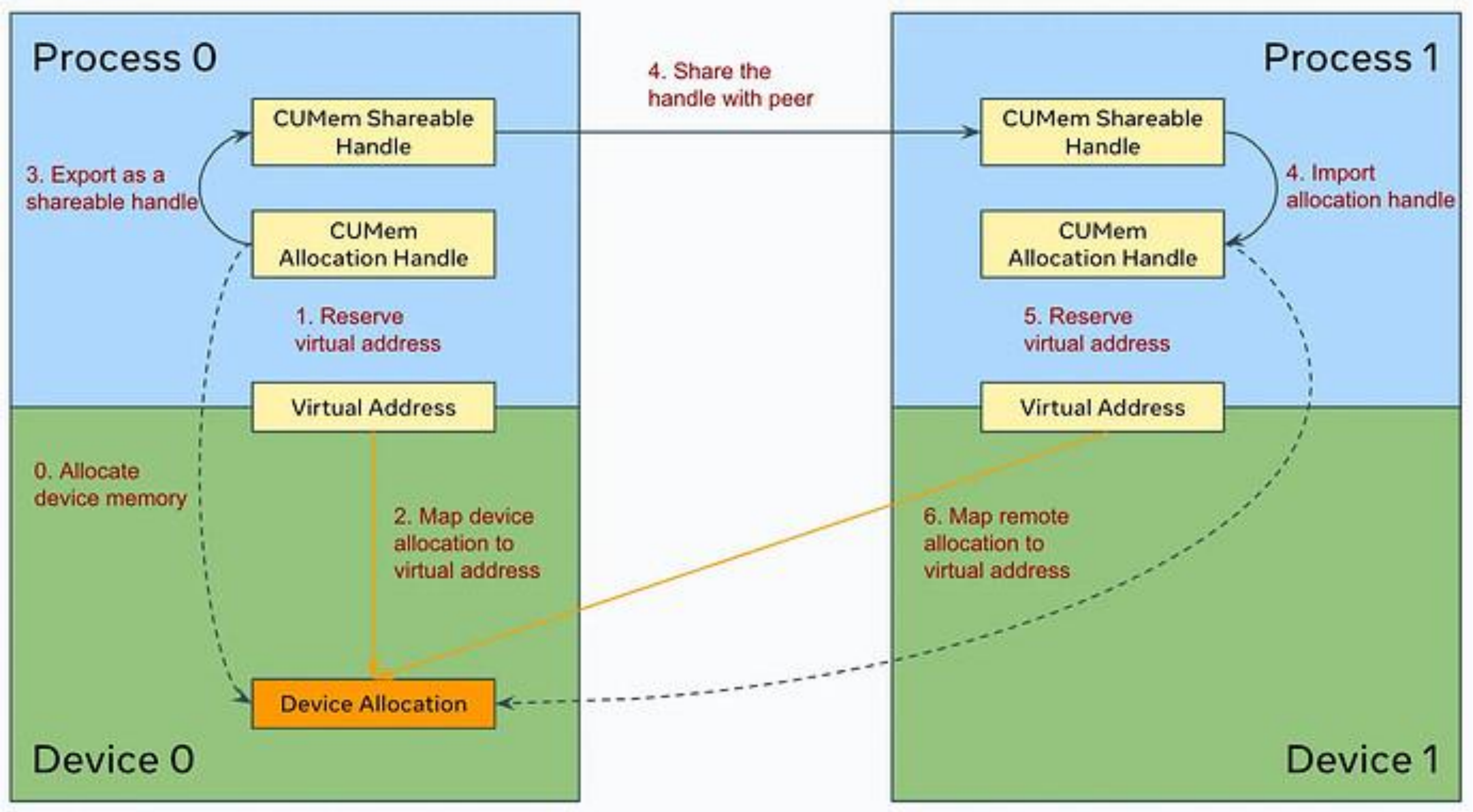
What about simply using more GPUs per model instance?



Symmetric Memory

Remote Memory Access in PyTorch

- idea of leveraging remote memory accesses much or widespread in CUDA Code
- PyTorch introduced the concept of Symmetric Memory: abstractions of these accesses natively in PyTorch (and e.g. Triton kernels)
- e.g. for overlapping compute/comm within kernels




```python
# allocate symmetric memory tensor
t = symm_mem.empty(4096, device="cuda")

# establish symmetric memory and obtain the handle
hdl = symm_mem.rendezvous(t, dist.group.WORLD)

# get peer buffer
peer_buf = hdl.get_buffer(next_rank, t.shape, t.dtype)

# pointers for kernels
hdl.buffer_ptrs
hdl.signal_pad_ptrs # for synchronization

# direct operation
torch.add(peer_buf, 1.0, out=peer_buf)
```


dev-discuss.pytorch.org/t/pytorch-symmetricmemory-harnessing-nvlink-programmability-with-ease/2798

Optimizing Graph Neural Networks

Concluding Remarks

- Model Optimizations for GNNs like as on “normal” GPU platforms and as for other models

- Grace Hopper
  - makes data transfers easier
  - opens optimization potential relevant for AI4Science

- otherwise, interplay of many design decisions crucial
  - representation of graphs
  - distribution of data
  - combination of checkpointing/offloading/etc.
  - combination of different parallelization strategies

NVIDIA.

NVIDIA