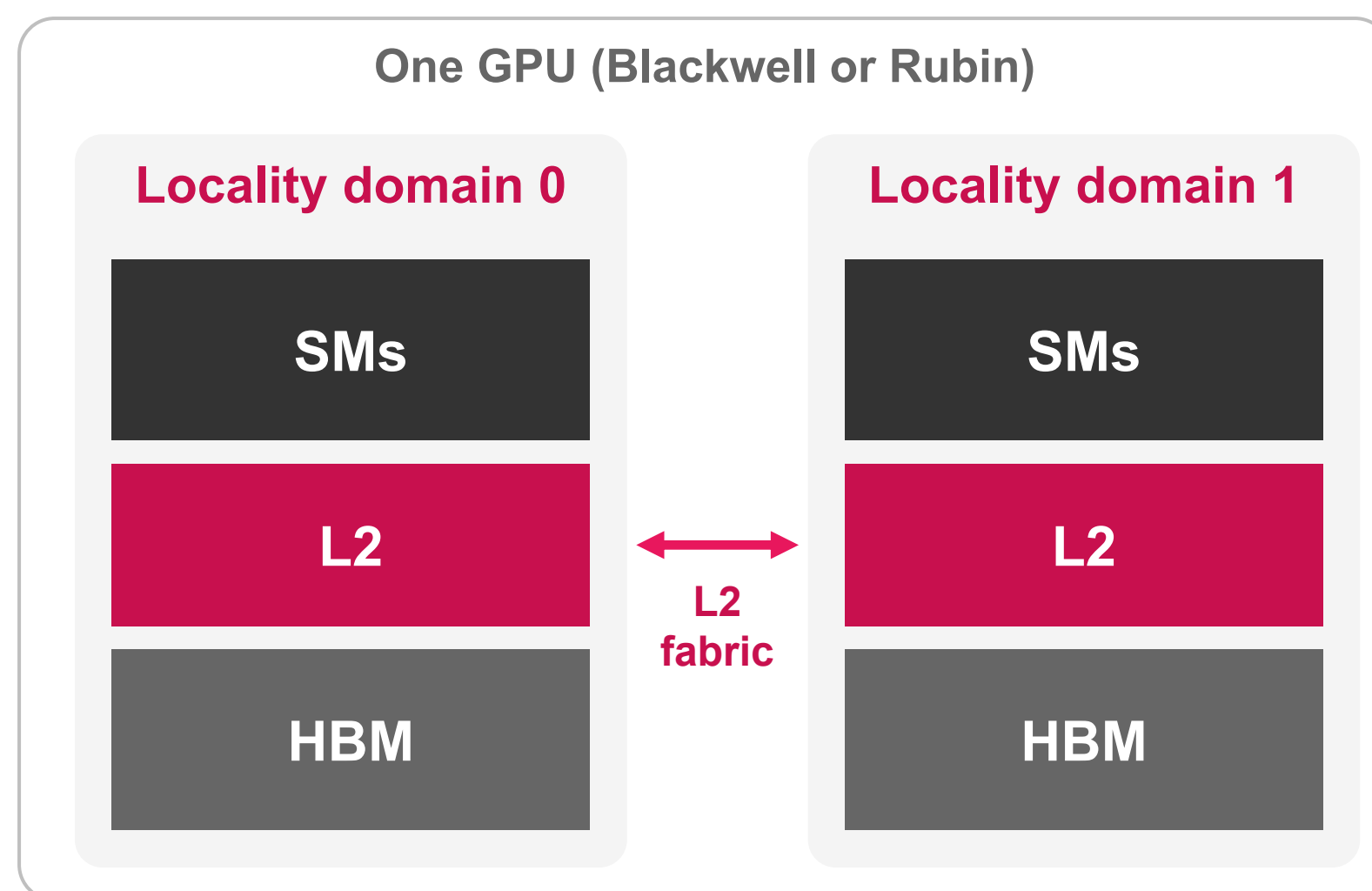




1 One GPU, two domains

A locality domain groups nearby **SMs**, **L2** and **HBM**. Access to the other domain stays coherent but crosses the L2 fabric.



Staying inside one domain skips the hop across the L2 fabric, which costs **latency** and **power** and, on Rubin, limits **bandwidth**.

