Hewlett Packard
Enterprise

PDC Summer School 2024 GPU Training

Tim Dykes & Harvey Richardson
HPE HPC/AI EMEA Research lab

August 20/21, 2024

HPE HPC/AI EMEA Research Lab

Partnering with leading organizations in the EMEA region to advance supercomputing R&D

Our Role

- Deep technical collaboration with industry, academia, and public sector.
- Long term technical relationships surrounding research, co-design, and operational support.
- Focus on new technologies, driving HPE products.
- Create reusable PoCs & European IP

Research Interests

- HPC, Cloud, AI, Quantum
- Data movement, analysis, and workflows
- Heterogeneous computing and novel accelerators
- Programming languages and models
- Compilers and mathematical optimisation
- Performance portability, security, and containerisation
- Energy efficiency and sustainability

Engagement Models

- Centres of Excellence
- Advanced Collaboration Centres
- Value-add projects
- Joint-funded research projects
- Nationally/internationally funded research projects
- Ph.D. and Placements

A research team in the CTO office of the HPC, AI & Labs business unit.

Advanced Collaboration Centers In EMEA

ARCHER2, UK

- First Cray EX system
- New training
- Now in production
- Next step to agree future projects
- Likely a Monitoring and Analytics project Power/Network/IO
- Application tuning and GPU testbed

KAUST, KSA

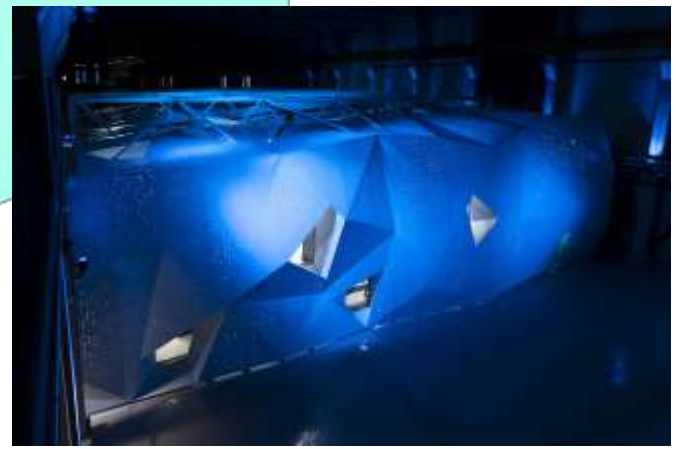
- Numerical linear algebra libraries
- Combustion CFD
- Deep Learning for Bio-Science

UoB & GW4

- Isambard AI
- Isambard 3
- ARM system tuning
- ARM ecosystem development
- Grace/Grace-Hopper
- Joint ARM, Cavium partnership
- Fujitsu A64fx
- MACS

LUMI

- European Pre-Exascale
- Initial projects covering Containerisation, Easybuild and porting to LUMI-G (GPUS).
- Federated HPC
- Holistic Energy-Awareness
- Secure/Trustworthy HPC



Who are the presenters?

Tim Dykes

Ph.D. in HPC, Astronomy & Computer Graphics @ University of Portsmouth, U.K.

Joined Cray Inc. in 2017

Cray acquired by HPE in 2019

Research Interests: Accelerators, programming models, memory hierarchies, scientific visualization, …

Harvey Richardson

BSc in Computational Physics, PhD in Physics, Heriot-Watt University, Edinburgh, UK

Previously worked at Thinking Machines Corporation, Sun Microsystems and joined Cray in 2010.

Cray acquired by HPE in 2019.

Interests: Computer architecture, performance, programming models, language standards

+ content from various colleagues and collaborators within HPE and AMD

GPU Training Agenda

01. GPU Architecture and Programming Models

- What does a GPU hardware architecture look like?
- How are they integrated into supercomputers?
- How do we use them?

02. Introductory Programming of AMD GPUs with HIP

- How can the HIP model be used to program the AMD GPUs available in Dardel?
- How does this compare to NVIDIA GPUs?
- Intro to hands-on

03. Advanced GPU Programming and Debugging

- How do I efficiently utilize these GPUs for scientific programming?
- How do I troubleshoot, diagnose, and resolve problems?

04. Portability Across GPU Architectures and Languages

- How do I write performance portable software?
- How do I use GPUs in [my favourite language]?

