



DISTRIBUTED GPU PROGRAMMING FOR EXASCALE

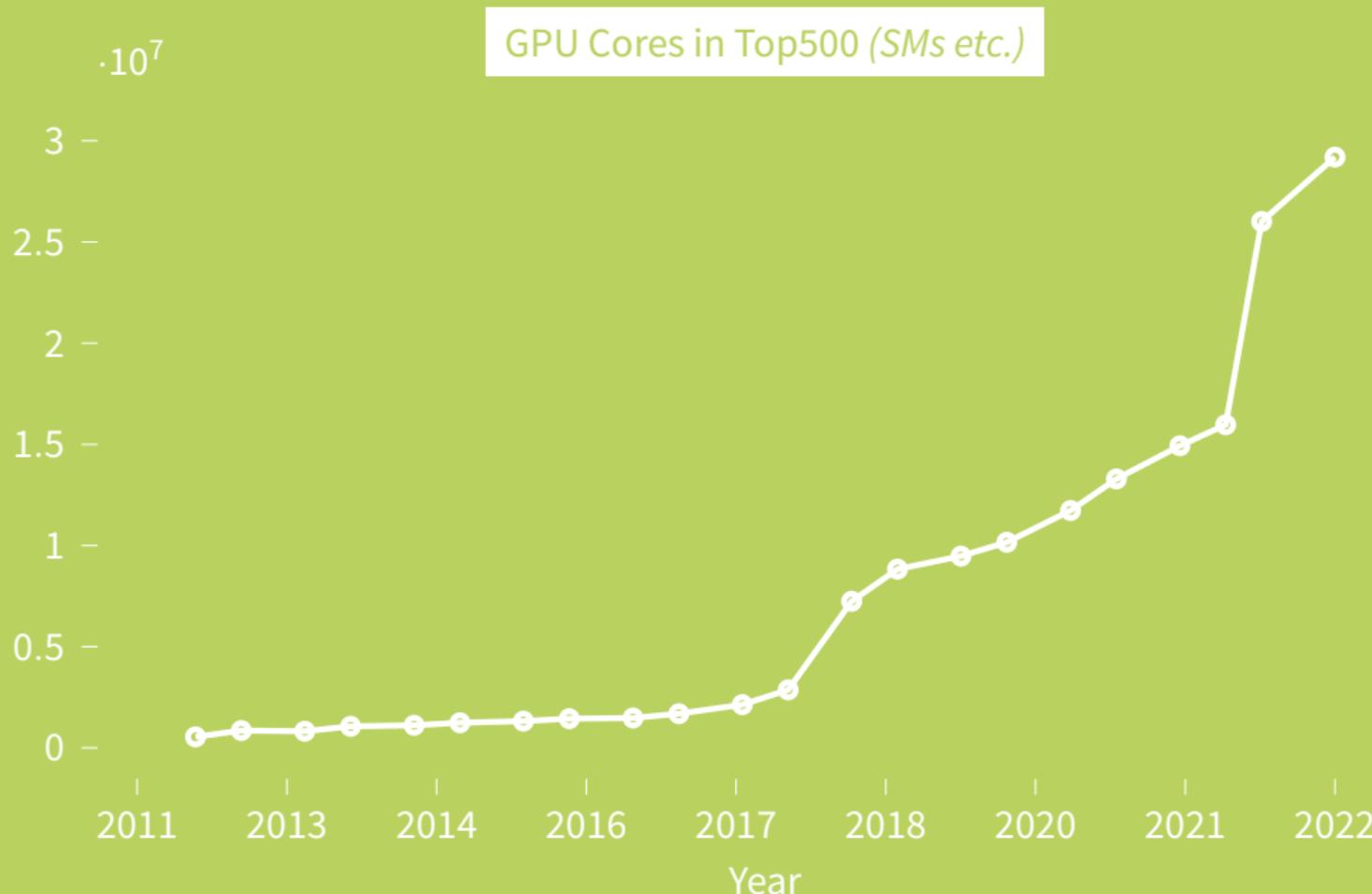
ISC23 TUTORIAL SESSION 1

21 May 2023 | Andreas Herten | Jülich Supercomputing Centre, Forschungszentrum Jülich

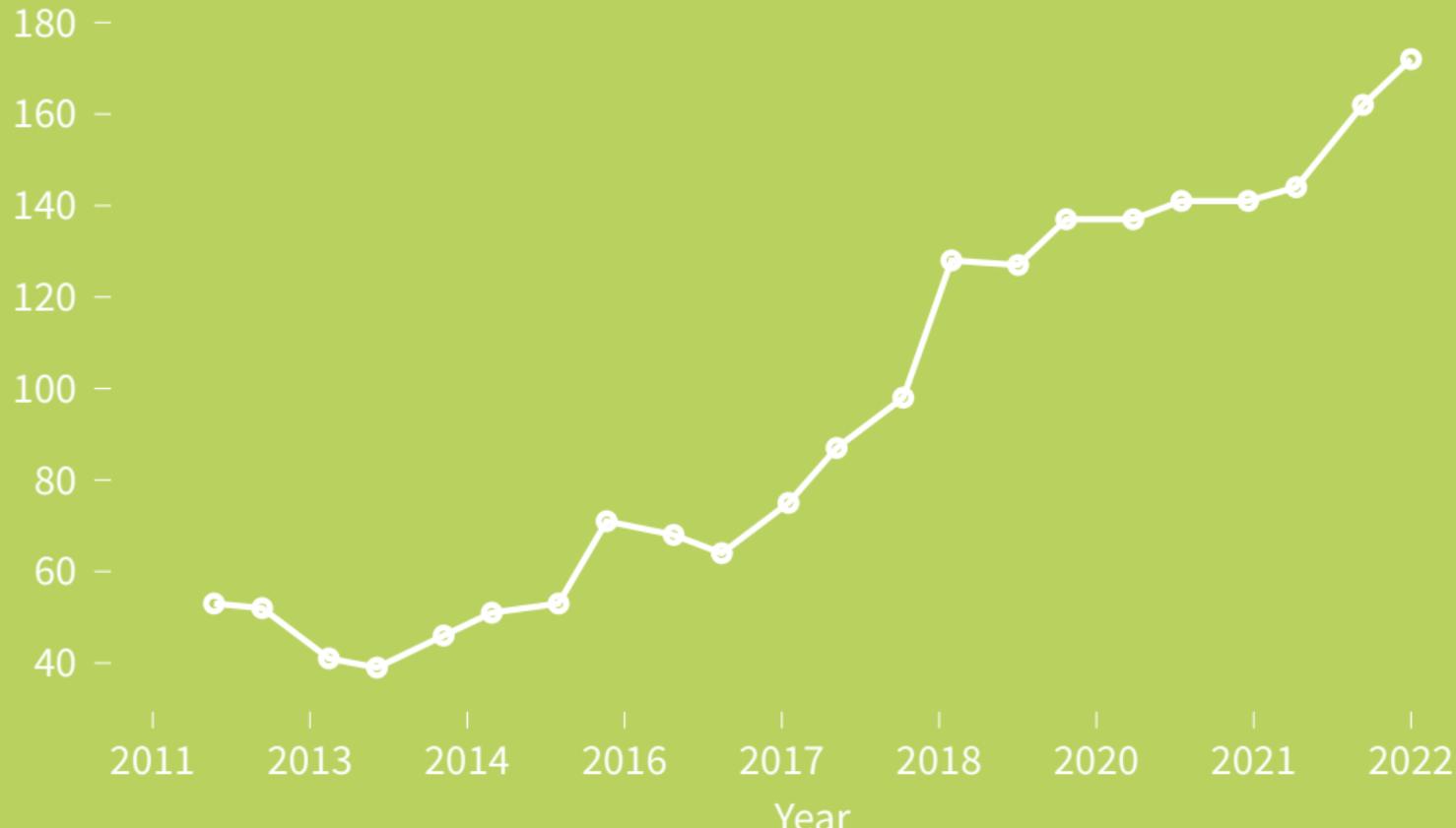
Welcome to

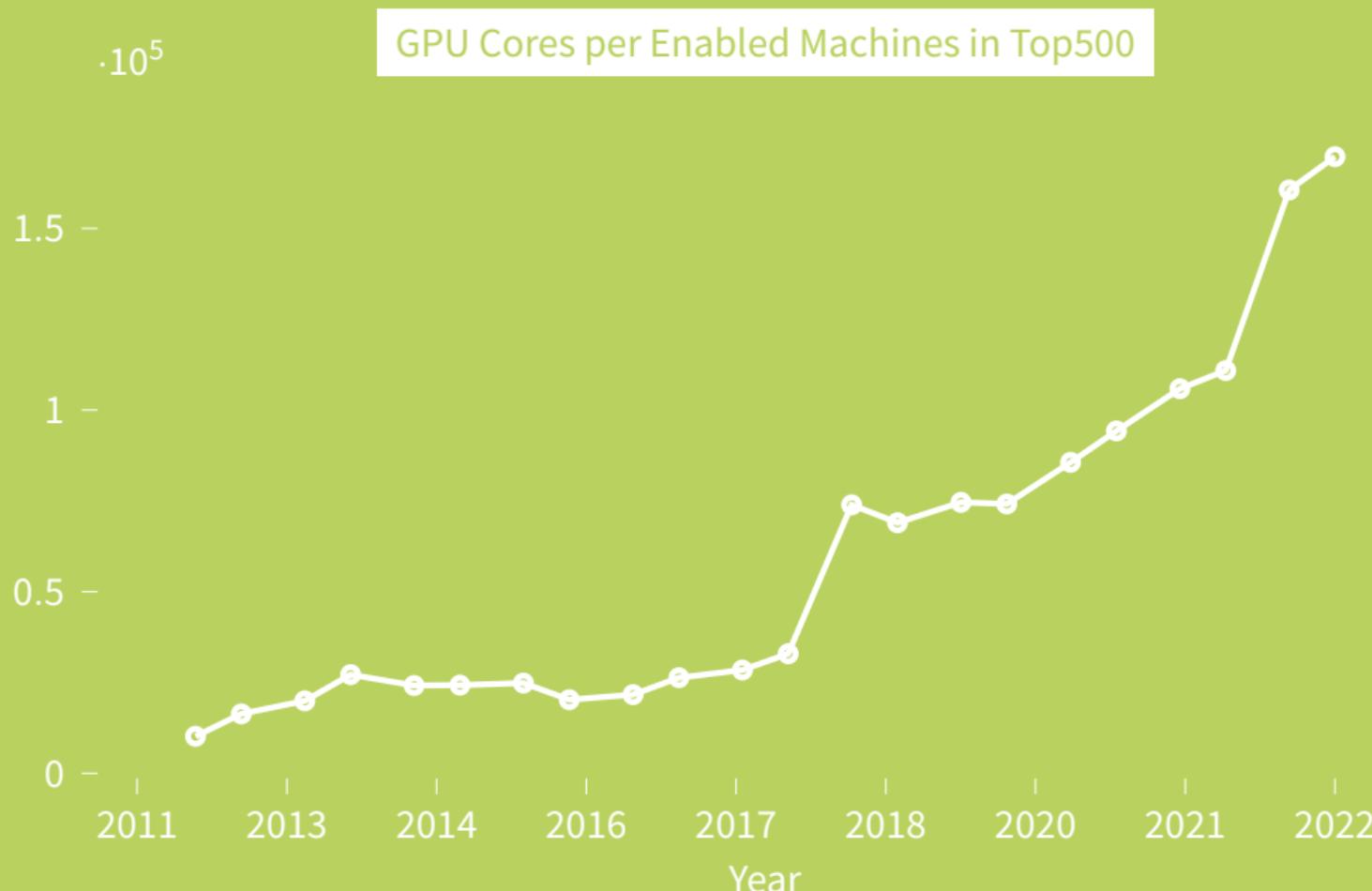
Efficient Distributed GPU Programming for Exascale,

an *ISC23 Tutorial*



GPU-enabled Machines in Top500





State of the GPUUnion

Landscape Overview

- Last 10 years
 - More and more GPUs installed in HPC machines
 - More and more HPC machines with GPUs
 - More and more GPUs in each machine

State of the GPUUnion

Landscape Overview

- Last 10 years
 - More and more GPUs installed in HPC machines
 - More and more HPC machines with GPUs
 - More and more GPUs in each machine
- Future
 - GPUs selected as technology for enabling Exascale
→ Even larger GPU machines with larger GPUs
 - Pre-Exascale systems: LUMI^A, Leonardo^N; JUWELS Booster^N, Perlmutter^N
 - Exascale systems: Frontier^A, El Capitan^A, Aurora^I; JUPITER[?]



ORNL's Frontier: > 35 000 GPUs, 1.5 EFLOP/s, 2022

32 % of all installed GPU cores in Frontier!
Rendering by ORNL

State of the GPUUnion

Landscape Overview

- Last 10 years
 - More and more GPUs installed in HPC machines
 - More and more HPC machines with GPUs
 - More and more GPUs in each machine
 - Future
 - GPUs
 - Even longer list of machines in longer list
- We help to Tame the Beasts!*
- Pre-Exascale systems: LUMI^A, Leonardo^N;
 - JUWELS Booster^N, Perlmutter^N
 - Exascale systems: Frontier^A, El Capitan^A,
 - Aurora^I; JUPITER[?]



ORNL's Frontier: > 35 000 GPUs, 1.5 EFLOP/s,
2022

32 % of all installed GPU cores in Frontier!
Rendering by ORNL

About Tutorial

Goals

- Prepare for large-scale GPU systems
- Learn GPU+MPI basics
- Apply optimization, analysis techniques
- Study CPU-less GPU+MPI
- Use advanced libraries to improve scaling efficiency

About Tutorial

Goals

- Prepare for large-scale GPU systems
- Learn GPU+MPI basics
- Apply optimization, analysis techniques
- Study CPU-less GPU+MPI
- Use advanced libraries to improve scaling efficiency
- Learn interactively!

About Tutorial

Goals

- Prepare for large-scale GPU systems
- Learn GPU+MPI basics
- Apply optimization, analysis techniques
- Study CPU-less GPU+MPI
- Use advanced libraries to improve scaling efficiency
- Learn interactively!

Non-Goals

- Optimize your GPU application; we teach tools and techniques, you need to apply!
- Learn MPI; we expect base-level knowledge of MPI, you don't need more.
- Discuss general scalability; GPU-independent features (like load balancing) are too broad a topic
- Learn CUDA; we expect principle knowledge of GPU programming
