13th International Workshop on OpenCL and SYCL

IWOCL 2025

Achieving High-throughput Strided Data Movement Across GPUs

Peter Thoman, UIBK

Peter Thoman & Philipp Gschwandtner
Parallel and Distributed Systems Group
University of Innsbruck, Austria

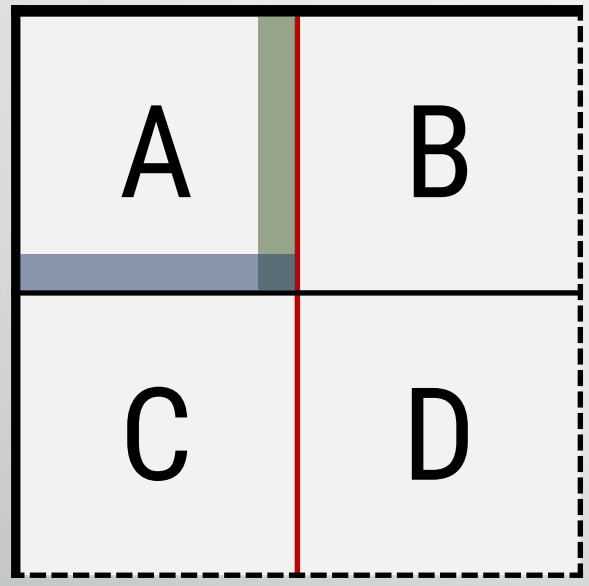
April 7-11, 2025 | Heidelberg, Germany | iwocl.org

KHRONOS GROUP

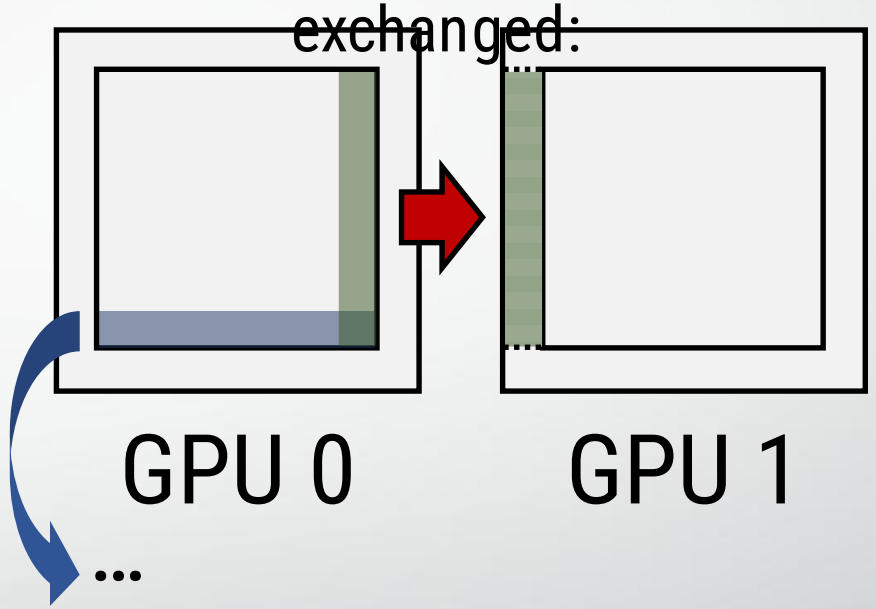
Background, Motivation & Goals

Background

Imagine a conventional 2D domain decomposition, with work split across GPUs:



After a compute step, data has to be exchanged:



→ **Row exchanges** are unproblematic

→ **Column exchanges** require strided transfer

→ See also: **Moritz'** keynote (does manual linearized staging)

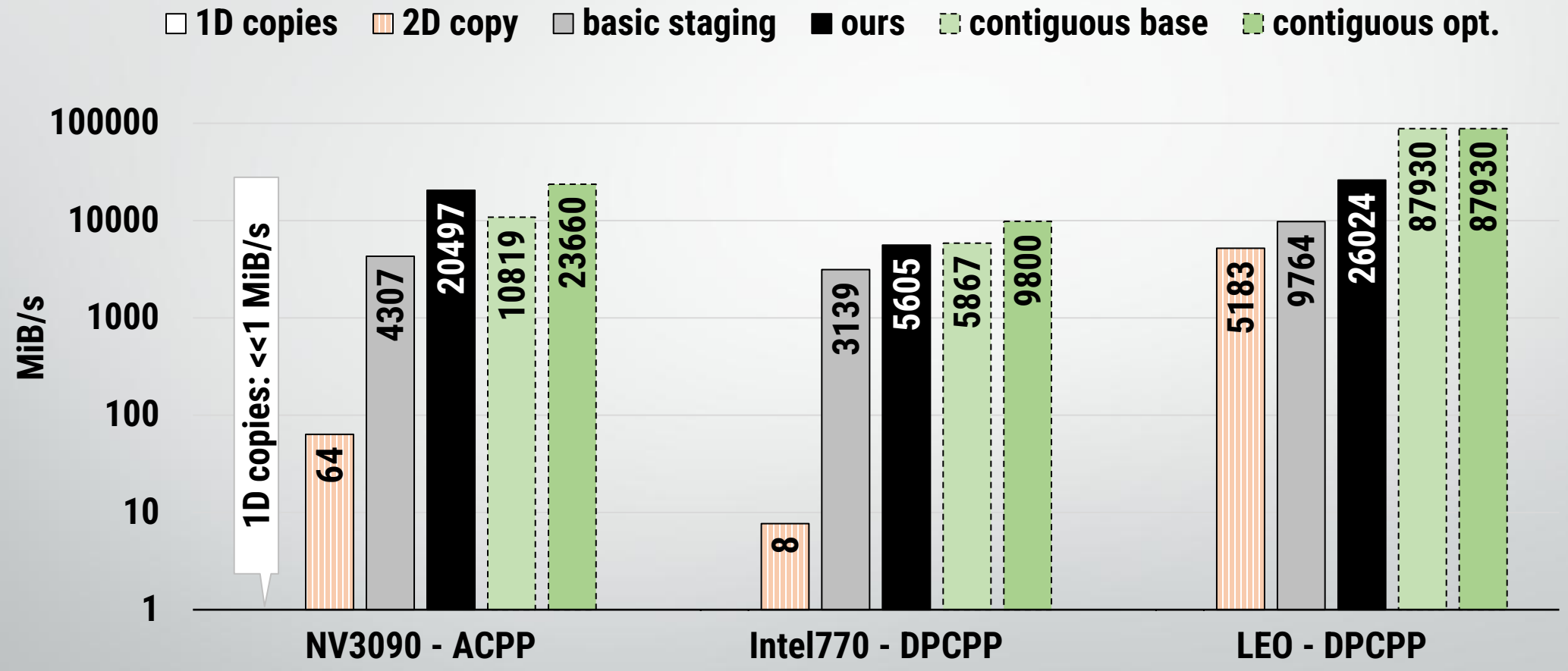
→ Even on FPGA's, **Christoph** noted that north/south splits more effective

