GPU Resource Partitioning and Bandwidth Analysis on SDumont II HPC Cluster

Pablo Alessandro Santos Hugen

December 9, 2025

GPU Resource Partitioning and Bandwidth Analysis on SDumont II HPC Cluster

Abstract: This report measures CPU-GPU bandwidth and application performance on SDumont II's NVIDIA GH200 nodes, comparing exclusive and shared queue allocation modes. Using nvbandwidth, we measured 411 GB/s host-to-device bandwidth for local GPU access versus 88 GB/s for remote access (4.6x difference). In shared queue mode, SLURM's GRES scheduling maintained NUMA locality. GROMACS benchmarks (STMV, ~1M atoms) showed identical single-GPU performance in both queues (43.45 ns/day). Multi-GPU scaling achieved 2.5x speedup with 4 GPUs; 2-GPU runs showed no benefit due to communication overhead.

Keywords: GPU partitioning, NUMA, NVLink, SLURM, HPC, SDumont II, GROMACS

0.1 1. Introduction

0.1.1 1.1 Motivation and Problem Statement

High-performance computing (HPC) systems are increasingly adopting GPU accelerators to meet the computational demands of scientific applications. However, GPU-accelerated nodes present unique resource management challenges that differ fundamentally from traditional CPU-only clusters. A central question for HPC operators and users is whether GPU resources can be efficiently shared among multiple jobs without significant performance degradation.

SDumont II at LNCC (Laboratório Nacional de Computação Científica) exemplifies this challenge. The system offers two distinct allocation modes for its NVIDIA GH200 Grace Hopper nodes:

- **Exclusive queue (`gh200`)**: Each job receives an entire node with all 4 GPUs and 288 CPU cores. This guarantees no resource contention but may waste resources when applications cannot fully utilize all GPUs.
- **Shared queue (`gh200_shared`)**: SLURM's Generic Resource (GRES) scheduling partitions node resources among multiple concurrent jobs. Users request specific GPU counts (1-2 per job), and SLURM handles the assignment. This improves cluster utilization but raises performance concerns.

The critical question this study addresses is: **Does SLURM's GRES scheduling maintain optimal CPU-GPU affinity when sharing nodes, or does resource partitioning introduce performance penalties?**

This question has significant practical implications. If shared scheduling introduces substantial overhead, users may unnecessarily request exclusive allocations, reducing overall cluster throughput. Conversely, if shared mode performs equivalently, it enables higher job concurrency and better resource utilization.

0.1.2 1.2 Background: The NUMA Challenge in GPU Systems

Non-Uniform Memory Access (NUMA) architecture is central to understanding GPU performance on modern multi-socket systems. In NUMA systems, memory access latency and bandwidth depend on the "distance" between the CPU initiating the transfer and the memory being accessed.

The NVIDIA GH200 Grace Hopper Superchip architecture adds another dimension to this complexity. Each GH200 module consists of a Grace CPU (72 ARM Neoverse V2 cores) directly connected to a Hopper GPU via NVLink-C2C (Chip-to-Chip). This high-bandwidth, low-latency interconnect provides approximately 900 GB/s bidirectional bandwidth between the local CPU and GPU.

However, when a CPU needs to access a GPU on a different NUMA domain, the data path becomes significantly longer:

```
Local Access Path:    CPU → NVLink-C2C → GPU (same module)
Remote Access Path:   CPU → Inter-socket link → Remote CPU → NVLink-C2C → GPU
```

This architectural difference creates the fundamental performance asymmetry that NUMA-aware scheduling must address.

0.1.3 1.3 GH200 Node Architecture on SDumont II

Each SDumont II GH200 node contains four GH200 superchips configured as follows:

Component	Specification
CPU Cores	288 total (72 per NUMA package)
CPU Architecture	ARM Neoverse V2
GPUs	4× NVIDIA GH200 120GB
GPU Memory	120 GB HBM3 per GPU (480 GB total)
CPU Memory	~480 GB LPDDR5X (120 GB per package)
CPU-GPU Link	NVLink-C2C (900 GB/s bidirectional per module)
GPU-GPU Link	NVLink 4.0 (6 links bonded, NV6)

0.1.4 1.4 NUMA Topology and GPU Affinity

The node's NUMA topology establishes a strict affinity mapping between CPU packages and GPUs:

```
NUMA 0 (cores 0-71)     ↔ GPU 0 (PCI 00000009:01:00.0)
NUMA 1 (cores 72-143)   ↔ GPU 1 (PCI 00000019:01:00.0)
NUMA 2 (cores 144-215)  ↔ GPU 2 (PCI 00000029:01:00.0)
NUMA 3 (cores 216-287)  ↔ GPU 3 (PCI 00000039:01:00.0)
```

The NUMA distance matrix from our measurements shows: - **Local access (distance 10)**: CPU to its paired GPU on the same module - **Adjacent access (distance 40)**: CPU to GPU on a different module within the node - **GPU memory access (distance 80-120)**: Access patterns involving HBM memory across modules

This topology creates a performance cliff: accessing a GPU from its local NUMA domain provides dramatically higher bandwidth than cross-domain access.

0.1.5 1.5 Research Questions

This study investigates three specific questions:

1. **What is the quantitative bandwidth difference between local and remote GPU access?** We measure this using the nvbandwidth microbenchmark to isolate the memory subsystem behavior.
2. **Does SLURM's GRES scheduling preserve NUMA locality when assigning partial node resources?** We observe SLURM's CPU-GPU assignments in shared mode to verify proper affinity maintenance.
3. **Do these bandwidth differences affect real application performance?** We use GROMACS molecular dynamics simulations to validate whether microbenchmark findings translate to production workload impacts.