- Showcase all GPU platforms; we use an NVIDIA system and teach NVIDIA libraries and tools, other platforms (AMD) follow along (see last lecture)

About Tutorial

Goals

- Prepare for large-scale GPU systems
- Learn GPU+MPI basics
- Apply optimization, analysis techniques
- Study CPU-less GPU+MPI
- Use advanced libraries to improve scaling efficiency
- Learn interactively!

Non-Goals

- Optimize your GPU application; we teach tools and techniques, you need to apply!
- Learn MPI; we expect base-level knowledge of MPI, you don't need more.
- Discuss general scalability; GPU-independent features (like load balancing) are too broad a topic
- Learn CUDA; we expect principle knowledge of GPU programming
- Showcase all GPU platforms; we use an NVIDIA system and teach NVIDIA libraries and tools, other platforms (AMD) follow along (see last lecture)

Curriculum

1L	Lecture <i>Onboarding</i>	Tutorial Overview, Introduction to System <i>Accessing JUWELS Booster</i>
2L	Lecture	Introduction to MPI-Distributed Computing with GPUs
3H	Hands-On	Multi-GPU Parallelization <i>11:00 - 11:30: Coffee Break</i>
4L	Lecture	Performance and Debugging Tools
5L	Lecture	Optimization Techniques for Multi-GPU Applications
6H	Hands-On	Overlap Communication and Computation with MPI <i>13:00 - 14:00: Lunch Break</i>
7L	Lecture	Overview of NCCL and NVSHMEM in MPI Programs
8H	Hands-On	Using NCCL and NVSHMEM <i>16:00 - 16:30: Coffee Break</i>
9L	Lecture	Device-initiated Communication with NVSHMEM
10H	Hands-On	Device-initiated Communication with NVSHMEM
11L	Lecture	Outline of Advanced Topics and Conclusion

Tutorial Team

Hello from Europe



Simon Garcia

Acc. and Comm. for HPC
Barcelona Supercomputing Center



Andreas Herten

Accelerating Devices Lab
Jülich Supercomputing Centre



Markus Hrywniak

DevTech Compute
NVIDIA



Jiri Kraus

DevTech Compute
NVIDIA



Lena Oden

Prof. for Computer Engineering
University Hagen

TAs

Jayesh Badwaik
Chelsea John
Elena Tikhonova

System: JUWELS Booster

JUWELS Overall Architecture

JUWELS Cluster (2018)

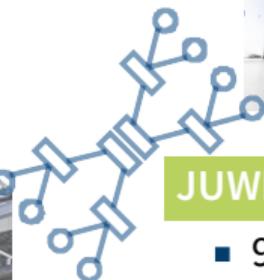
- 2511 compute nodes ($2 \times$ Skylake)
- 48 GPU nodes ($4 \times$ V100 w/ NVLink2)
- Mellanox EDR 100 Gbit/s network,
fat-tree topology (1:2@L1)
- 12 PFLOP/s



JUWELS Overall Architecture

JUWELS Cluster (2018)

- 2511 compute nodes ($2 \times$ Skylake)
- 48 GPU nodes ($4 \times$ V100 w/ NVLink2)
- Mellanox EDR 100 Gbit/s network,
fat-tree topology (1:2@L1)
- 12 PFLOP/s



JUWELS Booster (2020)

- 936 compute nodes ($2 \times$ AMD Rome,
 $4 \times$ A100 w/ NVLink3)
- Mellanox HDR 200 Gbit/s network,
DragonFly+ topology
- 73 PFLOP/s

JUWELS Overall Architecture

JUWELS Cluster (2018)

- 251
- 48
- Mell
- fat-
- 12



Top500 Nov-2020:

- #1 Europe
- #7 World
- #1* GreenTop500

21 May 2023

ke)
nk2)
rk,



JUWELS Booster (2020)