Motivation

- **Celerity** is a runtime system which implements a SYCL-like API on distributed memory clusters
  - ➔ Uses various SYCL implementations as back-ends (which might in turn support multiple target platforms)

- Distributes computation across
  1. **GPUs in a node** → Needs fast **device** ⇔ **device** transfers
  2. **Nodes in a cluster** → Needs fast **device** ⇔ **host** transfers

Motivation – Copy Performance



* Transferring column of 1 32 bit float value, 256 byte stride; “ours” uses tuned copy plan and settings for each target platform

Observations & Goals

Observations

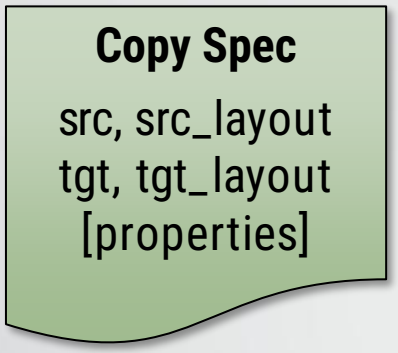
- The baseline option available in standard SYCL (repeated 1D copies) is **unusable**
- “Native” 2D copy implementations are much better, but still **unusable** on consumer hardware
- Expending the effort to implement manual staging yields *decent* results, but **far from optimal**
- It is possible to do **far** better for this use case on **all** hardware, but it introduces a **lot of complexity**

Goals

1) Build a flexible **language and framework** for constructing optimized **copy plans**
2) **Investigate** the performance **of various strategies** across hardware and copy scenarios in detail
3) Provide a **user-friendly library** which allows **for high-performance strided data movement** with a simple interface

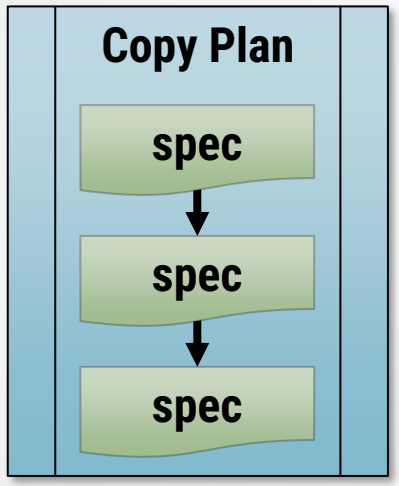
A Framework for Data Movement

Basic Elements

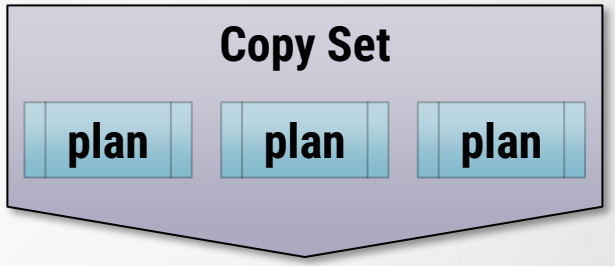


Describes either

- an **input** to the system, *or*
- a **single fundamental copy** operation within it



A **sequence** of copy operations which depend on their predecessor



A **set** of **independent** copy plans which may be executed in **parallel**

Customization Points

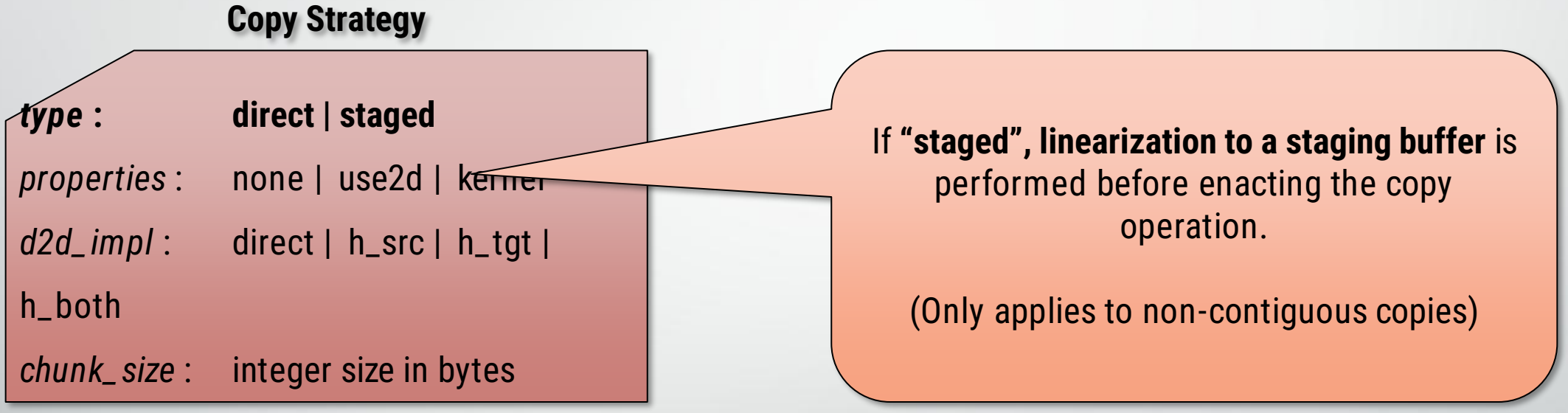
- Implementation choices are determined by a **copy strategy**