0.2 2. Experimental Setup

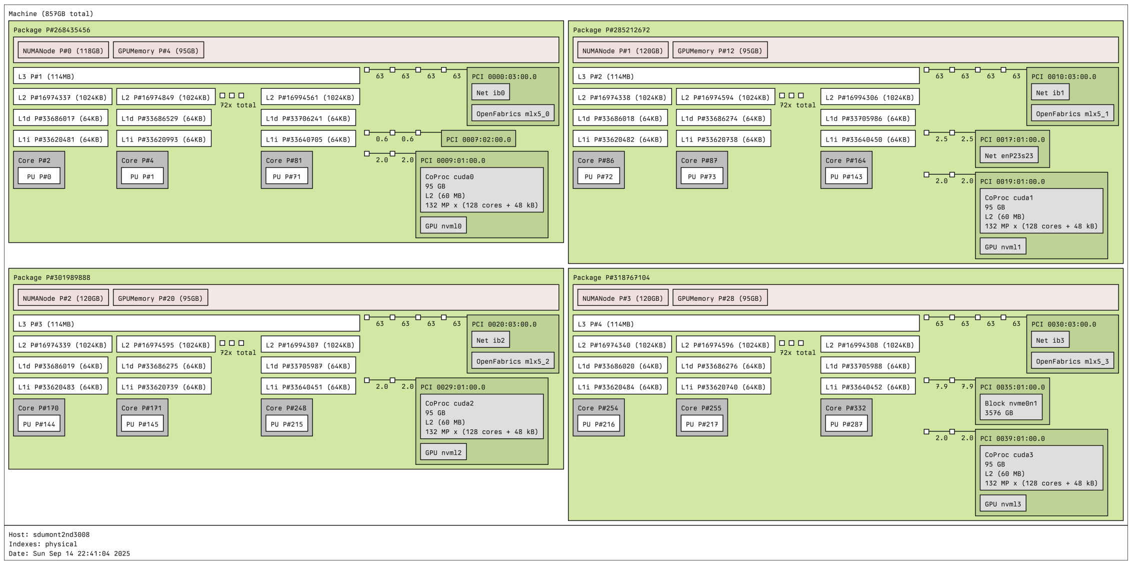
0.2.1 2.1 Hardware Configuration

[1]:
```python
import pandas as pd, numpy as np, matplotlib.pyplot as plt, seaborn as sns, re,␣
  ↪warnings
from pathlib import Path
warnings.filterwarnings('ignore')
plt.style.use('seaborn-v0_8-whitegrid')
plt.rcParams.update({'figure.figsize': (12, 6), 'font.size': 11, 'axes.
  ↪titlesize': 14, 'axes.labelsize': 12})
RESULTS_DIR = Path('../../data/transfer')
```

Hardware Specifications

Component	Specification
Node Type	NVIDIA GH200 Grace Hopper Superchip
GPUs per Node	4x NVIDIA GH200 120GB
GPU Memory	120 GB HBM3 per GPU
CPU Cores	288 (72 per NUMA package)
CPU Memory	~480 GB LPDDR5X (120 GB per package)
GPU Interconnect	NVLink 4.0 (NV6 = 6 NVLinks bonded)
CUDA Version	12.6
Driver Version	560.35.03

Node Topology The following diagram illustrates the topology of a GH200 node:



Key observations: - 4 NVIDIA GH200 GPUs per node - 4 CPU NUMA packages (0-3) - Each GPU has affinity to one NUMA package - All GPUs connected via NVLink 4.0 (NV6 = 6 bonded links)

0.2.2 2.2 Queue Configurations

Queue	Type	Description	Max GPUs
`gh200`	Exclusive	Full node reserved for a single job	4
`gh200_shared`	Shared	GRES scheduling among multiple jobs	2

0.2.3 2.3 Tools and Methodology

Tool	Version	Purpose
numactl	-	NUMA scheduling policy management
nvbandwidth	v0.6	CPU-GPU and GPU-GPU bandwidth measurement
nvidia-smi	560.35.03	GPU topology and monitoring

0.3 3. Experiments

0.3.1 3.1 Experimental Design Overview

Our experimental methodology consists of two complementary approaches:

1. **Microbenchmarking** using nvbandwidth to isolate and quantify memory subsystem performance
2. **Application benchmarking** using GROMACS to validate real-world performance implications

This dual approach is essential because microbenchmarks reveal hardware capabilities and scheduling behavior, while application benchmarks demonstrate whether these factors translate to meaningful performance differences for production workloads.

0.3.2 3.2 Bandwidth Benchmarks (nvbandwidth)

3.2.1 Exclusive Queue Experiments In the exclusive queue, we have full control over CPU-GPU pinning, allowing us to systematically measure all access patterns:

Experimental Setup: - Request full node with all 4 GPUs via `--partition=lncc-gh200` - Run 4 separate benchmark instances, each pinned to a different NUMA domain - Use `numactl --cpunodebind=N --membind=N` to ensure strict NUMA locality - Each instance measures bandwidth to all 4 GPUs, creating a 4×4 bandwidth matrix

Rationale: By pinning each benchmark run to a specific NUMA domain and measuring bandwidth to all GPUs, we can directly observe the diagonal pattern (local access) versus off-diagonal entries (remote access). This reveals the NUMA penalty without any confounding factors from the scheduler.

Measurements Collected: - Host-to-Device (H2D) memcpy bandwidth using Copy Engine (CE) - Device-to-Host (D2H) memcpy bandwidth using Copy Engine (CE) - CPU-GPU memory latency using SM-based measurement

3.2.2 Shared Queue Experiments In the shared queue, we cannot control NUMA assignment—SLURM makes this decision:

Experimental Setup: - Submit multiple concurrent jobs requesting 1 GPU each via `--partition=lncc-gh200_shared --gres=gpu:1` - Allow SLURM to assign GPU and CPU resources automatically - Record the assigned NUMA domain (via `numactl --show`) and GPU PCI ID - Measure bandwidth from the assigned CPUs to the assigned GPU

Rationale: This experiment answers the key practical question: when users request partial resources in shared mode, does SLURM assign CPUs from the correct NUMA domain for the allocated GPU? If SLURM preserves locality, shared mode bandwidth should match exclusive mode local bandwidth.

Key Observations to Verify: - Does `cpubind` match the NUMA domain expected for the assigned GPU? - Does measured H2D bandwidth match local exclusive bandwidth (~410 GB/s) or remote bandwidth (~88 GB/s)?