- 936 compute nodes (2× AMD Rome, 4× A100 w/ NVLink3)
- Mellanox HDR 200 Gbit/s network, DragonFly+ topology
- 73 PFLOP/s

JUWELS Booster Overview

Node Configuration

Arch Atos Bull Sequana XH2000

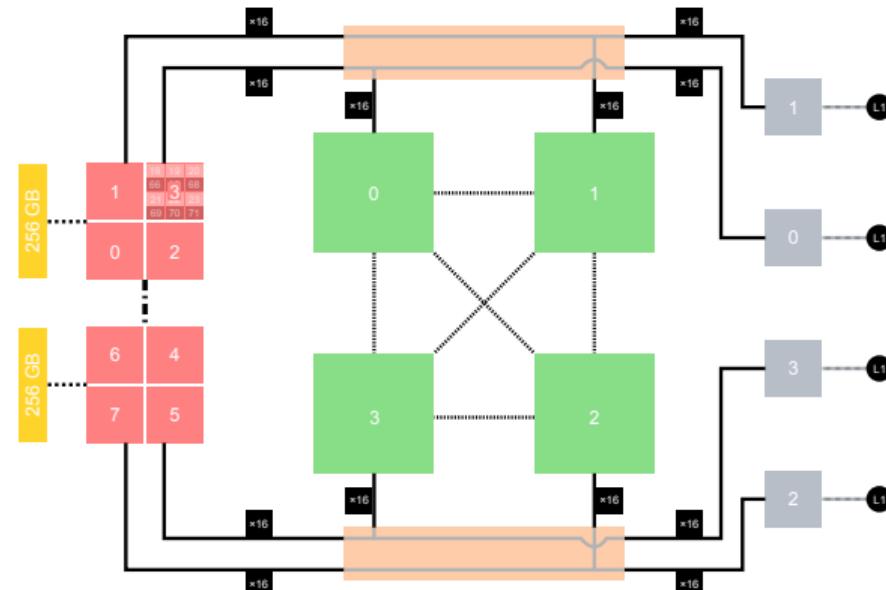
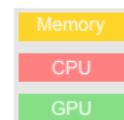
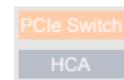
CPU 2 × AMD EPYC 7402:

2_{Socket} × 24_{Core} × 2_{SMT},
2 × 256 GB DDR4-3200 RAM;
NPS-4

GPU 4 × NVIDIA A100 40 GB, NVLink3

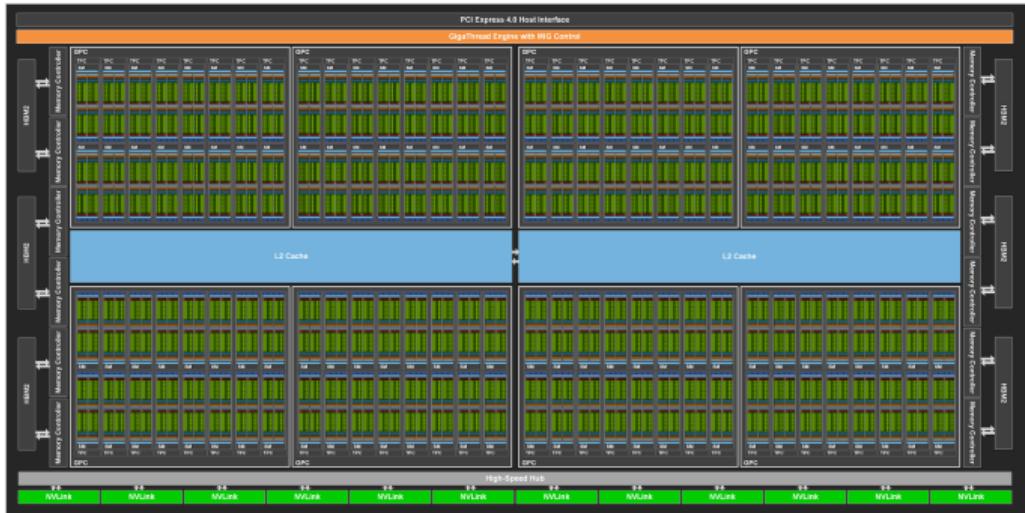
HCA 4 × Mellanox HDR200
(200 Gbit/s) InfiniBand
ConnectX 6

etc 2 × PCIe Gen 4 switch



NVIDIA A100 Tensor Core GPU

- JUWELS Booster:
3744 A100 GPUs
- A100: 108 SMs, large/fast L1+L2,
fast memory, NVLink3, PCIe4, ...



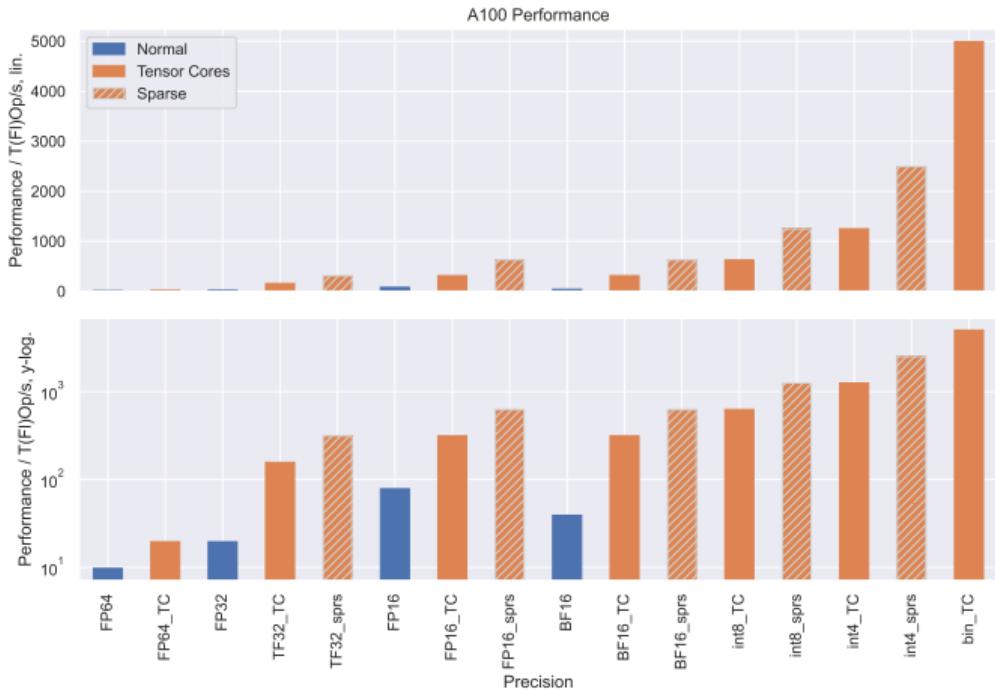
NVIDIA A100 Tensor Core GPU

- JUWELS Booster:
3744 A100 GPUs
- A100: 108 SMs, large/fast L1+L2,
fast memory, NVLink3, PCIe4, ...
- Tensor Cores



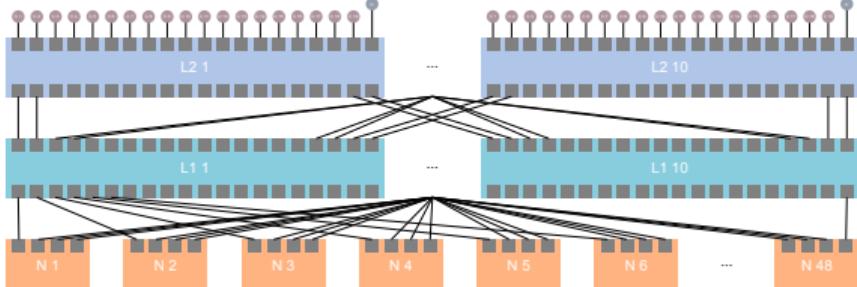
NVIDIA A100 Tensor Core GPU

- JUWELS Booster:
3744 A100 GPUs
- A100: 108 SMs, large/fast L1+L2,
fast memory, NVLink3, PCIe4, ...
- Tensor Cores, various
performances:
 $73 \text{ PFLOP}/S_{\text{FP64TC}}$ $584 \text{ PFLOP}/S_{\text{FP32TC}}$
 $1.16 \text{ EFLOP}/S_{\text{FP16TC}}$ $18.7 \text{ EOP}/S_{\text{BinTC}}$