Hewlett Packard
Enterprise

01. GPU Architecture and Programming Models

01. GPU Architecture and Programming Models: Overview

- A Brief History
- Cray EX Supercomputer Architecture
- Compute Blade Architecture
- GPU High Level Architecture
- Dardel GPU Blades
- MI250X GPU Architecture
- GPU Programming Models

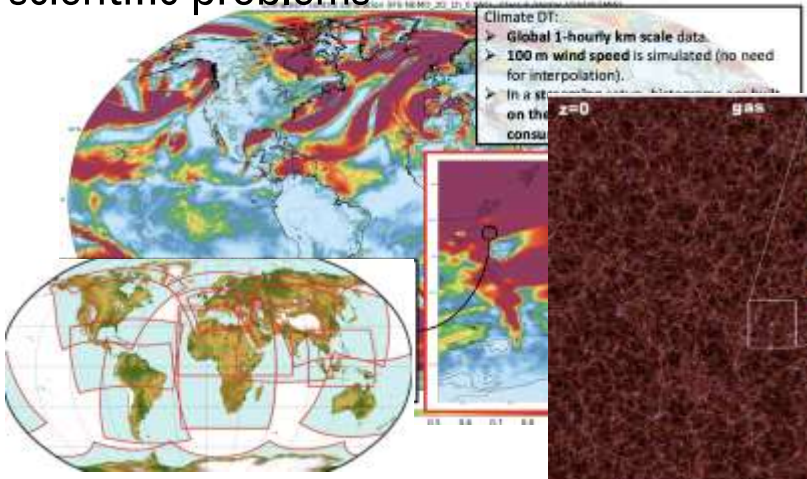
A [very] brief history of Graphics Programming Units

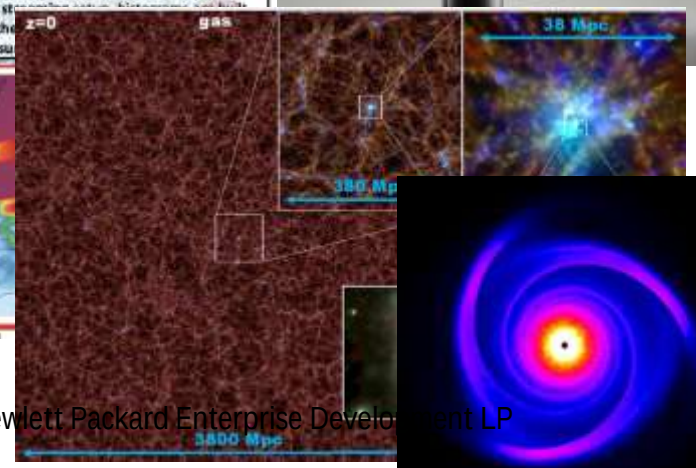
- The early years: 70s/80s
  - Increase in GUI-based computers increased demand for graphics capabilities
  - Simple framebuffers and 2d graphics accelerators for text/image display

- The advent of the GPU: 90s
  - 3D graphics for gaming and professional workstations
  - Hardware requirements for graphical interfaces, real-time rendering, offline rendering, image processing, video editing, etc.

- Transition to General Purpose Computing: 00s
  - Researchers realised GPU hardware is well suited to many scientific problems too
  - The term GPGPU is coined

- The rise of GPU-Accelerated Supercomputing: 10s
  - GPU programmability improves
  - Supercomputing community and vendors cotton on
  - GPU hardware evolution and specialisation