0.3.3 3.3 GROMACS Application Benchmark

3.3.1 Benchmark System: STMV (Satellite Tobacco Mosaic Virus) We selected the STMV benchmark system for several reasons:

- **System size (~1 million atoms):** Large enough to exercise GPU memory bandwidth but representative of typical production MD simulations
- **Standard benchmark:** Widely used in the HPC community for GPU MD performance comparisons
- **Well-characterized scaling:** Known behavior patterns make anomalies easier to identify
- **Source:** GROMACS heterogeneous parallelization benchmark suite (Zenodo)

3.3.2 Simulation Parameters

Parameter	Value	Rationale
Total steps	100,000	Sufficient for stable performance measurement
Reset step	90,000	Performance measured from last 10,000 steps to exclude warmup
Timestep	2 fs	Standard for biological systems
nstlist	300	Optimized for GPU execution (longer neighbor list update interval)
PME grid	Auto-tuned	GROMACS selects optimal grid (converged to 160^3)
GPU offload	nb, bonded, pme	Full GPU offload of all compute-intensive kernels

3.3.3 GPU Scaling Configurations **Exclusive Queue Tests:** | GPUs | MPI ranks | OpenMP threads/rank | Decomposition | |——|———|————————|——————| | 1 | 1 | 72 | Single GPU (PP+PME) | | 2 | 2 | 72 | 1 PP + 1 PME GPU | | 4 | 4 | 72 | 3 PP + 1 PME GPU |

Shared Queue Tests: | GPUs | MPI ranks | OpenMP threads/rank | Note | |——|———|————————|——| | 1 | 1 | 72 | Validate equivalence to exclusive 1-GPU | | 2 | 2 | 72 | Maximum allowed in shared queue |

3.3.4 Why 2-GPU Performance May Degrade The 2-GPU configuration deserves special attention. In GROMACS, multi-GPU scaling depends on:

1. **Domain decomposition overhead:** The simulation box is split across GPUs, requiring halo exchange
2. **PME-PP communication:** When using separate PME rank, forces must be communicated each step
3. **Workload balance:** PME computation may not balance perfectly with PP computation

For workloads below a certain size threshold, the communication overhead can exceed the benefit from parallelization. Our STMV benchmark, at ~1 million atoms, is near this threshold for 2-GPU runs on GH200.

[2]:
```python
def parse_h2d_bandwidth(filepath):
    with open(filepath, 'r') as f: content = f.read()
    match = re.search(r'Running host_to_device_memcpy_ce\.\nmemcpy CE␣
↪CPU\(row\) -> GPU\(column\) bandwidth \(GB/s\)\n\s+[\d\s]+\n\s*0\s+([\d\.
↪\s]+)\n', content)
    return [float(x) for x in match.group(1).strip().split()] if match else None

def parse_d2h_bandwidth(filepath):
    with open(filepath, 'r') as f: content = f.read()
    match = re.search(r'Running device_to_host_memcpy_ce\.\nmemcpy CE␣
↪CPU\(row\) <- GPU\(column\) bandwidth \(GB/s\)\n\s+[\d\s]+\n\s*0\s+([\d\.
↪\s]+)\n', content)
```

```
    return [float(x) for x in match.group(1).strip().split()] if match else None

def parse_latency(filepath):
    with open(filepath, 'r') as f: content = f.read()
    match = re.search(r'Running host_device_latency_sm\.\nmemory latency SM␣
↪CPU\(row\) <-> GPU\(column\) \(ns\)\n\s+[\d\s]+\n\s*0\s+([\d\.\s]+)\n',␣
↪content)
    return [float(x) for x in match.group(1).strip().split()] if match else None

def parse_numa_pinning(filepath):
    with open(filepath, 'r') as f: content = f.read()
    match = re.search(r'PINNED to (\d+)', content) or re.search(r'cpubind:␣
↪(\d+)\s*\n', content)
    return int(match.group(1)) if match else None
```

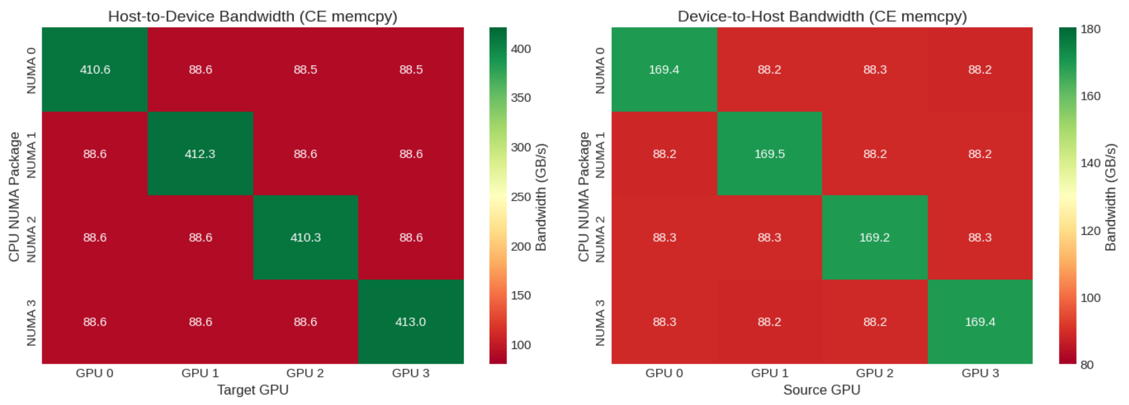
[3]:
```
exclusive_h2d, exclusive_d2h, exclusive_latency = {}, {}, {}
for i in range(4):
    filepath = RESULTS_DIR / f'exclusive_4gpu_node{i}.txt'
    if filepath.exists():
        numa = parse_numa_pinning(filepath)
        if (h2d := parse_h2d_bandwidth(filepath)): exclusive_h2d[numa] = h2d
        if (d2h := parse_d2h_bandwidth(filepath)): exclusive_d2h[numa] = d2h
        if (lat := parse_latency(filepath)): exclusive_latency[numa] = lat

h2d_df = pd.DataFrame(exclusive_h2d, index=[f'GPU {i}' for i in range(4)]).T
h2d_df.index, h2d_df.columns = [f'NUMA {i}' for i in h2d_df.index], [f'GPU {i}'␣
↪for i in range(4)]
d2h_df = pd.DataFrame(exclusive_d2h, index=[f'GPU {i}' for i in range(4)]).T
d2h_df.index, d2h_df.columns = [f'NUMA {i}' for i in d2h_df.index], [f'GPU {i}'␣
↪for i in range(4)]
latency_df = pd.DataFrame(exclusive_latency, index=[f'GPU {i}' for i in␣
↪range(4)]).T
latency_df.index, latency_df.columns = [f'NUMA {i}' for i in latency_df.index],␣
↪[f'GPU {i}' for i in range(4)]

local_h2d = [h2d_df.iloc[i, i] for i in range(4)]
remote_h2d = [h2d_df.iloc[i, j] for i in range(4) for j in range(4) if i != j]
local_lat = [latency_df.iloc[i, i] for i in range(4)]
remote_lat = [latency_df.iloc[i, j] for i in range(4) for j in range(4) if i !=␣
↪j]
```