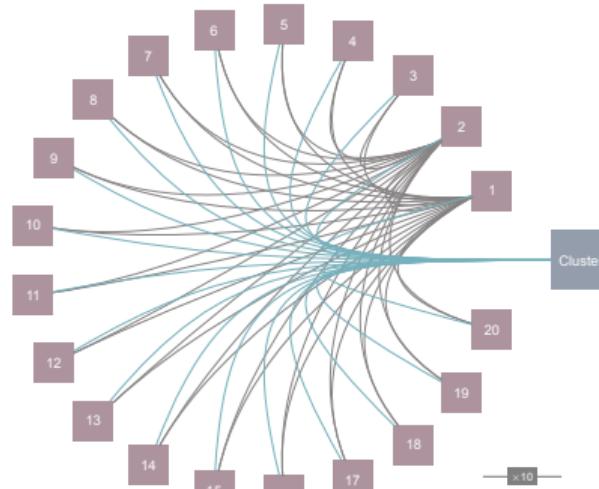


JUWELS Booster Overview

Network Configuration: DragonFly+ Network



In-Cell (48 nodes): Full fat-tree in 2 levels



Inter-Cell (20 cells): 10 links between each pair of cells

JUWELS

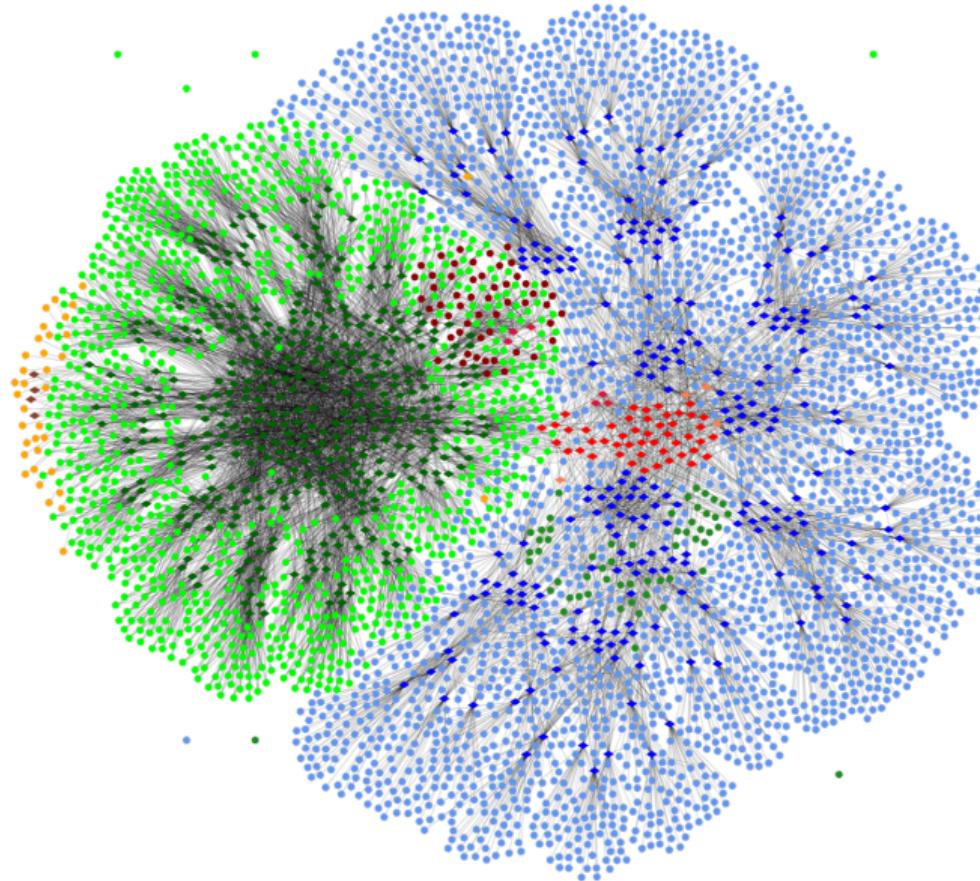
Cluster Booster Integration

Fully integrated system: **JUWELS** with Cluster and Booster modules

- File system: GPFS
- Network: InfiniBand
- Workload management: Slurm
- Resource management: ParaStation / ParaStation Slurm

Picture legend:

- | | |
|--------------------|-------------------|
| ■ Cluster CPU node | ■ Booster node |
| ■ Cluster GPU node | ■ Booster switch |
| ■ Cluster switch | ■ Booster gateway |
| ■ Cluster gateway | ■ JUST |
| ■ Top-level switch | ■ Service node |



AMD EPYC Rome

Some Processor Basics

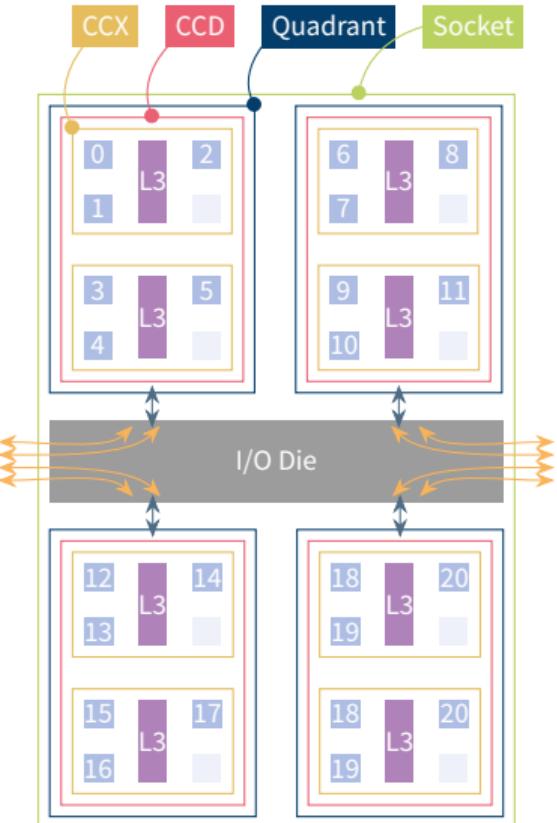
- JUWELS Booster: AMD EPYC Rome 7402 (24 core (SMT-2) × 2 sockets)
- EPYC processor: Built as Multi-Chip Module
→ Many building blocks, hierarchies

AMD EPYC Rome

Some Processor Basics

- JUWELS Booster: AMD EPYC Rome 7402 (24 core (SMT-2) \times 2 sockets)
- EPYC processor: Built as Multi-Chip Module
→ Many building blocks, hierarchies

3 cores	Core-Complex (CCX), shared L3 (<i>max 4 cores</i>)
2 CCXs	Core Complex Die (CCD)
1 CCD	1 quadrant (<i>max 2 CCDs per quadrant</i>)
4 quadrants	1 socket; NPS-4: 4 NUMA domains per socket
2 sockets	1 node (<i>only 1 shown</i>)
I/O Die	Connections between quadrants and outside
RAM	8 memory channels, 2 per quadrant



AMD EPYC Rome

Some Processor Basics

- JUWELS Booster: AMD EPYC Rome 7402 (24 core (SMT-2) \times 2 sockets)
- EPYC processor: Built as Multi-Chip Module
→ Many building blocks, hierarchies

3 cores Core-Complex (CCX), shared L3 (*max 4 cores*)

2 CCXs Core Complex Die (CCD)

1 CCD 1 quadrant (*max 2 CCDs per quadrant*)

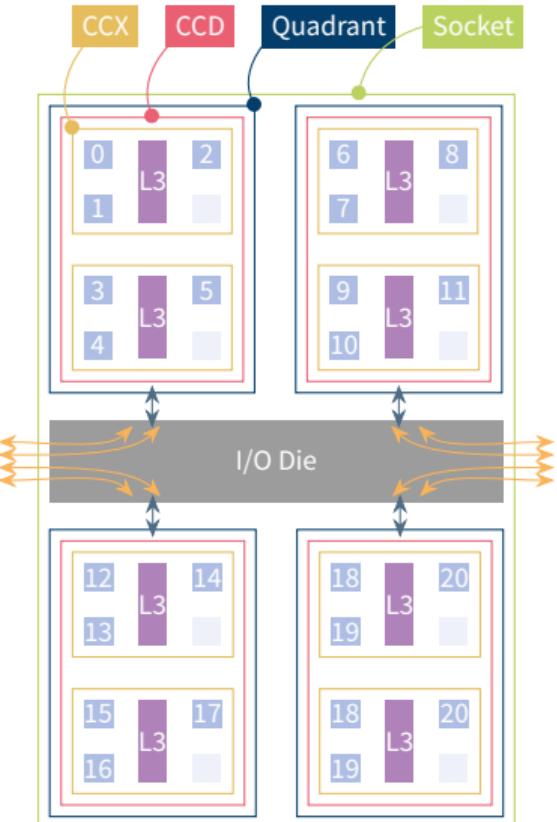
4 quadrants 1 socket; NPS-4: 4 NUMA domains per socket

2 sockets 1 node (*only 1 shown*)

I/O Die Connections between quadrants and outside

RAM 8 memory channels, 2 per quadrant

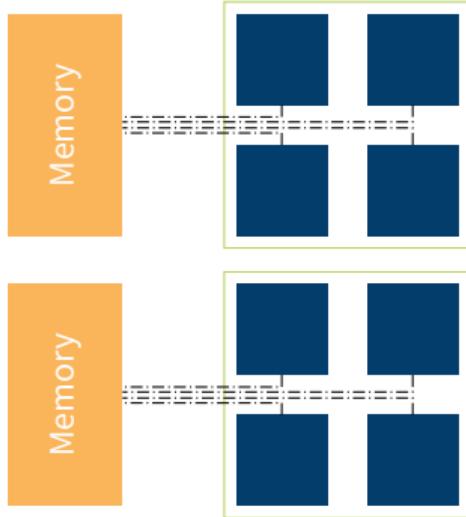
⇒ Complex topology!