Concept and CUDA APIs: [CUDA Programming Guide §4.7](#)

2 Why stay local



Latency · Blackwell, Rubin

Apps rewritten around local data: [scientific computing \(QUDA\)](#), GNNs, few others



Power · Blackwell, Rubin

Power-limited work that avoids cross-domain traffic: compute-bound LLM training and prefill



Bandwidth · Rubin

Memory-bound work: LLM inference, post-training, memory-bound parts of training

3 Two ways to localize

Process localization **MPS + MLOPart**

- Use it to localize with fewer code changes
- One process per domain; kernels unchanged
- DDP duplicates weights, FSDP adds comms
- Comms need an explicit un-localize step

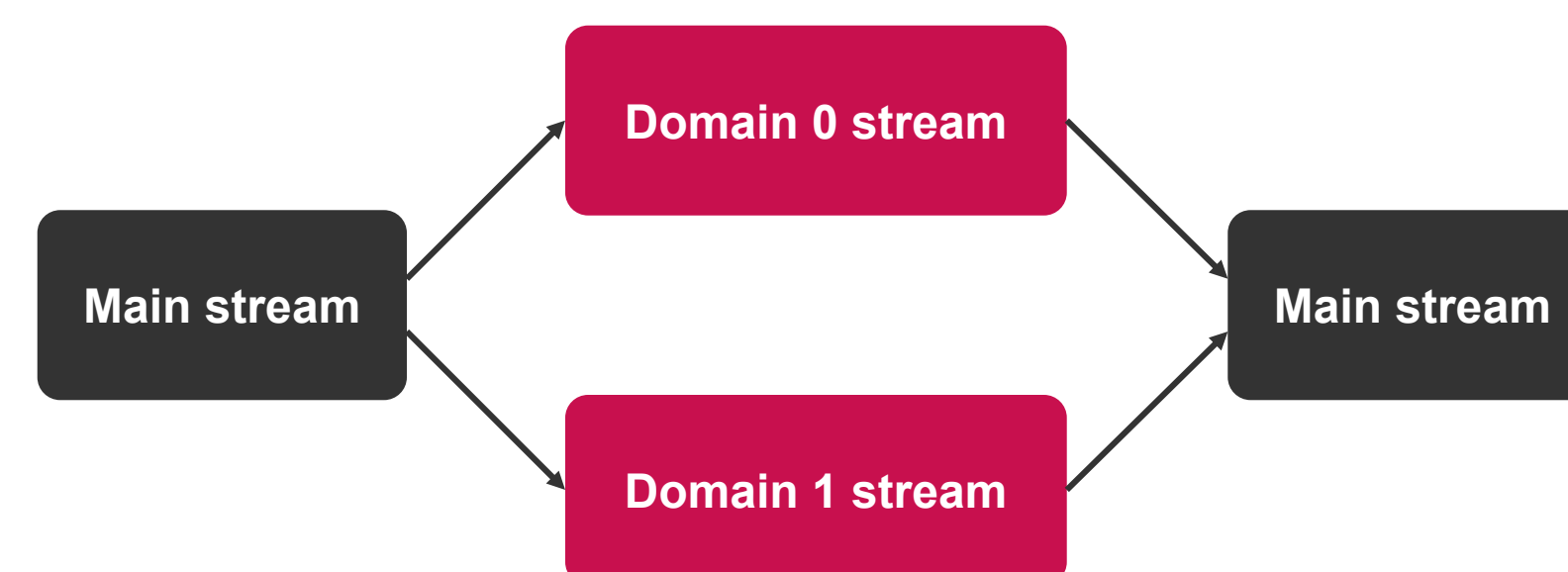
Docs: [MPS](#) · [MLOPart](#)

Programmatic localization **PyTorch APIs**

- Use it to localize only where it pays off, reducing overheads
- Localize per allocation and per stream
- Domains still read each other's memory
- Native un-localize for comms, more tuning room

PRs: [#198279](#) · [#199128](#) · [#199127](#)

4 Compute: green contexts



`torch.cuda.execute_on_streams` maps a function over per-domain inputs and returns the outputs, joined on the caller stream ([#199128](#)); per-domain split: [#198279](#), [CUDA guide §4.6](#)

```
from torch.cuda.green_contexts import (
    GreenContext, get_num_locality_domains)

n = get_num_locality_domains(0)
gcs = GreenContext.split(
    locality_domain_ids=tuple(range(n)),
    device_id=0)
streams = [gc.Stream() for gc in gcs]

xs = x.chunk(n) # one shard per domain
# localized execution
ys = torch.cuda.execute_on_streams(
    streams, lambda xi: xi @ w, xs)
y = torch.cat(ys) # non-localized execution
```

5 Memory: localized allocator

LocalizedAllocator backs a regular `MemPool` with memory on one domain ([#199127](#)).

```
alloc = torch.cuda.LocalizedAllocator(
    locality_domain_id=0, device="cuda:0")
with torch.cuda.device(alloc.device_id):
    pool = torch.cuda.MemPool(allocator=alloc, allocator())
    with torch.cuda.use_mem_pool(pool):
        x = torch.randn(N, device="cuda:0")
```

Placement only: pair the pool with a green-context stream on the same domain to localize compute too. Driver API: [localized VMM allocations](#).