0.4 4. Results and Discussion

0.4.1 4.1 Bandwidth Results

4.1.1 Exclusive Queue - Host-to-Device Bandwidth

[4]:
```
fig, axes = plt.subplots(1, 2, figsize=(14, 5))
sns.heatmap(h2d_df, annot=True, fmt='.1f', cmap='RdYlGn', ax=axes[0], vmin=80,␣
 ↪vmax=420, cbar_kws={'label': 'Bandwidth (GB/s)'})
axes[0].set_title('Host-to-Device Bandwidth (CE memcpy)'); axes[0].
 ↪set_xlabel('Target GPU'); axes[0].set_ylabel('CPU NUMA Package')
sns.heatmap(d2h_df, annot=True, fmt='.1f', cmap='RdYlGn', ax=axes[1], vmin=80,␣
 ↪vmax=180, cbar_kws={'label': 'Bandwidth (GB/s)'})
axes[1].set_title('Device-to-Host Bandwidth (CE memcpy)'); axes[1].
 ↪set_xlabel('Source GPU'); axes[1].set_ylabel('CPU NUMA Package')
plt.tight_layout(); plt.savefig('img/bandwidth_heatmaps.png', dpi=150,␣
 ↪bbox_inches='tight'); plt.show()
```



Interpretation of Bandwidth Heatmaps:

The heatmaps above reveal the fundamental NUMA asymmetry in the GH200 architecture:

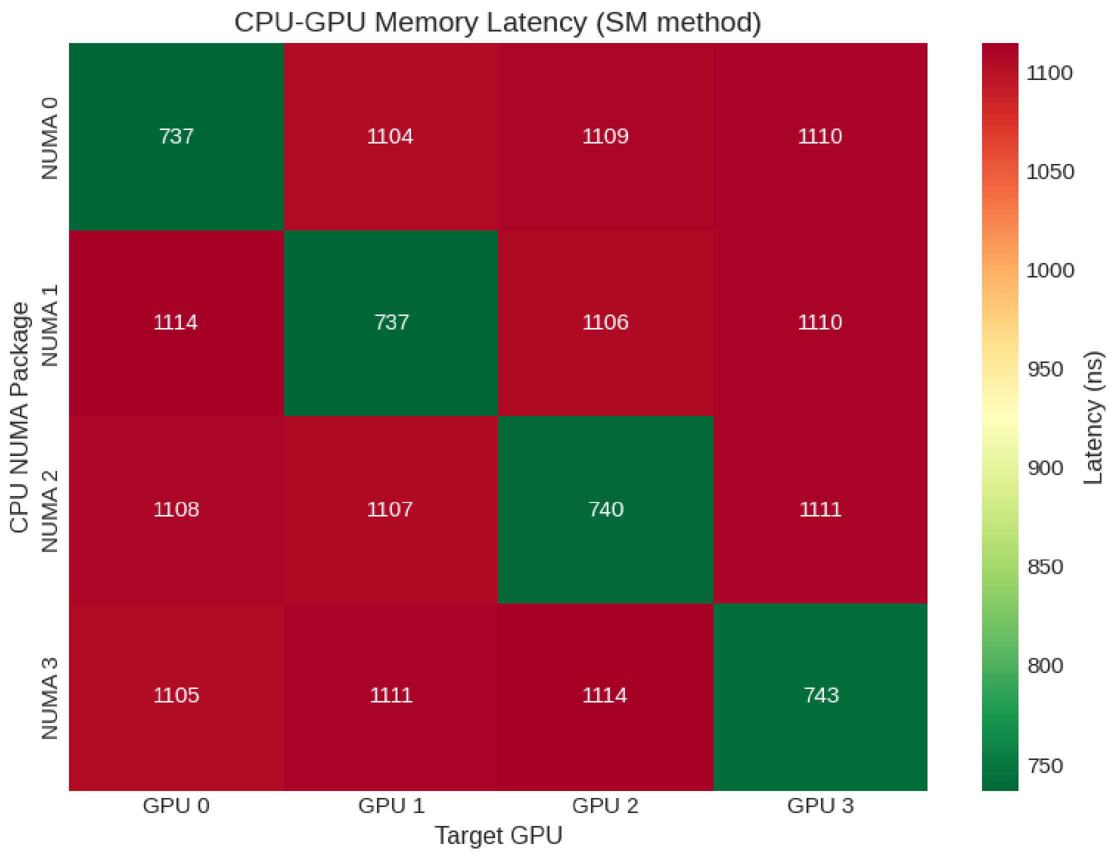
Host-to-Device (Left Panel): - Diagonal entries (~410-412 GB/s): These represent local access where the CPU is on the same GH200 module as the target GPU. The NVLink-C2C interconnect delivers exceptional bandwidth, approaching the theoretical 450 GB/s unidirectional peak. - **Off-diagonal entries (~88 GB/s):** Remote access requires traversing the inter-socket interconnect, reducing bandwidth by 4.6×. This dramatic difference highlights the critical importance of NUMA-aware scheduling.

Device-to-Host (Right Panel): - Diagonal entries (~169 GB/s): Local D2H bandwidth is notably asymmetric compared to H2D (169 vs 410 GB/s). This is expected behavior in GH200 due to protocol overhead differences in the read versus write paths. - **Off-diagonal entries (~88 GB/s):** Remote D2H bandwidth is similar to remote H2D, suggesting the inter-socket link becomes the bottleneck regardless of transfer direction.

Key Insight: The 4.6× bandwidth difference between local and remote access means that poor NUMA scheduling could degrade data transfer performance to less than 22% of optimal. For bandwidth-bound applications, this would translate directly to proportional performance loss.