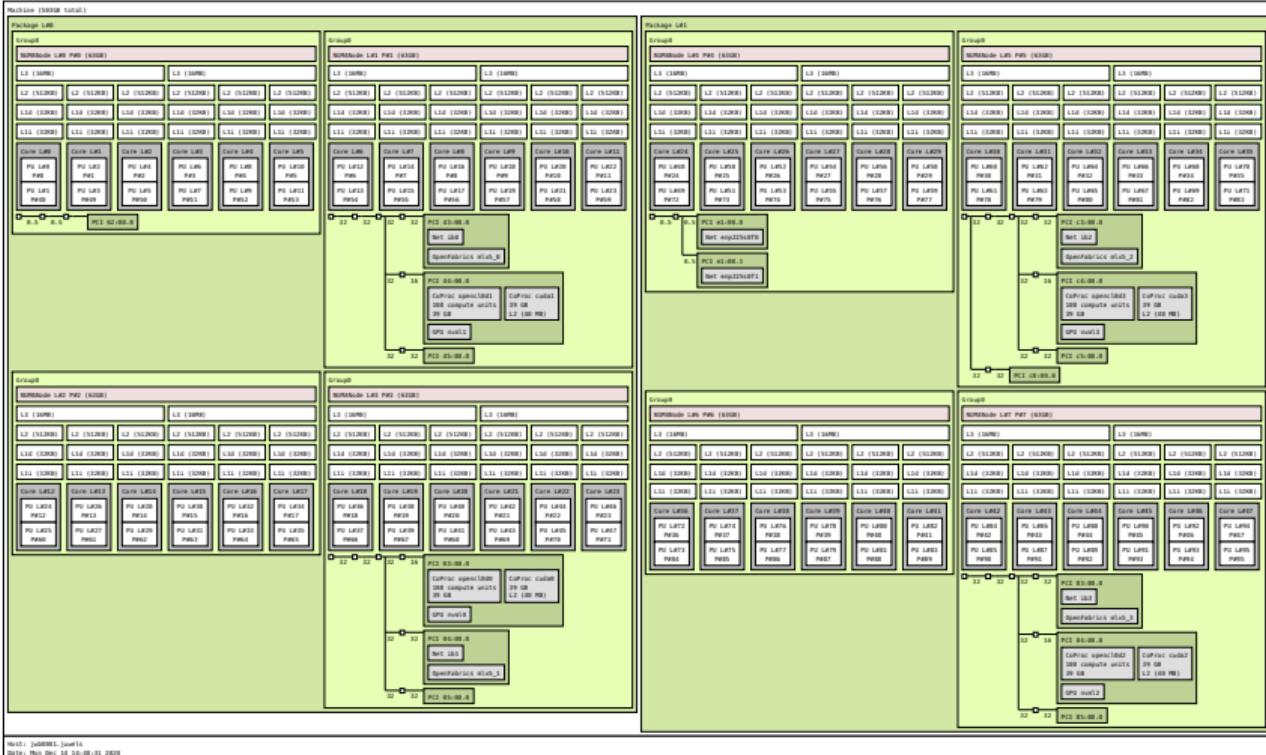
PCIe Affinity

3 C 2 CCX 1 CCD 4 Q 2 S

- PCIe via I/O die → also hierarchy
- GPUs, HCAs connected via PCIe (through PCIe switch)

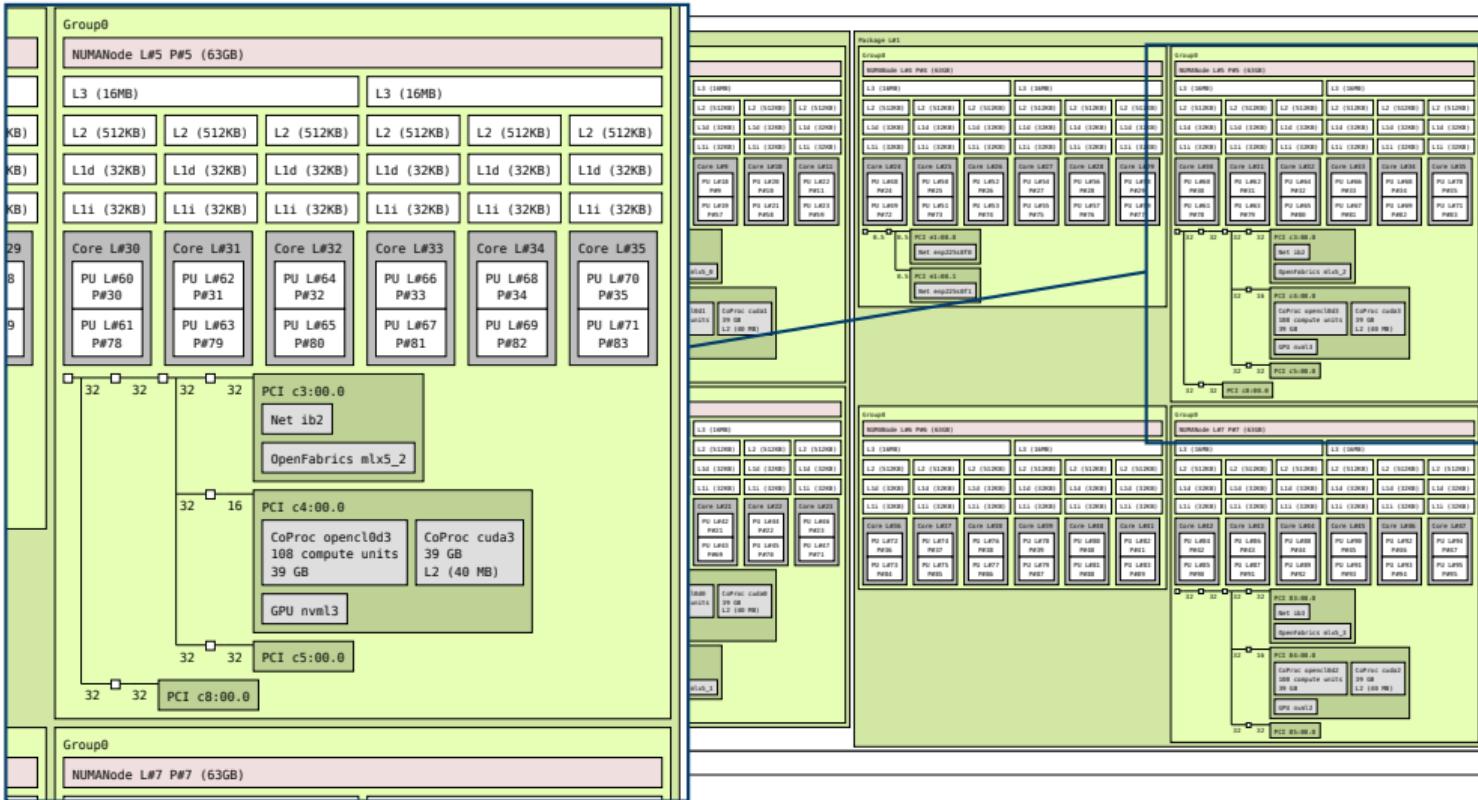


- PCIe via
 - GPUs, H



PCIe Affinity

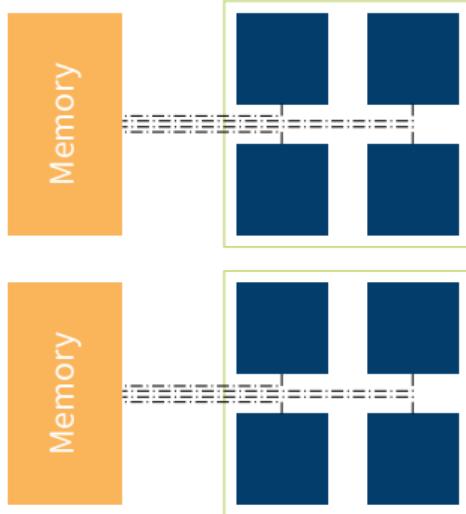
3C 2CCX 1CCD 4Q 2S



PCIe Affinity

3 C 2 CCX 1 CCD 4 Q 2 S

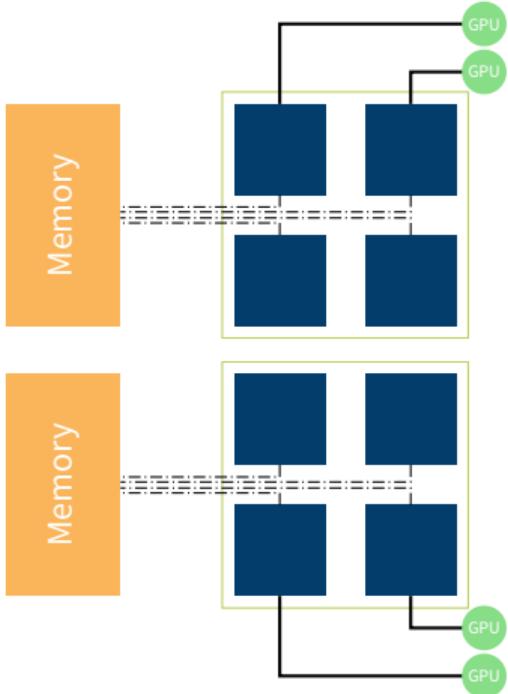
- PCIe via I/O die → also hierarchy
 - GPUs, HCAs connected via PCIe (through PCIe switch)
- *True GPU, HCA affinity only by half of chiplets*
But impact application-dependent and possibly quite low