6 torch.compile support

- Dynamo traces `use_mem_pool` and pool placement reaches Inductor ([#185057](#), merged)
- Event fork/join on user streams next to input mutations: fix in review ([#195573](#))
- Several pools in one CUDA graph ([#187929](#), merged); in CUDA graph trees: in progress ([#189230](#))

7 Comms need care

- Each domain drives **50% of NVLink**: use both domains to saturate it on Rubin ([NVSHMEM guide](#))
- Copy engines reach full speed only on localized buffers
- GPUDirect RDMA on Grace/Vera CPUs needs **non-localized** buffers: comms ops/kernels may require special handling
- NVLink SHARP: one multicast team per domain, reduce hierarchically

8 Measured gains

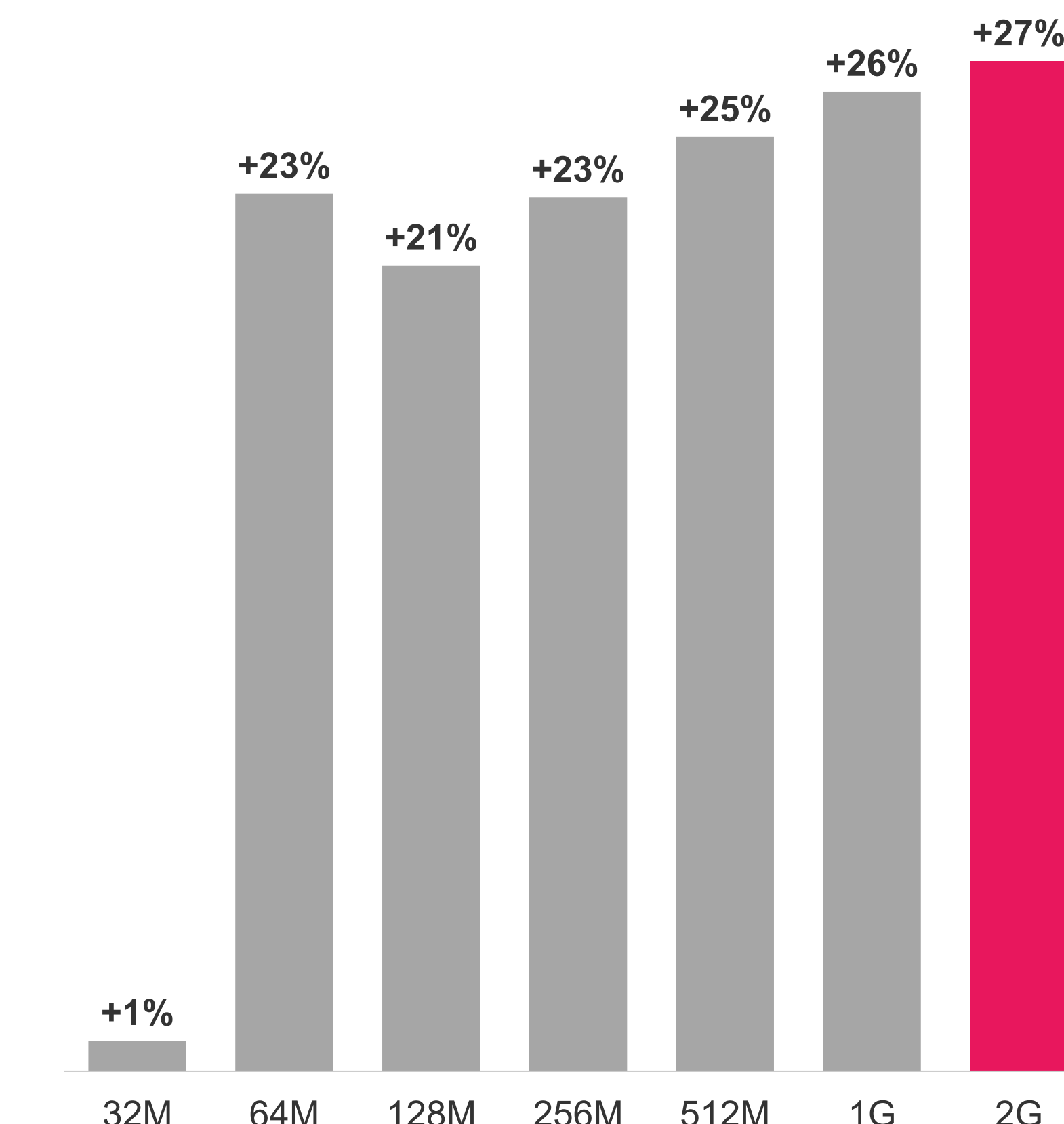
+27%

effective bandwidth, FP32
elementwise ops, 2 GiB
13.5 → 17.1 TB/s

-11%

GPU power, Rubin stream-add,
8 elements per thread

Speedup from localization, elementwise ops
Rubin, FP32, eager, by input size (32 MiB and up)



- Measured with CUDA graphs over a sequence of different elementwise ops, to minimize launch overhead and reproduce real L2 and instruction-cache effects
- Below 32 MiB, results vary from small losses to small gains, depending on quantization effects and fork/join overhead

Scan the QR code at top right

Download this poster as a PDF with every link, and rate the session.