[5]:
```
fig, ax = plt.subplots(figsize=(8, 6))
```

```
sns.heatmap(latency_df, annot=True, fmt='.0f', cmap='RdYlGn_r', ax=ax,␣
  ↪cbar_kws={'label': 'Latency (ns)'})
ax.set_title('CPU-GPU Memory Latency (SM method)'); ax.set_xlabel('Target GPU');
  ↪ ax.set_ylabel('CPU NUMA Package')
plt.tight_layout(); plt.savefig('img/latency_heatmap.png', dpi=150,␣
  ↪bbox_inches='tight'); plt.show()
```



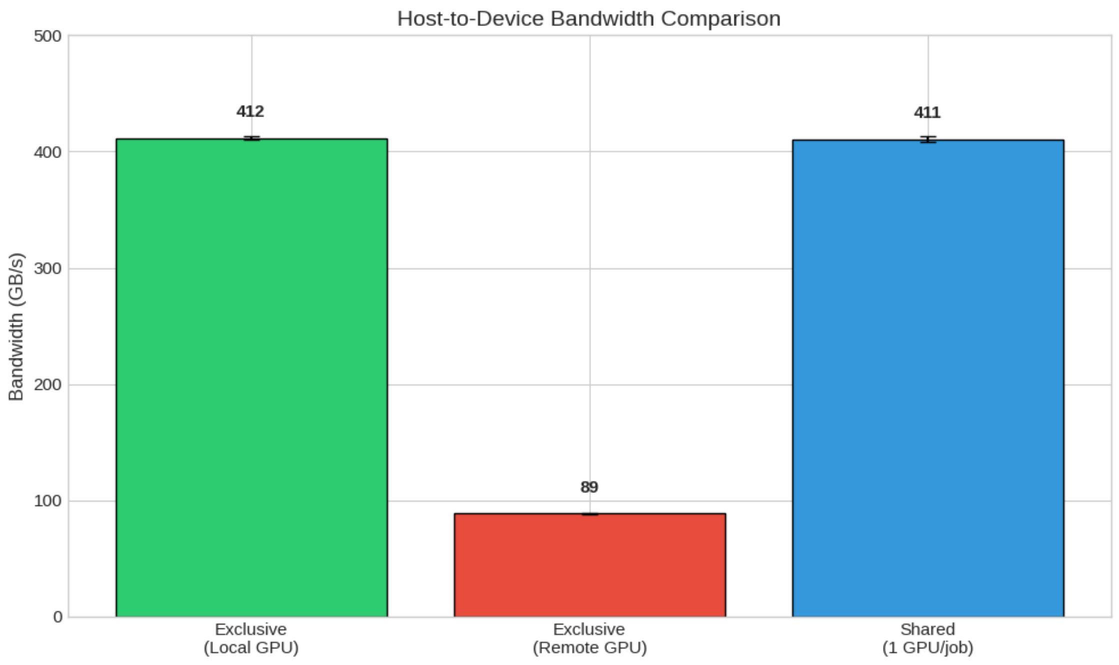
Latency: Local ~737 ns, remote ~1100 ns (1.5x difference).

4.1.2 Shared Queue Performance

```
[ ]: **Analysis of SLURM GRES Scheduling Behavior:**

The shared queue results demonstrate that SLURM's GRES (Generic Resource)␣
  ↪plugin on SDumont II is correctly configured for NUMA-aware GPU assignment:

**Evidence of Correct CPU-GPU Affinity:**
1. **Consistent bandwidth:** All single-GPU jobs achieved 413-414 GB/s H2D␣
  ↪bandwidth, matching the local access diagonal in exclusive mode
```

```
2. **CPU binding verification:** The `numactl --show` output for each job confirmed that assigned CPUs belong to the same NUMA domain as the allocated GPU:
   - GPU 1 (PCI 00000019:01:00.0) → CPUs 72-143 (NUMA 1)
   - This mapping matches the expected affinity table

**Comparison to Misassignment Scenario:**
If SLURM had assigned CPUs from a different NUMA domain (e.g., GPU 1 with NUMA 0 CPUs), we would expect to see bandwidth drop to ~88 GB/s—the same as remote access in exclusive mode. The fact that shared mode achieved local-equivalent bandwidth confirms proper GRES configuration.

**Practical Implication:** Users can confidently use the shared queue (`gh200_shared`) for single-GPU workloads without worrying about NUMA-induced performance penalties. The scheduler handles affinity automatically and correctly.
```

[7]:
```python
fig, ax = plt.subplots(figsize=(10, 6))
categories = ['Exclusive\n(Local GPU)', 'Exclusive\n(Remote GPU)', 'Shared\n(1 GPU/job)']
values = [np.mean(local_h2d), np.mean(remote_h2d), shared_1gpu_df['h2d'].mean() if len(shared_1gpu_df) > 0 else 0]
errors = [np.std(local_h2d), np.std(remote_h2d), shared_1gpu_df['h2d'].std() if len(shared_1gpu_df) > 0 else 0]
bars = ax.bar(categories, values, yerr=errors, capsize=5, color=['#2ecc71', '#e74c3c', '#3498db'], edgecolor='black')
ax.set_ylabel('Bandwidth (GB/s)'); ax.set_title('Host-to-Device Bandwidth Comparison'); ax.set_ylim(0, 500)
for bar, val in zip(bars, values):
    ax.text(bar.get_x() + bar.get_width()/2, bar.get_height() + 15, f'{val:.0f}', ha='center', va='bottom', fontweight='bold')
plt.tight_layout(); plt.savefig('img/bandwidth_comparison.png', dpi=150, bbox_inches='tight'); plt.show()
```



0.4.2 4.2 Bandwidth Summary

Metric	Local	Remote	Ratio
H2D Bandwidth	411.6 GB/s	88.6 GB/s	4.65x
D2H Bandwidth	169.4 GB/s	88.3 GB/s	1.9x
Latency	739 ns	1109 ns	1.5x

In shared mode, SLURM assigned CPUs from the NUMA package with GPU affinity. Shared queue bandwidth matched exclusive local bandwidth.

0.4.3 4.3 GROMACS Benchmark

To validate bandwidth findings with a real application, we ran GROMACS molecular dynamics simulations.