PCIe Affinity

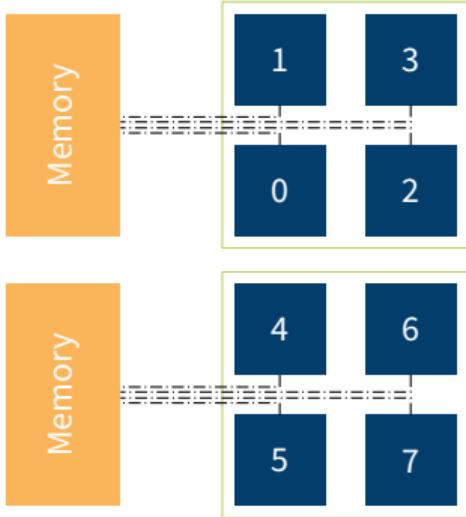


- PCIe via I/O die → also hierarchy
 - GPUs, HCAs connected via PCIe (through PCIe switch)
- *True GPU, HCA affinity only by half of chiplets*
But impact application-dependent and possibly quite low



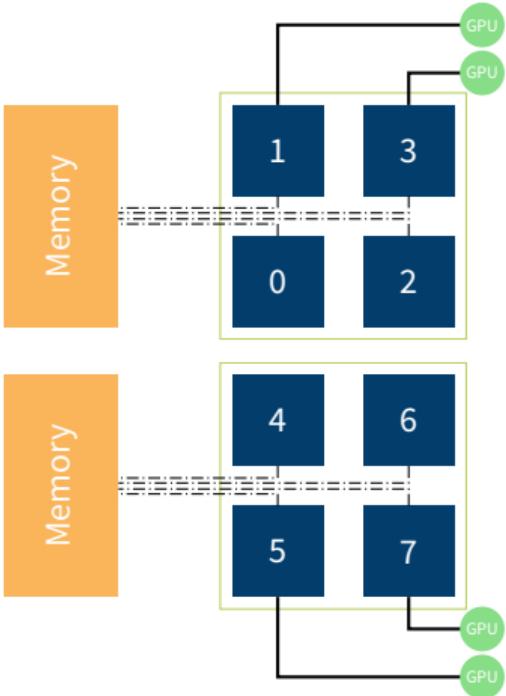
NUMA Numbering

- NUMA domains (= quadrants) are numbered



NUMA Numbering

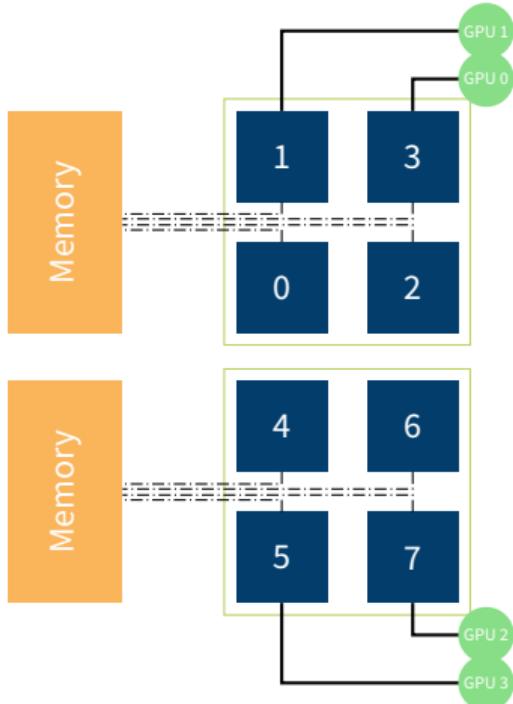
- NUMA domains (= quadrants) are numbered
- But no 1:1 relation between IDs



NUMA Numbering

- NUMA domains (= quadrants) are numbered
- But no 1:1 relation between IDs
- Complete affinity table

Rank	NUMA Domain	GPU ID	HCA ID
0	3	0	0
1	1	1	1
2	7	2	2
3	5	3	3

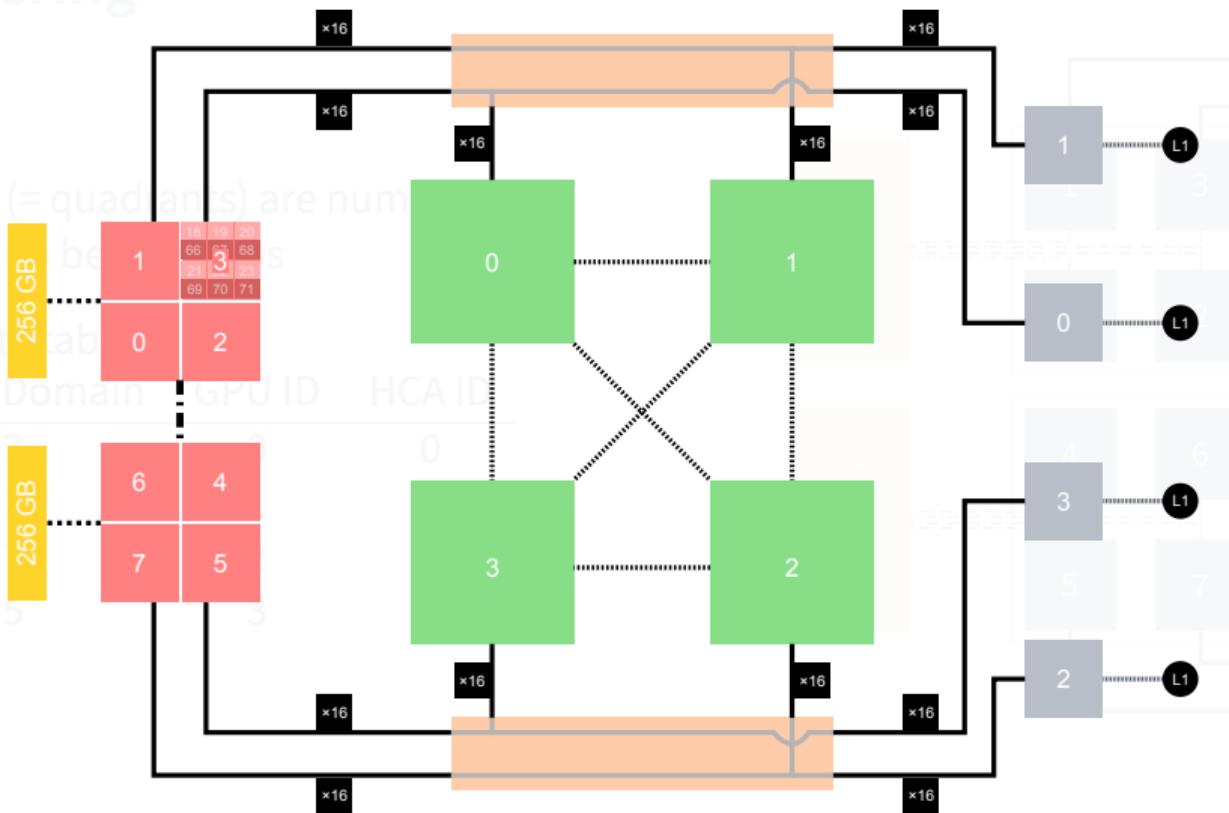


NUMA Numbering

- NUMA domains (= quadrants) are numbered 0-3
- But no 1:1 relation between memory and CPU
- Complete affinity table available

Rank	NUMA Domain	GPU ID	HCA ID
0	2	1	1
1	2	0	0
2	3	1	1
3	3	0	0

Legend:
Memory (Yellow)
CPU (Red)
GPU (Green)



Affinity in Practice

CPU and GPU Affinity

- We configured Slurm (*PSSlurm*) to use best topology **as default**

Affinity in Practice

CPU and GPU Affinity

- We configured Slurm (*PSSlurm*) to use best topology **as default**
- *PSSlurm* sets CPU affinity masks with *GPU-first-approach*: Fill cores with GPU affinity first

Affinity in Practice

CPU and GPU Affinity

- We configured Slurm (*PSSlurm*) to use best topology **as default**
- *PSSlurm* sets CPU affinity masks with *GPU-first-approach*: Fill cores with GPU affinity first
- *PSSlurm* sets `$CUDA_VISIBLE_DEVICES` to associated GPU

Affinity in Practice

CPU and GPU Affinity

- We configured Slurm (*PSSlurm*) to use best topology **as default**
- *PSSlurm* sets CPU affinity masks with *GPU-first-approach*: Fill cores with GPU affinity first
- *PSSlurm* sets `$CUDA_VISIBLE_DEVICES` to associated GPU
- Example GPU affinity for 2 tasks:

```
$ srun -n 2 --cpu-bind=verbose env | grep 'PMI_RANK\|CUDA_VIS'  
cpu_bind=THREADS - jwb0001, task 0 0 [18307]: mask 0x40000 set  
cpu_bind=THREADS - jwb0001, task 1 1 [18309]: mask 0x40 set  
PMI_RANK=0  
CUDA_VISIBLE_DEVICES=0  
PMI_RANK=1  
CUDA_VISIBLE_DEVICES=1
```