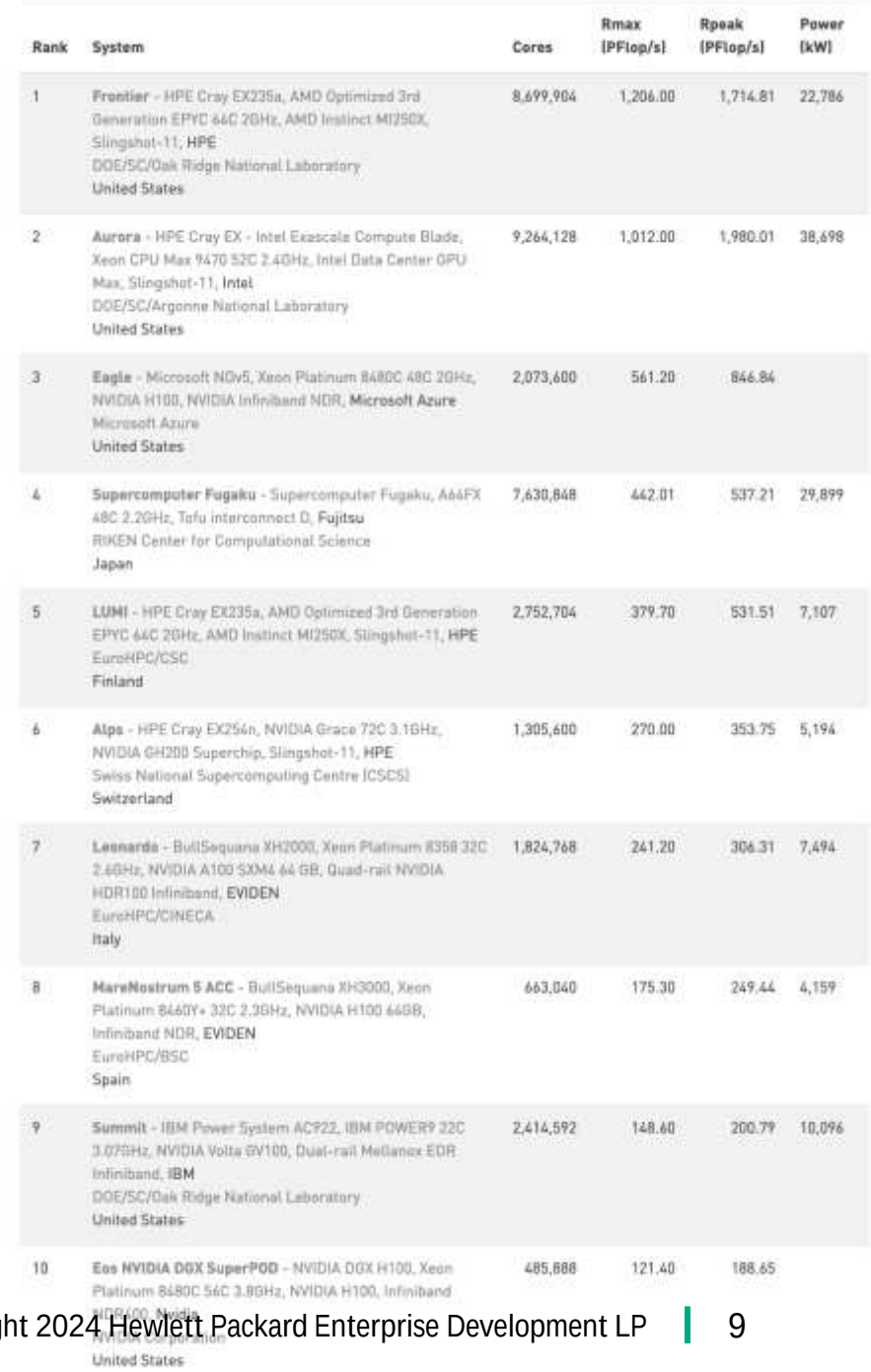
GPUs Rise to Power in Supercomputing

- Early 00s, heterogeneous HPC systems start to show promise with:
  - TSUBAME 1.0: AMD Opteron + ClearSpeed CS600 accelerators ~2006
  - IBM Roadrunner: AMD Opteron + IBM PowerXCell 8i Processors ~2007/8
  - Tianhe 1 / 1A: Intel Xeon + AMD ATI Radeon HD 4870, then later Nvidia Tesla m2050s

- In intervening years, GPU architectures diverge and specialize for supercomputing
  - Increased parallelism, reducing hw features for rendering pipelines, texture mapping, ray tracing
  - Increased precision capability (emphasis on double over single for graphics)
  - High bandwidth memory architectures (over low latency for graphics)
  - Data center integration
  - Programmability improvements (!)

- Not to mention huge additional investment from AI explosion!