Copy Strategy

type :	direct | staged
properties :	none | use2d | kernel
d2d_impl :	direct | h_src | h_tgt | h_both
chunk_size :	integer size in bytes

Customization Points

- Implementation choices are determined by a **copy strategy**



Customization Points

- Implementation choices are determined by a **copy strategy**

Copy Strategy

type :	direct | staged
***properties* :**	**none | use2d | kernel**
d2d_impl :	direct | h_src | h_tgt | h_both
chunk_size :	integer size in bytes

Determines how individual steps are enacted.

- **use2d** enables use of native 2d transfers
- **kernel** enables use of (de-)linearization kernel

Both only have an effect when the HW/SW stack has this capability.

Customization Points

- Implementation choices are determined by a **copy strategy**

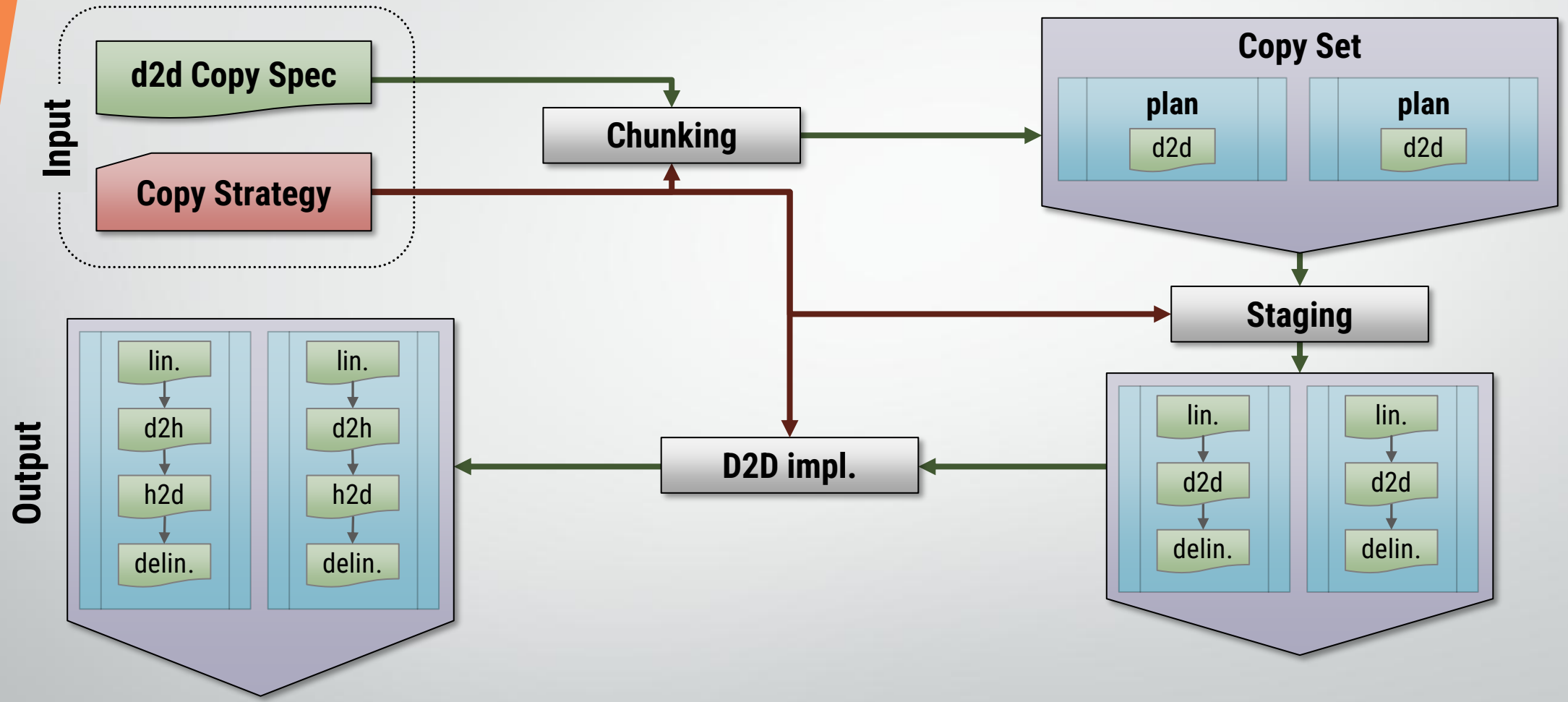
Copy Strategy

type :	direct | staged
properties :	none | use2d | kernel
***d2d_impl* :**	**direct | h_src | h_tgt | h_both**
chunk_size :	integer size in bytes

For device to device (d2d) copies, determines if manual staging through host memory should be enabled.