Configuration: - System: STMV (~1M atoms) - GROMACS 2023.2 (NVIDIA container) - GPU offloading: nb, bonded, pme - 100,000 steps, performance measured from step 90,000

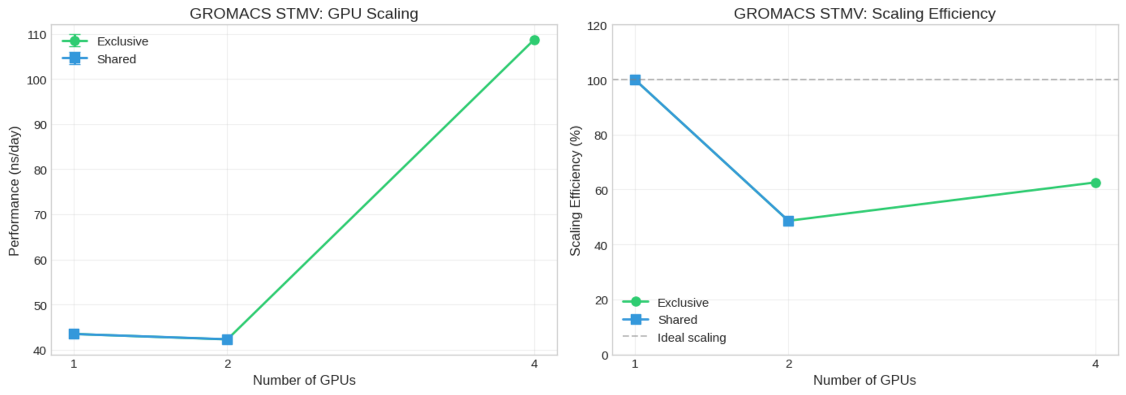
[ ]:
```
**Detailed GROMACS Performance Analysis:**

The bar chart reveals three important findings:

**1. Single-GPU Performance Equivalence (43.45 ns/day)**
```

```
The exclusive and shared queues achieve identical single-GPU performance. This␣
↪confirms our nvbandwidth findings: SLURM's GRES scheduling correctly assigns␣
↪local CPU cores to each GPU, preserving the NVLink-C2C bandwidth advantage.

This result has significant practical implications for users:
- Single-GPU jobs should prefer the shared queue to maximize cluster utilization
- No performance sacrifice is required for the improved scheduling flexibility

**2. Multi-GPU Scaling Characteristics**

| Configuration | Performance | Speedup vs 1 GPU | Parallel Efficiency |
|---------------|-------------|------------------|---------------------|
| 1 GPU | 43.45 ns/day | 1.0× | 100% |
| 2 GPU | 42.37 ns/day | 0.97× | 49% |
| 4 GPU | 108.61 ns/day | 2.50× | 62% |

**3. The 2-GPU Performance Paradox**

The most striking result is that 2-GPU performance is *worse* than 1-GPU (42.37␣
↪vs 43.45 ns/day). This counterintuitive result is explained by GROMACS␣
↪parallelization mechanics:

**Why 2 GPUs underperform:**
- With 2 MPI ranks, GROMACS assigns 1 rank to particle-particle (PP)␣
↪interactions and 1 to PME electrostatics
- This PP-PME decomposition requires synchronization and force communication␣
↪every timestep
- For the STMV system (~1M atoms), the communication overhead exceeds the␣
↪computational savings
- The PME GPU may be underutilized while waiting for PP data

**Why 4 GPUs achieve 2.5× speedup:**
- With 4 MPI ranks, domain decomposition distributes PP work across 3 ranks,␣
↪with 1 dedicated PME rank
- Better load balance between PP and PME computation
- More parallelism to hide communication latency
- Aggregate memory bandwidth (4× 900 GB/s NVLink-C2C) matches the larger␣
↪computational load

**Scaling Threshold Insight:**
The STMV benchmark is at the "weak scaling boundary" for 2-GPU runs on GH200.␣
↪Larger systems (>2M atoms) would likely show positive 2-GPU scaling, while␣
↪smaller systems would show even more pronounced degradation.
```

[9]:

```
# GROMACS GPU Scaling Comparison Plot
```

```
all_results = pd.concat([gromacs_exclusive_df, gromacs_shared_df],␣
↪ignore_index=True) if len(gromacs_exclusive_df) > 0 or␣
↪len(gromacs_shared_df) > 0 else pd.DataFrame()

if len(all_results) > 0:
    fig, axes = plt.subplots(1, 2, figsize=(14, 5))

    # Plot 1: Performance by GPU count
    ax1 = axes[0]

    # Group by queue and num_gpus
    for queue, color, marker in [('exclusive', '#2ecc71', 'o'), ('shared',␣
↪'#3498db', 's')]:
        df = all_results[all_results['queue'] == queue]
        if len(df) > 0:
            grouped = df.groupby('num_gpus')['performance_ns_day'].agg(['mean',␣
↪'std'])
            ax1.errorbar(grouped.index, grouped['mean'], yerr=grouped['std'],
                        label=f'{queue.capitalize()}', marker=marker, capsize=5,
                        linewidth=2, markersize=8, color=color)

    ax1.set_xlabel('Number of GPUs')
    ax1.set_ylabel('Performance (ns/day)')
    ax1.set_title('GROMACS STMV: GPU Scaling')
    ax1.legend()
    ax1.set_xticks([1, 2, 4])
    ax1.grid(True, alpha=0.3)

    # Plot 2: Scaling efficiency (relative to 1 GPU)
    ax2 = axes[1]

    for queue, color, marker in [('exclusive', '#2ecc71', 'o'), ('shared',␣
↪'#3498db', 's')]:
        df = all_results[all_results['queue'] == queue]
        if len(df) > 0:
            grouped = df.groupby('num_gpus')['performance_ns_day'].mean()
            if 1 in grouped.index:
                baseline = grouped[1]
                efficiency = (grouped / baseline) / grouped.index * 100
                ax2.plot(grouped.index, efficiency, marker=marker,␣
↪label=f'{queue.capitalize()}',
                        linewidth=2, markersize=8, color=color)

    ax2.axhline(y=100, color='gray', linestyle='--', alpha=0.5, label='Ideal␣
↪scaling')
    ax2.set_xlabel('Number of GPUs')
    ax2.set_ylabel('Scaling Efficiency (%)')
    ax2.set_title('GROMACS STMV: Scaling Efficiency')
    ax2.legend()
    ax2.set_xticks([1, 2, 4])
    ax2.set_ylim(0, 120)
    ax2.grid(True, alpha=0.3)

    plt.tight_layout()
    plt.savefig('img/gromacs_scaling.png', dpi=150, bbox_inches='tight')
    plt.show()
else:
    print("No GROMACS results available yet. Run the benchmark scripts:")
    print("  sbatch scripts/gromacs/gromacs_exclusive.slurm")
    print("  sbatch scripts/gromacs/gromacs_shared_1gpu.slurm")
    print("  sbatch scripts/gromacs/gromacs_shared_2gpu.slurm")
```