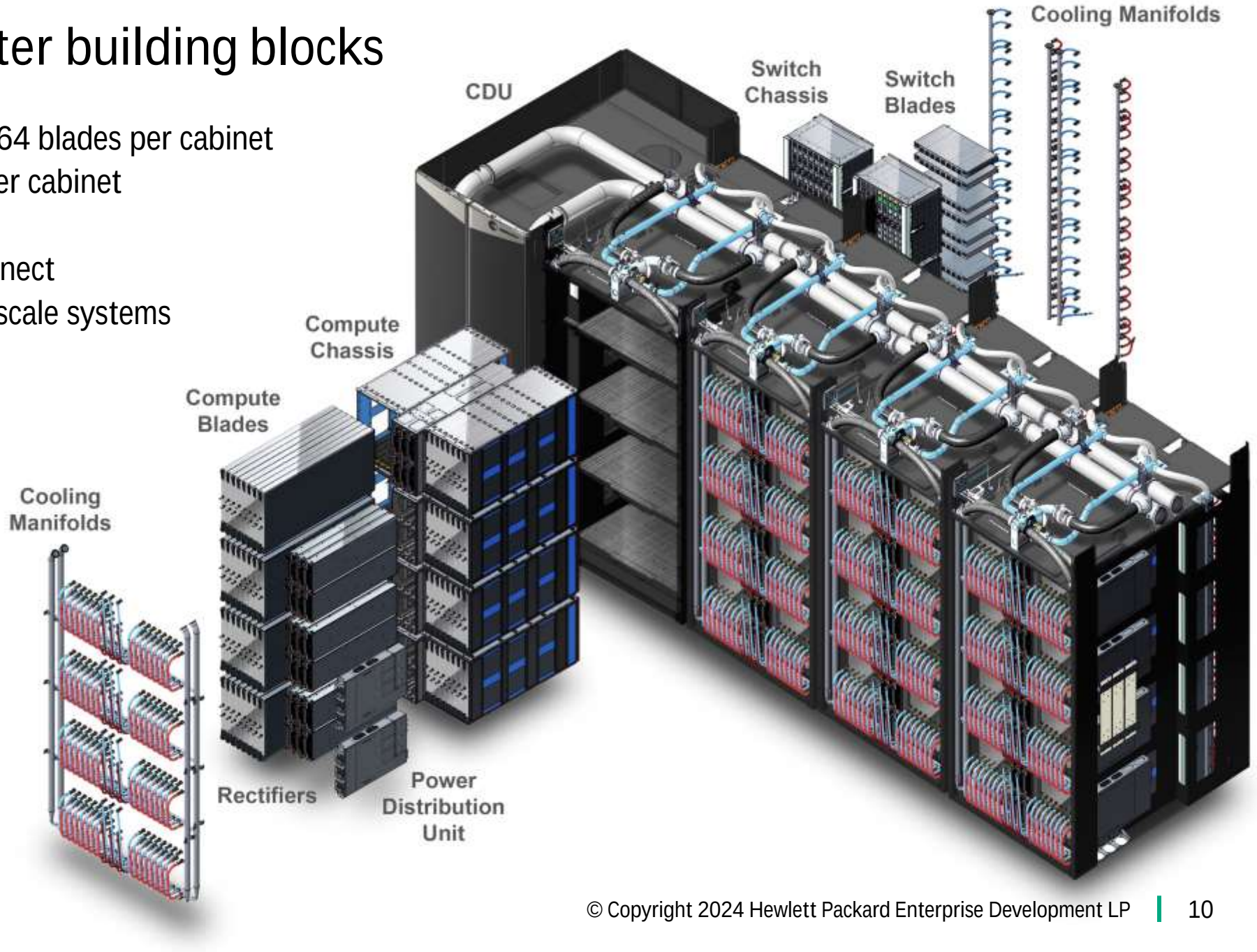
- Leading up to … the current day:
  - GPUs become the cornerstone of most (but not all) large scale supercomputer deployments, via 3 major hardware vendors.
  - 9 out of top 10; 10 out of green 10; 193 out of top 500;



Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, **HPE** DOE/SC/Oak Ridge National Laboratory **United States**	8,699,904	1,206.00	1,714.81	22,786
2	**Aurora** - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, **Intel** DOE/SC/Argonne National Laboratory **United States**	9,264,128	1,012.00	1,980.01	38,698
3	**Eagle** - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, NVIDIA Infiniband NDR, **Microsoft Azure** Microsoft Azure **United States**	2,073,600	561.20	846.84	
4	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, **Fujitsu** RIKEN Center for Computational Science **Japan**	7,630,848	442.01	537.21	29,899
5	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, **HPE** EuroHPC/CSC **Finland**	2,752,704	379.70	531.51	7,107
6	**Alps** - HPE Cray EX254n, NVIDIA Grace 72C 3.1GHz, NVIDIA GH200 Superchip, Slingshot-11, **HPE** Swiss National Supercomputing Centre (CSCS) **Switzerland**	1,305,600	270.00	353.75	5,194
7	**Leonardo** - BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 64 GB, Quad-rail NVIDIA HDR100 Infiniband, **EVIDEN** EuroHPC/CINECA **Italy**	1,824,768	241.20	306.31	7,494
8	**MareNostrum 5 ACC** - BullSequana XH3000, Xeon Platinum 8460Y+ 32C 2.3GHz, NVIDIA H100 64GB, Infiniband NDR, **EVIDEN** EuroHPC/BSC **Spain**	663,040	175.30	249.44	4,159
9	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, **IBM** DOE/SC/Oak Ridge National Laboratory **United States**	2,414,592	148.60	200.79	10,096
10	**Eos NVIDIA DGX SuperPOD** - NVIDIA DGX H100, Xeon Platinum 8480C 56C 3.8GHz, NVIDIA H100, Infiniband NDR400, Nvidia NVIDIA Corporation **United States**	485,888	121.40	188.65	

Cray EX supercomputer building blocks

- 1 – 100s of cabinets with up to 64 blades per cabinet
- Up to 512 CPUs or 256 GPUs per cabinet
- 1 - 100K+ nodes
- HPE Slingshot 200GbE interconnect
- 3 Exascale systems, lots of petascale systems in construction / delivery