- **direct**: no staging
- **h_src**: stage in host mem “close” to src
- **h_tgt**: stage in host mem “close” to tgt
- **h_both**: stage in host mem twice

Manifestation Pipeline



Implementation Notes

Focusing on some highlights

General

- All operations can be performed and are unit **tested** individually
  - Optional validity and equivalence checks at runtime, enabled by default on debug builds
- Implementation of **asynchronicity** is much more effective with a small number of manually-managed in-order queues
  - Submission in a thread pool with one thread per queue
- On NUMA systems, accurate **first-touch memory placement** of host staging memory matters
- Three supported backends: **SimSYCL** for development/CI, and tuned implementations for **DPCPP** and **AdaptiveCPP** (ACPP)

DPCPP Backend Implementation / Tuning

- Three extensions used:
  - SYCL_EXT_ONEAPI_PEER_ACCESS : allows querying and enabling peer access without requiring vendor-specific API
  - SYCL_EXT_ONEAPI_MEMCPY2D : provides access to 2d copy operations, performance equivalent to direct “native” vendor APIs in our testing
  - SYCL_EXT_INTEL_QUEUE_IMMEDIATE_COMMAND_LIST : resulted in ~10% performance gain in some very fine-grained microbenchmarks

→ Overall, having peer access / memcpy2d available as vendor-independent abstractions is valuable, and appears to be “free”

ACPP Backend Implementation / Tuning

- AdaptiveCpp_enqueue_custom_operation was used to access native 2d copies (of e.g. CUDA)
- **ACPP_ALLOW_INSTANT_SUBMISSION** is *essential* for achieving peak performance in some of our benchmark workloads
- **ACPP_EXT_COARSE_GRAINED_EVENTS** enables a minor performance boost
- Interestingly, for ACPP manual thread placement had a very significant performance impact, up to 1.5x in our testing

→ In total, the tuning impact is high, up to 3.7x in the most affected microbenchmark configuration

Linearization Kernel Optimization

- **Multi-versioning** (template cascade) for
  - Various fixed small data types
  - 32- or 64-bit indexing
  - Matching or diverging fragment counts
- Per-vendor **workgroup size tuning**

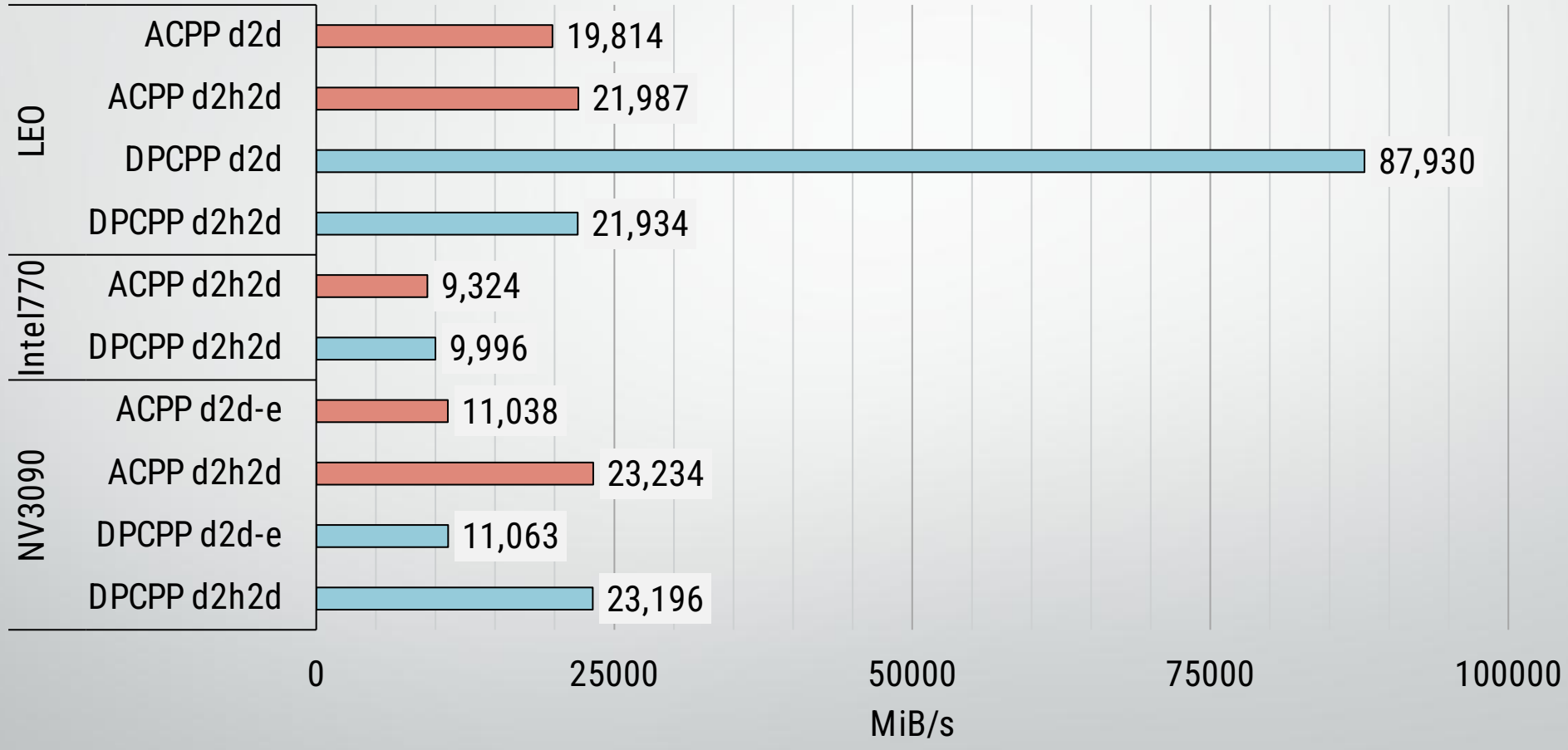
Evaluation

Experiment Setup

	NV3090	Intel770	LEO
CPU	2x AMD EPYC 7763	2x Intel Xeon Gold 5317	1x Xeon Platinum 8358
GPUs	4x NVIDIA RTX 3090	2x Intel Arc A770	4x NVIDIA A100-SXM-64GB
Interconnect	16x PCIe 4.0 (host)	8x PCIe 4.0 (host)	4x NVLink 4.0 (direct)
Runtime	CUDA 12.6	Level Zero 12.55.8	CUDA 12.1
Driver	560.35.03	1.3.29138	530.30.02
OS	Ubuntu 24.04.1 LTS	Ubuntu 24.04.1 LTS	Red Hat Ent. Linux 8.7