DragonFly+ Topology

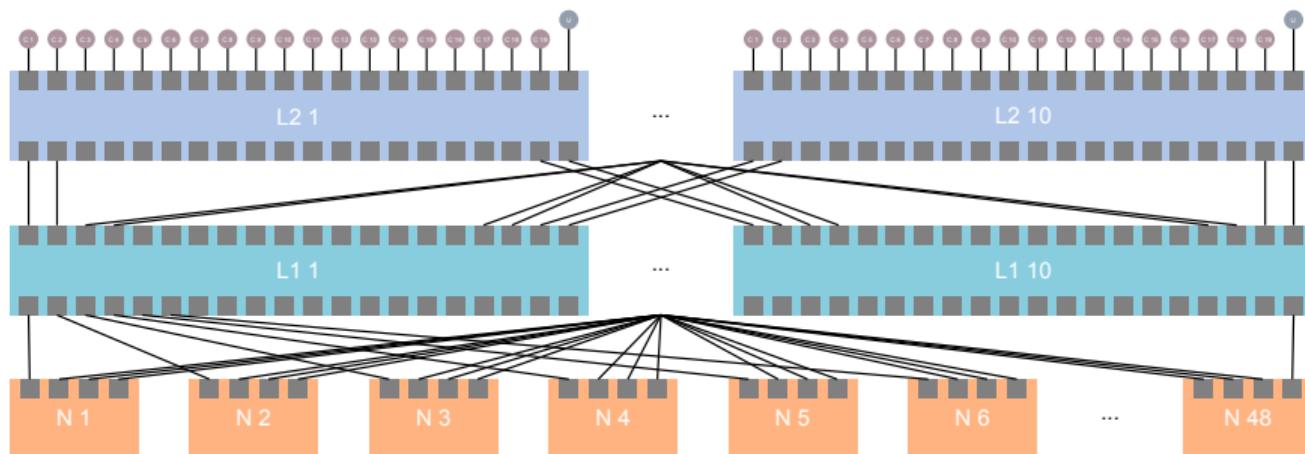
Overview

- 3744 end points; adaptive routing
- HDR200: 200 Gbit/s per link per direction
- Dragonfly+ Topology selected as good way to balance factors (space, investment, performance)
- Two-level topology: local (in-cell), non-local (inter-cell)
- Still learning practical implications of topology

DragonFly+ Topology

In-Cell Topology

- In-cell: full fat-tree in 2 levels
 - 48 nodes, each 4 links (*strided to 4 L1 switches*)
 - 10 L1 switches, 40 port (each L1 switch: 2 links to L2, strided to 10 L2 switches)
 - 10 L2 switches, 40 port (each L2 switch: 1 link to L2 switch of all other cells, 1 link to cluster)



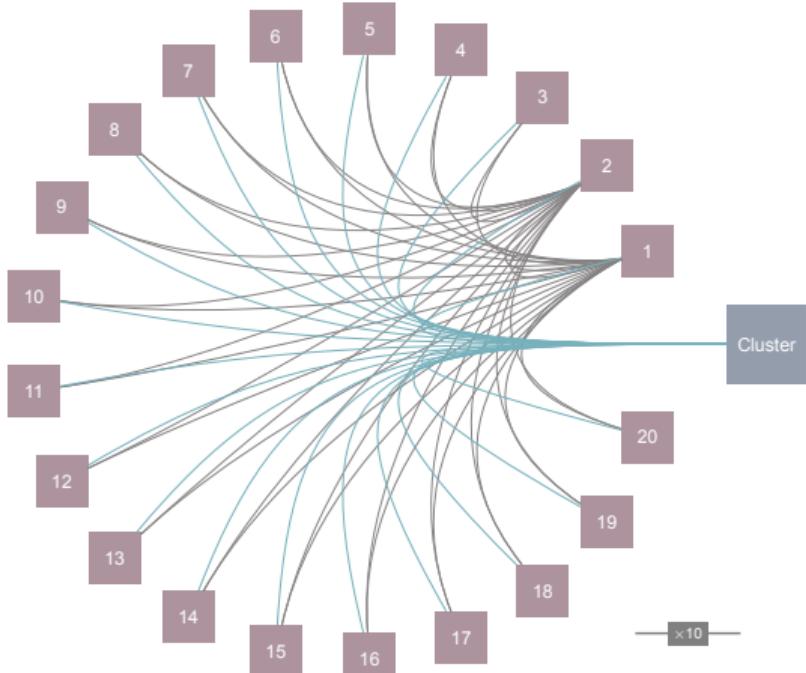
DragonFly+ Topology

Intra-Cell Topology

- Inter-cell: 20 cells

- 10 links between each pair of cells
- 10 links per cell to JUWELS Cluster
- Filesystem access via cell 20
- Bi-section bandwidth between N cells:

$$\mathcal{B}(N) = \lfloor (N/2)^2 \rfloor \times (10 \times bw_1)$$



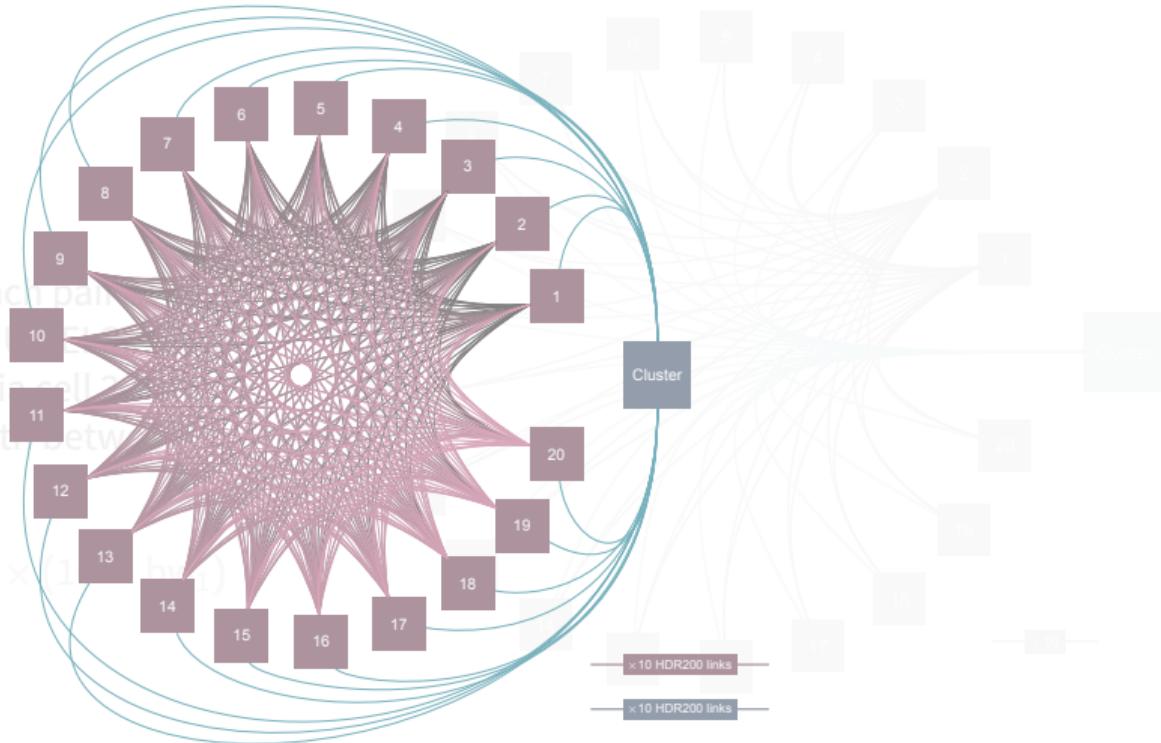
DragonFly+ Topology

Intra-Cell Topology

- Inter-cell: 20 cells

- 10 links between each pair of cells
- 10 links per cell to JSC Cluster
- Filesystem access via cell-to-cell links
- Bi-section bandwidth between two cells:

$$\mathcal{B}(N) = \lfloor (N/2)^2 \rfloor \times 10 \text{ Gbps}$$



Vendors, Models

GPU Vendors in Tutorial

- First Exascale machine: Frontier, with **AMD** GPUs!
- This tutorial: **NVIDIA** examples
 - JUWELS Booster: machine for tutorial
 - No large-scale AMD GPU installation accessible yet *for us*
 - We have most experience with NVIDIA
- **But:** Everything similar/identical for other vendors (especially AMD)
- More on other vendors in last session