Cray EX System Compute Building Blocks

Compute Blade

Up to 4 Compute Nodes

Chassis

Up to 8 Compute Blades

Up to 32 Compute Nodes

Cabinet

1x Rank 0 Network Group

Passive Electrical Network

8 Chassis

Up to 256 Compute Nodes

System

Rank 1 Network

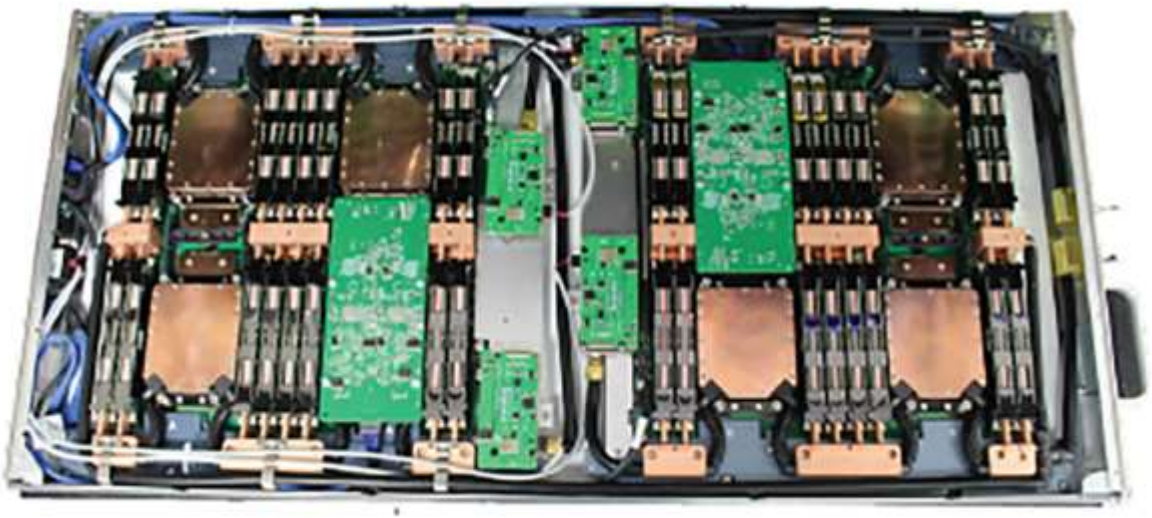
Active Optical Network

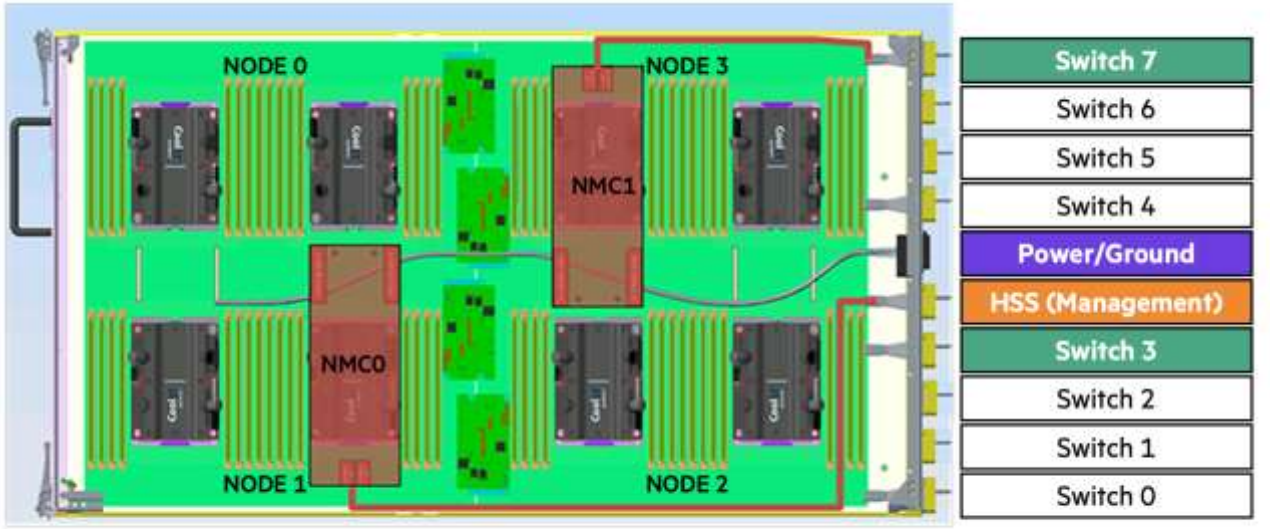
Hundreds of Cabinets

Up to 10s of thousands of nodes

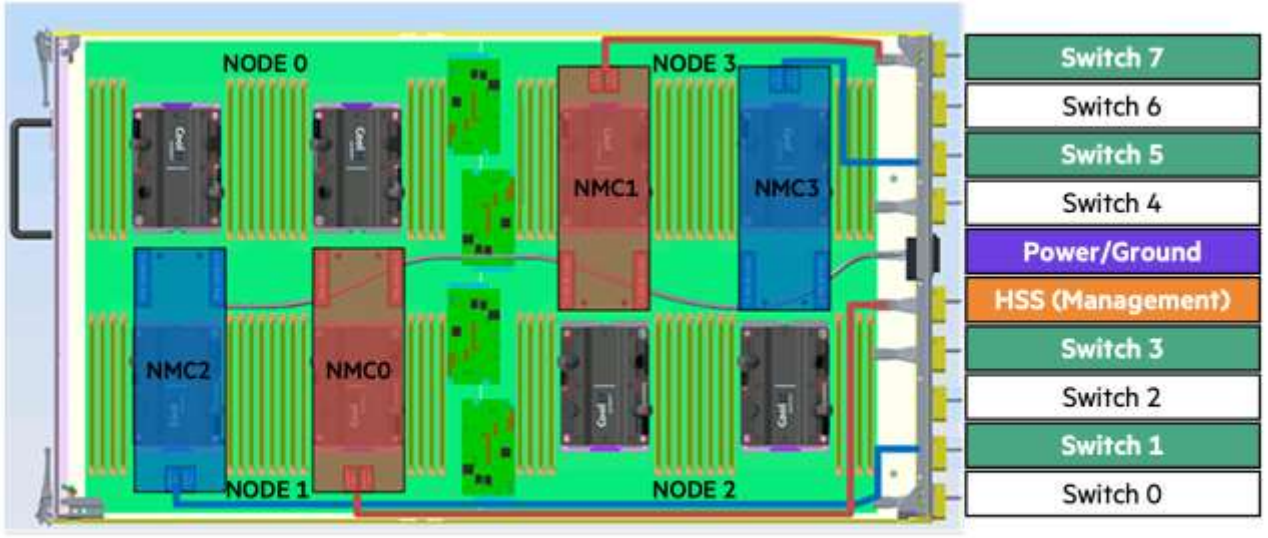
CPU Blades

- Your home computer usually would have a motherboard with 1 CPU
- Supercomputers have custom designed blades with multiple CPUs, memory, network
- EX: 4 nodes, liquid cooled, multiple vendor architectures