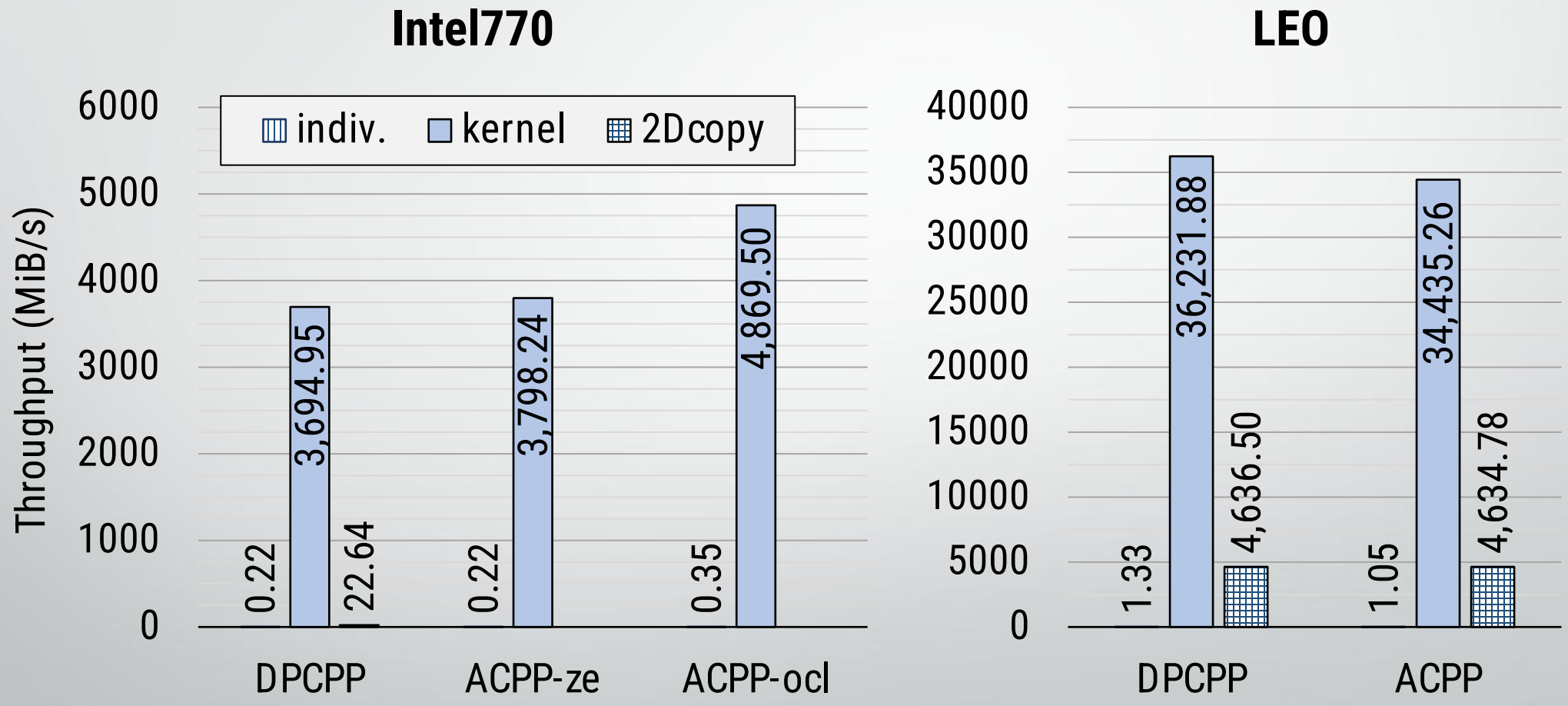
- SYCL implementations: AdaptiveCPP v24.10.0, Intel oneAPI 2025.0.4
- 50 runs per data point, 500 for microbenchmarks < 1ms