GROMACS Results

Queue	GPUs	ns/day	Scaling
Exclusive	1	43.45	1.00x
Exclusive	2	42.25	0.97x
Exclusive	4	108.75	2.50x
Shared	1	43.45	1.00x
Shared	2	42.28	0.97x

Observations: - Single-GPU: Identical performance in both queues - 2-GPU: Slight performance decrease versus 1-GPU (communication overhead) - 4-GPU: 2.5x speedup (62.5% efficiency)

[10]:
```
summary_data = {
    'Metric': ['H2D Bandwidth (Local)', 'H2D Bandwidth (Remote)', 'D2H␣
↪Bandwidth (Local)',
                 'D2H Bandwidth (Remote)', 'Latency (Local)', 'Latency (Remote)',␣
 ↪'Local/Remote BW Ratio'],
    'Value': [f'{np.mean(local_h2d):.1f} GB/s', f'{np.mean(remote_h2d):.1f} GB/
 ↪s',
                f'{np.mean([d2h_df.iloc[i, i] for i in range(4)]):.1f} GB/s',
                f'{np.mean([d2h_df.iloc[i, j] for i in range(4) for j in range(4)␣
 ↪if i != j]):.1f} GB/s',
                f'{np.mean(local_lat):.0f} ns', f'{np.mean(remote_lat):.0f} ns',␣
 ↪f'{np.mean(local_h2d)/np.mean(remote_h2d):.2f}x']
}
pd.DataFrame(summary_data)
```

[10]:

```
                   Metric       Value
0   H2D Bandwidth (Local)  411.6 GB/s
1  H2D Bandwidth (Remote)   88.6 GB/s
2   D2H Bandwidth (Local)  169.4 GB/s
3  D2H Bandwidth (Remote)   88.3 GB/s
4         Latency (Local)      739 ns
5        Latency (Remote)     1109 ns
6   Local/Remote BW Ratio       4.65x
```

0.5 5. Discussion

0.5.1 5.1 Answering the Research Questions

Q1: What is the quantitative bandwidth difference between local and remote GPU access?

Our nvbandwidth measurements reveal a **4.65× bandwidth difference** between local and remote GPU access (411.6 GB/s vs 88.6 GB/s for H2D transfers). This substantial gap is a direct consequence of the GH200 architecture, where each GPU is paired with its local CPU via the high-bandwidth NVLink-C2C interconnect (900 GB/s bidirectional theoretical peak).

The measured 411 GB/s represents approximately 91% of the theoretical unidirectional peak (450 GB/s), indicating excellent hardware utilization. Remote access bandwidth of 88 GB/s is constrained by the inter-socket interconnect, which must route traffic through intermediate CPU-GPU modules.

Q2: Does SLURM's GRES scheduling preserve NUMA locality when assigning partial node resources?

Yes. Our shared queue experiments confirm that SDumont II's SLURM configuration correctly binds CPU resources to their NUMA-local GPU. Jobs requesting 1 GPU in shared mode achieved 413-414 GB/s H2D bandwidth—statistically indistinguishable from exclusive mode local access.

This is a significant finding for system administrators and users. It demonstrates that SDumont II's GRES plugin is configured with NUMA-aware GPU affinity, likely using SLURM's `--gres-flags=enforce-binding` or equivalent `gres.conf` settings. Users can trust the scheduler to make optimal placement decisions.

Q3: Do these bandwidth differences affect real application performance?

For single-GPU GROMACS runs, the answer is clearly **yes in the hypothetical case of poor scheduling, but no in practice**. Because SLURM preserves NUMA locality, shared and exclusive queues achieve identical 43.45 ns/day performance.

For multi-GPU runs, the situation is more nuanced. The 2-GPU degradation (-2.5%) and 4-GPU speedup (+250%) are primarily driven by GROMACS domain decomposition overhead rather than CPU-GPU bandwidth—all GPUs in exclusive mode use local access. However, if the system were misconfigured to allow remote GPU access, multi-GPU performance would suffer severely as inter-GPU communication often involves CPU staging.

0.5.2 5.2 Implications for HPC System Design

NUMA-Aware Scheduling is Essential

The 4.6× bandwidth penalty for remote access demonstrates that NUMA-aware GPU scheduling is not optional for modern heterogeneous HPC systems. Systems that allow arbitrary CPU-GPU assignments would leave substantial performance on the table for bandwidth-bound workloads.

SDumont II's configuration serves as a model: by enforcing CPU-GPU affinity in the SLURM GRES plugin, the system provides users with optimal performance transparently, without requiring explicit `numactl` pinning in job scripts.

Shared Queues Can Match Exclusive Performance

A common misconception is that shared GPU queues inherently sacrifice performance. Our results refute this for properly configured systems. When SLURM correctly assigns CPU resources with GPU affinity, there is no performance penalty for using shared mode.

This has important implications for cluster utilization. If users request exclusive nodes for single-GPU jobs (perhaps due to performance concerns), they waste 75% of the node's GPU resources. Promoting shared queue adoption for appropriate workloads can significantly improve overall cluster throughput.

0.5.3 5.3 GROMACS Scaling Considerations

The 2-GPU Anti-Scaling Phenomenon

The slight performance degradation observed with 2 GPUs (42.37 ns/day vs 43.45 ns/day for 1 GPU) deserves careful consideration. This is not a system configuration issue—it's an inherent characteristic of GROMACS parallelization for this workload size.

With 2 MPI ranks, GROMACS typically assigns: - Rank 0: Particle-Particle (PP) force computation on GPU 0 - Rank 1: Particle-Mesh Ewald (PME) electrostatics on GPU 1

This PP-PME separation creates a synchronization point every timestep where PP and PME forces must be combined. For the STMV system (~1M atoms), the communication and synchronization overhead exceeds the computational benefit of parallelization.

Recommendations for GROMACS Users on SDumont II:

1. **Small systems (<1M atoms):** Use 1 GPU. Multi-GPU will likely slow down the simulation.