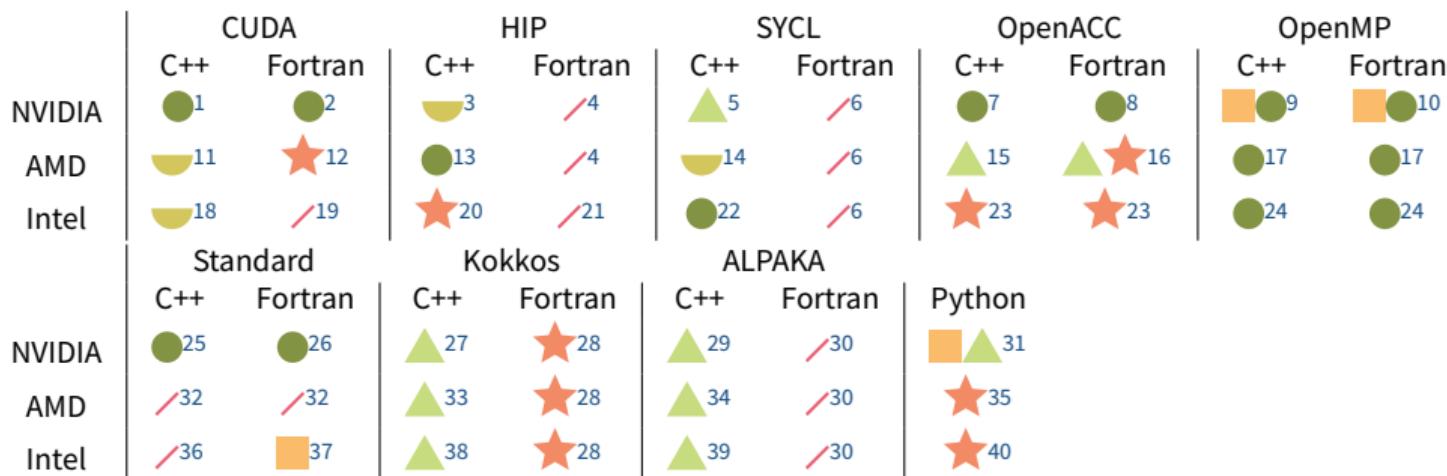
GPU Programming Models

- Full vendor support
- Indirect, but comprehensive support, by vendor
- Vendor support, but not (yet) entirely comprehensive

- Comprehensive support, but not by vendor
- Limited, probably indirect support – but at least some
- / No direct support available, but

of course one could ISO-C-bind your way through it or directly link the libraries

C++ C++ (sometimes also C)
Fortran Fortran



See appendix for explanations, or doi:10.34732/xdvblg-r1bvif

Programming Model in Tutorial

- Tutorial: **CUDA** for GPU programming
 - Many other possibilities, especially for NVIDIA GPUs
 - Some: higher-level abstractions, mapping back to CUDA
 - Conceptionally all similar
 - No significant changes needed to use MPI
- Transfer CUDA knowledge to other models

Summary and Conclusions

Summary and Conclusions

- Exascale and Pre-Exascale systems mainly based on GPUs, with thousands of devices
- Many advanced technologies in place to enable large-scale GPU applications
- Tutorial with team experienced in distributed GPU workloads
- Supercomputer of tutorial: JUWELS Booster, European flagship system based on A100 GPUs and HDR200 InfiniBand network

Summary and Conclusions

- Exascale and Pre-Exascale systems mainly based on GPUs, with thousands of devices
- Many advanced technologies in place to enable large-scale GPU applications
- Tutorial with team experienced in distributed GPU workloads
- Supercomputer of tutorial: JUWELS Booster, European flagship system based on A100 GPUs and HDR200 InfiniBand network

Thank you
for your attention!
a.herten@fz-juelich.de

Appendix

Appendix

Vendor/Programming Model Table

Appendix

Vendor/Programming Model Table:

GPU Vendor/Programming Model Table I

- 1: CUDA C/C++ is supported on NVIDIA GPUs through the [CUDA Toolkit](#)
- 2: CUDA Fortran, a proprietary Fortran extension, is supported on NVIDIA GPUs via the [NVIDIA HPC SDK](#)
- 3: [HIP](#) programs can directly use NVIDIA GPUs via a CUDA backend; HIP is maintained by AMD
- 4: No such thing like HIP for Fortran, but AMD offers Fortran interfaces to HIP and ROCm libraries in [hipfort](#)
- 5: SYCL can be used on NVIDIA GPUs with *experimental* support either in [SYCL](#) directly or in [DPC++](#), or via [hipSYCL](#)
- 6: No such thing like SYCL for Fortran
- 7: OpenACC C/C++ supported on NVIDIA GPUs directly (and best) through NVIDIA HPC SDK; additional, somewhat limited support by [GCC C compiler](#) and in LLVM through [Clacc](#)
- 8: OpenACC Fortran supported on NVIDIA GPUs directly (and best) through NVIDIA HPC SDK; additional, somewhat limited support by GCC Fortran compiler and [Flacc](#)
- 9: OpenMP in C++ supported on NVIDIA GPUs through NVIDIA HPC SDK (albeit [with a few limits](#)), by GCC, and Clang; see [OpenMP ECP BoF on status in 2022](#).
- 10: OpenMP in Fortran supported on NVIDIA GPUs through NVIDIA HPC SDK (but not full OpenMP feature set available), by GCC, and Flang
- 25: pSTL features supported on NVIDIA GPUs through [NVIDIA HPC SDK](#)
- 26: Standard Language parallel features supported on NVIDIA GPUs through NVIDIA HPC SDK
- 27: [Kokkos](#) supports NVIDIA GPUs by calling CUDA as part of the compilation process
- 28: Kokkos is a C++ model, but an official compatibility layer ([Fortran Language Compatibility Layer, FLCL](#)) is available.

GPU Vendor/Programming Model Table II

- 29: [Alpaka](#) supports NVIDIA GPUs by calling CUDA as part of the compilation process; also, an OpenMP backend can be used
- 30: Alpaka is a C++ model
- 31: There is a vast community of offloading Python code to NVIDIA GPUs, like [CuPy](#), [Numba](#), [cuNumeric](#), and many others; NVIDIA actively supports a lot of them, but has no direct product like *CUDA for Python*; so, the status is somewhere in between
- 11: [hipify](#) by AMD can translate CUDA calls to HIP calls which runs natively on AMD GPUs
- 12: AMD offers a Source-to-Source translator to convert some CUDA Fortran functionality to OpenMP for AMD GPUs ([gpufort](#)); in addition, there are ROCm library bindings for Fortran in [hipfort](#) OpenACC/CUDA Fortran Source-to-Source translator
- 13: [HIP](#) is the preferred native programming model for AMD GPUs
- 14: SYCL can use AMD GPUs, for example with [hipSYCL](#) or [DPC++ for HIP AMD](#)
- 15: OpenACC C/C++ can be used on AMD GPUs via GCC or Clacc; also, Intel's [OpenACC to OpenMP Source-to-Source translator](#) can be used to generate OpenMP directives from OpenACC directives
- 16: OpenACC Fortran can be used on AMD GPUs via GCC; also, AMD's [gpufort](#) Source-to-Source translator can move OpenACC Fortran code to OpenMP Fortran code, and also Intel's translator can work
- 17: AMD offers a dedicated, Clang-based compiler for using OpenMP on AMD GPUs: [AOMP](#); it supports both C/C++ (Clang) and Fortran (Flang, [example](#))

GPU Vendor/Programming Model Table III

- 32: Currently, no (known) way to launch Standard-based parallel algorithms on AMD GPUs
- 33: Kokkos supports AMD GPUs through HIP
- 34: Alpaka supports AMD GPUs through HIP or through an OpenMP backend
- 35: AMD does not officially support GPU programming with Python (also not semi-officially like NVIDIA), but third-party support is available, for example through [Numba](#) (currently inactive) or a [HIP version of CuPy](#)
- 18: [SYCLomatic](#) translates CUDA code to SYCL code, allowing it to run on Intel GPUs; also, Intel's [DPC++ Compatibility Tool](#) can transform CUDA to SYCL
- 19: No direct support, only via ISO C bindings, but at least an example can be [found on GitHub](#); it's pretty scarce and not by Intel itself, though
- 20: [CHIP-SPV](#) supports mapping CUDA and HIP to OpenCL and Intel's Level Zero, making it run on Intel GPUs
- 21: No such thing like HIP for Fortran
- 22: [SYCL](#) is the prime programming model for Intel GPUs; actually, SYCL is only a standard, while Intel's implementation of it is called [DPC++ \(Data Parallel C++\)](#), which extends the SYCL standard in various places; actually actually, Intel namespaces everything *oneAPI* these days, so the *full* proper name is Intel oneAPI DPC++ (which incorporates a C++ compiler and also a library)
- 23: OpenACC can be used on Intel GPUs by translating the code to OpenMP with [Intel's Source-to-Source translator](#)
- 24: Intel has [extensive support for OpenMP](#) through their latest compilers

GPU Vendor/Programming Model Table IV

- 36: Intel supports pSTL algorithms through their [DPC++ Library](#) (oneDPL; [GitHub](#)). It's heavily namespaced and not yet on the same level as NVIDIA
- 37: With [Intel oneAPI 2022.3](#), Intel supports DO CONCURRENT with GPU offloading
- 38: Kokkos supports Intel GPUs through SYCL
- 39: [Alpaka v0.9.0](#) introduces experimental SYCL support; also, Alpaka can use OpenMP backends
- 40: Not a lot of support available at the moment, but notably [DPNP](#), a SYCL-based drop-in replacement for Numpy, and [numba-dpex](#), an extension of Numba for DPC++.