Contiguous Baseline



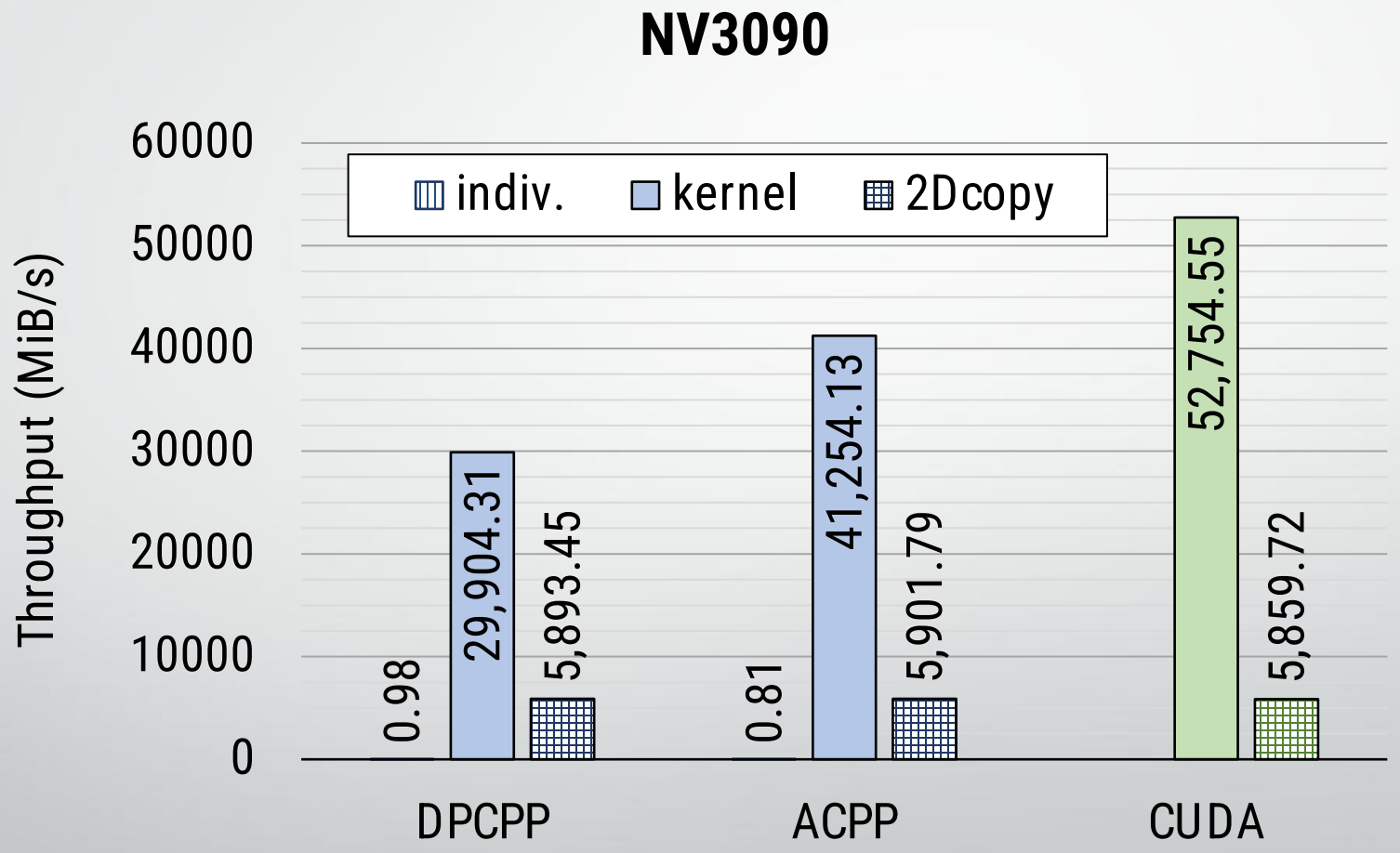
Transferring contiguous 64 MiB blocks

Linearization Performance



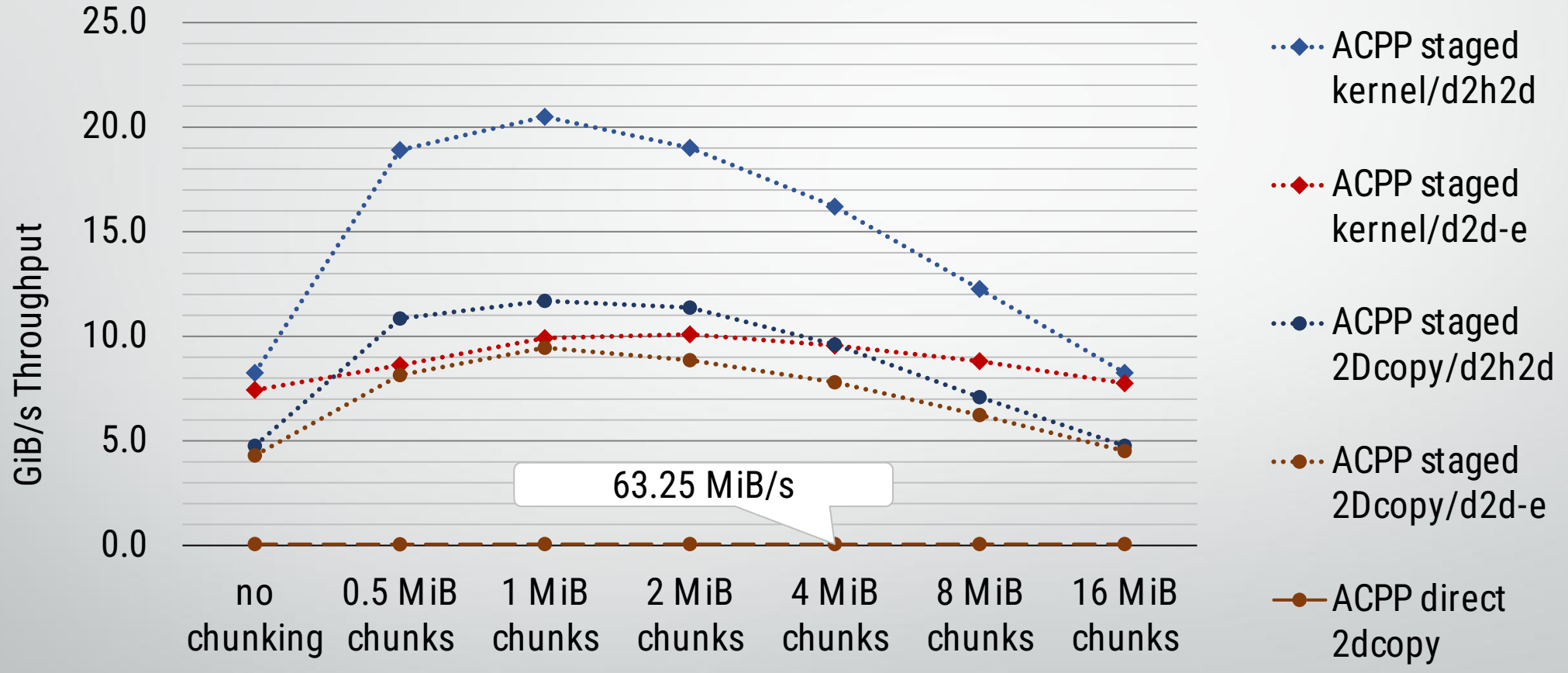
Linearizing a single column of 32 million 4-byte elements, 256 byte stride