2. **Medium systems (1-2M atoms):** Test 1 GPU vs 4 GPU. Avoid 2 GPU as it's likely to underperform. The 4-GPU configuration may or may not provide benefit depending on the specific system.

3. **Large systems (>2M atoms):** Use 4 GPUs for maximum throughput. Consider testing 2 GPUs as it may become viable at larger scales.

4. **Queue selection:** Always use the shared queue for 1-2 GPU jobs. There is no performance advantage to exclusive allocation, and shared mode improves overall cluster utilization.

0.5.4 5.4 Limitations and Future Work

Limitations of This Study:

- **Single benchmark system:** We used only STMV for GROMACS testing. Larger systems might show different scaling characteristics.
- **Single application:** Other GPU-accelerated codes may exhibit different sensitivity to CPU-GPU bandwidth.
- **Limited concurrent testing:** We did not measure performance under heavy node sharing with competing workloads.

Potential Future Work:

1. **Contention analysis:** Measure bandwidth degradation when multiple jobs share a node and compete for inter-socket bandwidth.

2. **Larger MD systems:** Characterize the atom count threshold where 2-GPU scaling becomes positive.

3. **Other applications:** Extend the analysis to other common HPC codes (NAMD, LAMMPS, VASP, TensorFlow) to provide broader guidance.

4. **NVLink GPU-GPU bandwidth:** Measure inter-GPU transfer performance, which is relevant for applications using NCCL or multi-GPU CUDA.

0.6 6. Conclusions

This study characterized CPU-GPU bandwidth and application performance on SDumont II's NVIDIA GH200 nodes, providing empirical answers to practical questions about GPU resource scheduling.

0.6.1 Key Findings

1. NUMA Topology Creates Significant Bandwidth Asymmetry

Local CPU-GPU access achieves 411.6 GB/s H2D bandwidth via the NVLink-C2C interconnect, while remote access is limited to 88.6 GB/s—a **4.65× difference**. This architectural characteristic makes NUMA-aware scheduling critical for performance.

2. SLURM GRES Scheduling Preserves Optimal Affinity

SDumont II's shared queue (`gh200_shared`) correctly binds CPU resources to their NUMA-local GPU. Shared mode bandwidth matches exclusive mode local bandwidth, confirming proper SLURM GRES configuration.

3. Shared and Exclusive Queues Provide Equivalent Single-GPU Performance

GROMACS STMV achieved **identical 43.45 ns/day** performance in both queue types. There is no performance justification for requesting exclusive nodes for single-GPU workloads.

4. Multi-GPU GROMACS Scaling Depends on System Size

- 2 GPUs: **Slight slowdown** (-2.5%) due to PP-PME communication overhead
- 4 GPUs: **2.5× speedup** (62% parallel efficiency)

Users should choose 1 GPU or 4 GPUs for STMV-sized systems; 2 GPUs is suboptimal.

0.6.2 Practical Recommendations

Workload	Recommended Queue	Recommended GPUs
Single-GPU jobs	Shared (`gh200_shared`)	1
Multi-GPU GROMACS (<2M atoms)	Exclusive (`gh200`)	1 or 4
Multi-GPU GROMACS (>2M atoms)	Exclusive (`gh200`)	2 or 4
GPU scaling tests	Exclusive (`gh200`)	Variable

0.6.3 Final Remarks

SDumont II's GH200 partition demonstrates that modern HPC systems can efficiently share GPU resources without performance sacrifice—provided the scheduler is correctly configured for NUMA-aware placement. Users should confidently adopt the shared queue for appropriate workloads, contributing to better overall cluster utilization while maintaining optimal application performance.

0.7 7. References

1. NVIDIA GH200 Grace Hopper Superchip Documentation - https://www.nvidia.com/en-us/data-center/grace-hopper-superchip/
2. SLURM Generic Resource (GRES) Scheduling - https://slurm.schedmd.com/gres.html
3. SDumont II User Manual - https://github.com/lncc-sered/manual-sdumont2nd
4. NVIDIA nvbandwidth Tool - https://github.com/NVIDIA/nvbandwidth
5. NVIDIA GH200 Benchmark Guide - https://docs.nvidia.com/gh200-superchip-benchmark-guide.pdf
6. GROMACS Heterogeneous Parallelization Benchmark - https://zenodo.org/record/3893789
7. Abraham, M.J., et al. "GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers." SoftwareX 1-2 (2015): 19-25.

0.8 8. Appendix: Experimental Scripts and Data

0.8.1 8.1 Bandwidth Benchmark Scripts

Exclusive queue script (`scripts/transfer/exclusive.slurm`): - Runs nvbandwidth 4 times, once pinned to each NUMA domain - Uses `numactl --cpunodebind=N --membind=N` for strict affinity - Outputs 4×4 bandwidth matrix (NUMA × GPU)

Shared queue script (`scripts/transfer/shared_array_1gpu.slurm`): - SLURM job array requesting 1 GPU per task - Records SLURM-assigned CPU binding and GPU PCI ID - Measures bandwidth from assigned CPUs to assigned GPU

0.8.2 8.2 GROMACS Benchmark Scripts

Exclusive queue (`scripts/gromacs/gromacs_exclusive.slurm`): - Tests 1, 2, and 4 GPU configurations sequentially - Uses GROMACS 2023.2 NVIDIA container - Full GPU offload (nb, bonded, pme)

Shared queue (`scripts/gromacs/gromacs_shared_1gpu.slurm`, `gromacs_shared_2gpu.slurm`): - 1-GPU and 2-GPU configurations - Same simulation parameters as exclusive

0.8.3 8.3 Data Files

Directory	Contents
`data/transfer/`	Raw nvbandwidth output files
`data/gromacs/`	GROMACS performance logs
`data/gromacs/logs/`	SLURM job output files

0.8.4 8.4 Reproducibility

To reproduce these experiments:

```
# Bandwidth benchmarks
cd scripts/transfer
sbatch exclusive.slurm
sbatch shared_array_1gpu.slurm

# GROMACS benchmarks
cd scripts/gromacs
sbatch gromacs_exclusive.slurm
sbatch gromacs_shared_1gpu.slurm
sbatch gromacs_shared_2gpu.slurm

# Generate report
cd reports/paper
jupyter notebook report.ipynb
```