Linearization Performance



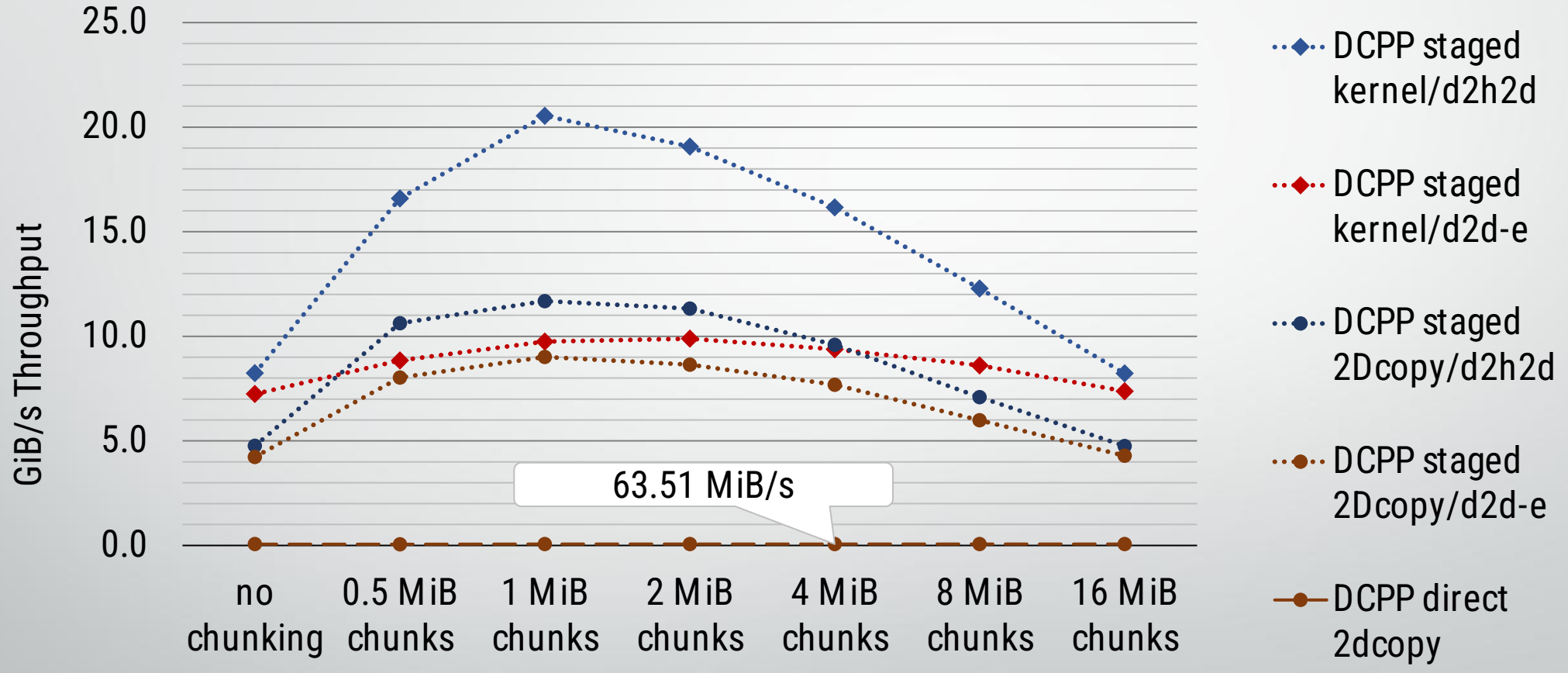
Linearizing a single column of 32 million 4-byte elements, 256 byte stride

Strided Device-to-device Copy (NV3090)



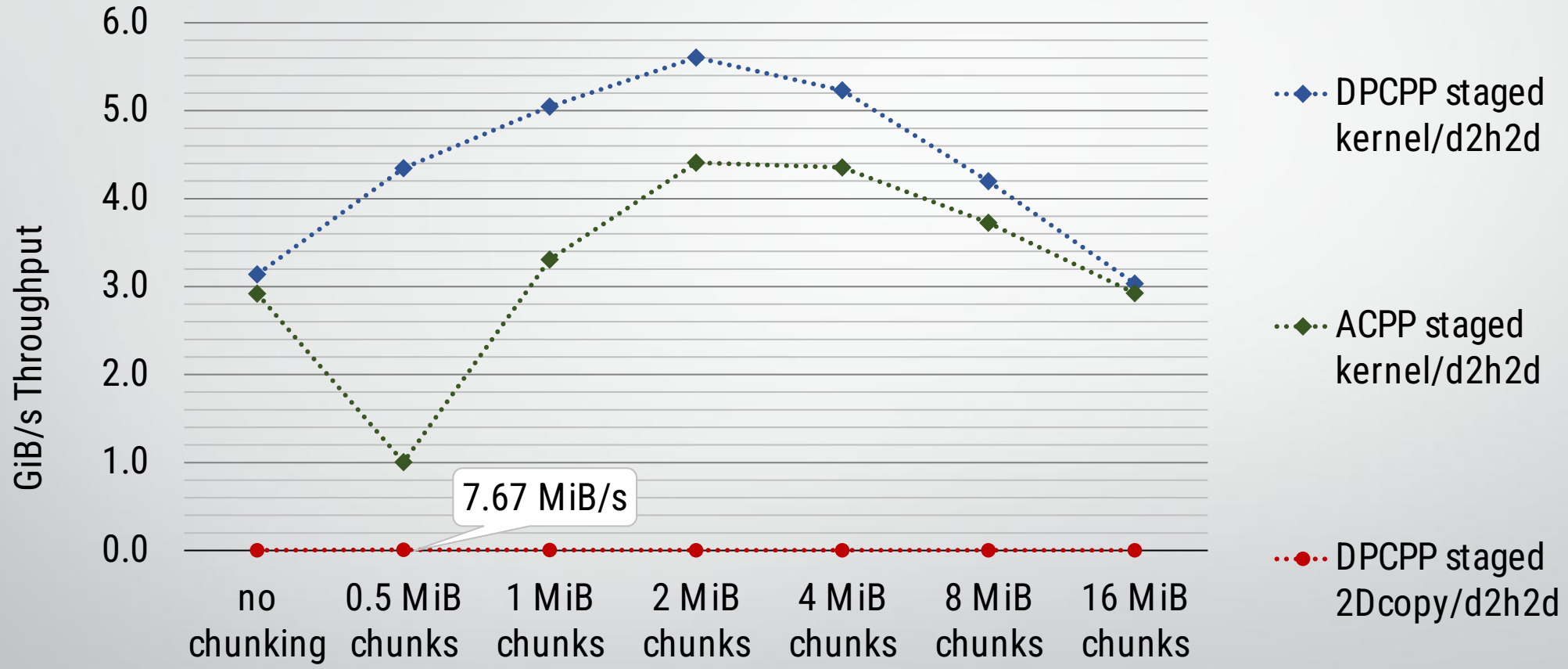
Transferring 16 MiB of strided 4-byte (float) data, 256 byte stride

Strided Device-to-device Copy (NV3090)



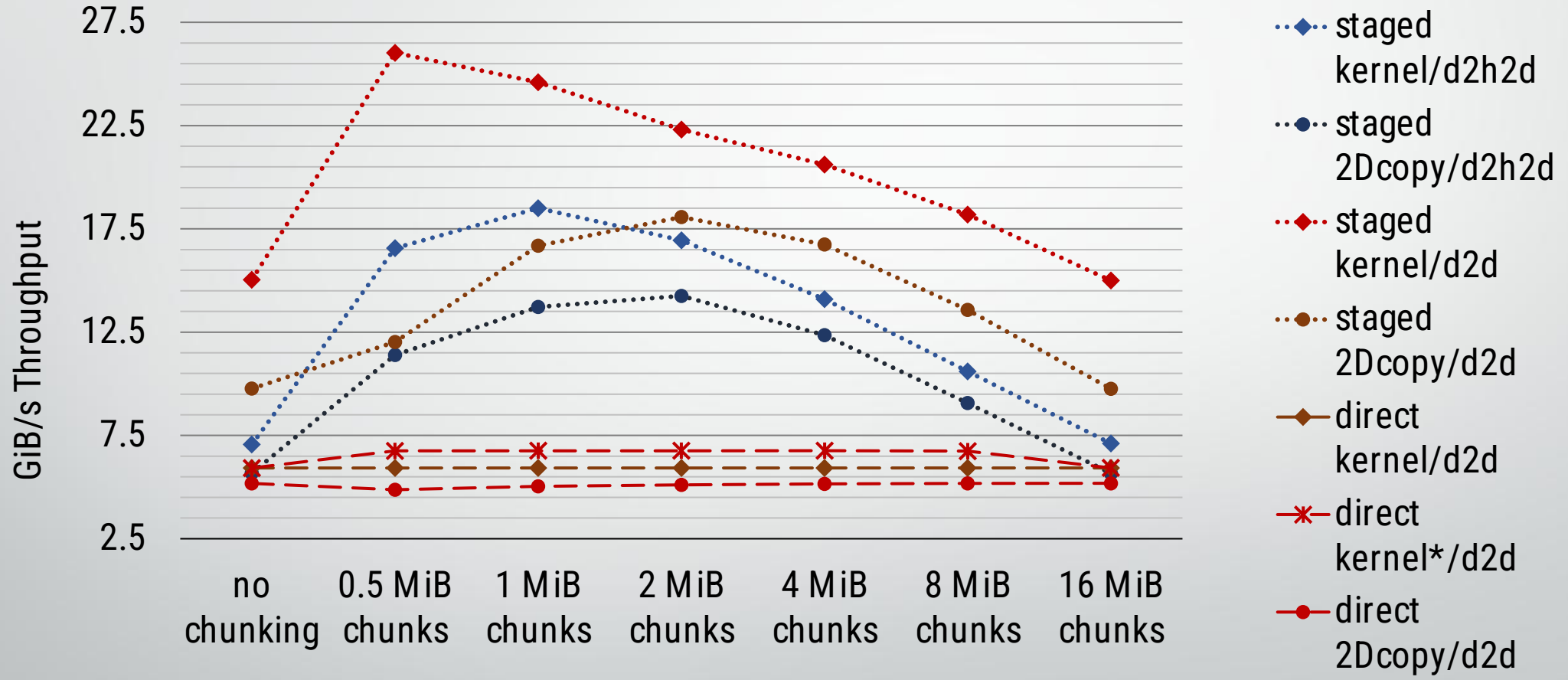
Transferring 16 MiB of strided 4-byte (float) data, 256 byte stride

Strided Device-to-device Copy (Intel770)



Transferring 16 MiB of strided 4-byte (float) data, 256 byte stride

Strided Device-to-device Copy (LEO)



kernel* : interleaved multi-device kernel execution policy

Closing Thoughts

Summary & Conclusion

- On consumer platforms, naïve transfer of strided data results in extremely poor performance
  - And even with dedicated interconnects, we can achieve a 5x performance gain
- Reaching high throughput in non-trivial, requires
  - Optimized parallel copy plans with appropriate staging
  - High-performance kernel-based (de-)linearization
  - Asynchronous, low-overhead enactment engine
    - With per-backend tuning

→ **gpu2dcopylib** does all that

Future Work & Limitations

- Impact of kernel-based linearization on potential concurrent computation overlap was not investigated
- Patterns of transfer involving more than 2 devices could benefit from specialized treatment (e.g. broadcast)
- Support for more complex (e.g. 3D) strides is required by some applications
- For end-user utility, deriving an optimal (or “good enough”) copy strategy automatically would be valuable

Thank you for your attention!

https://github.com/PeterTh/gpu2dcopylib

https://celerity.github.io

https://github.com/celerity/SimSYCL

This research is supported by the Austrian Research Promotion Agency (FFG)
via the UMUGUC project (FFG #4814683).
Compute time on Leonardo was provided by an AURELEO grant.

universität innsbruck

DPS
Distributed and Parallel Systems