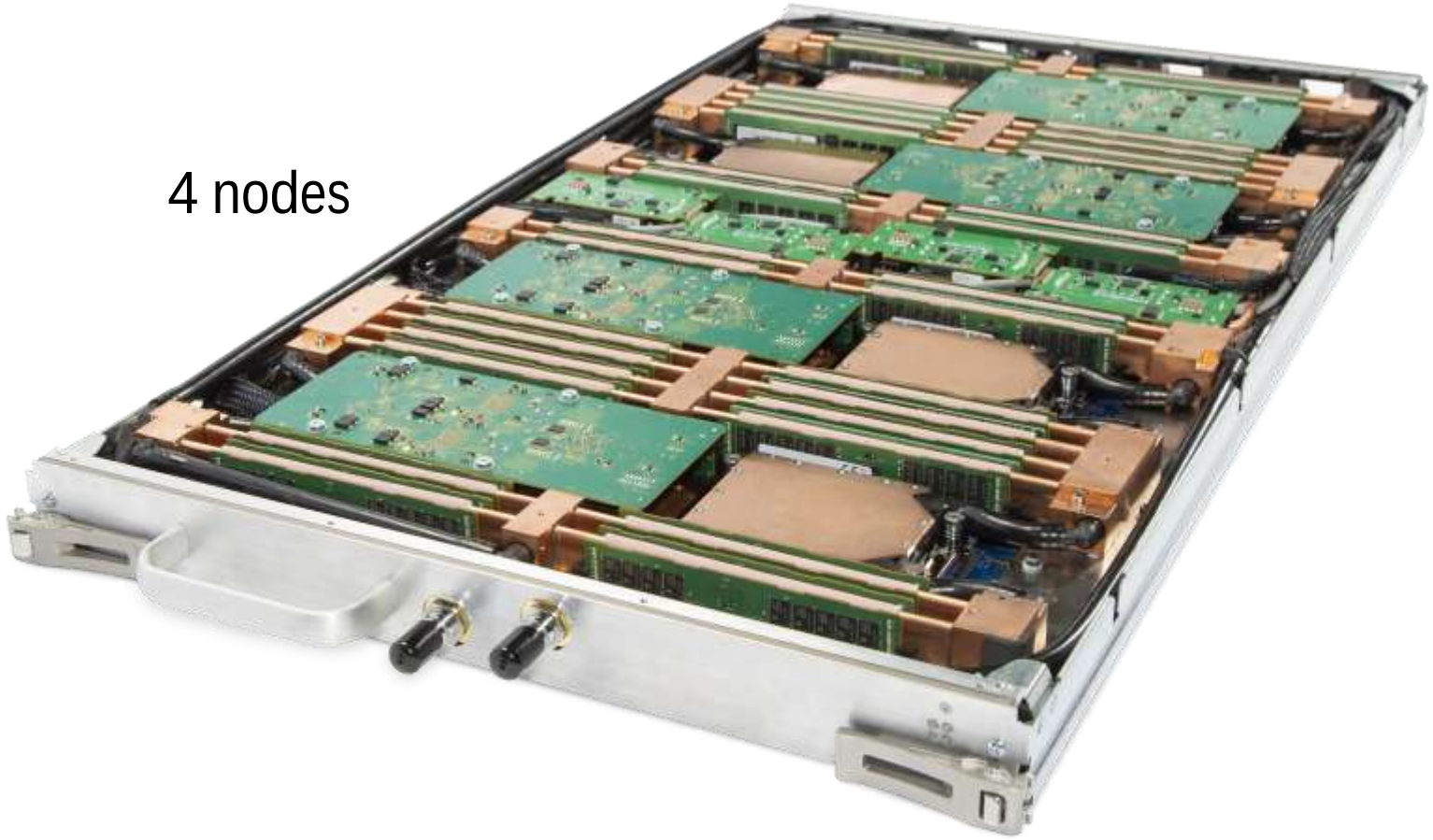
HPE Cray EX compute blade with single injection port per node



HPE Cray EX compute blade with dual injection port per node

Compute blade architecture for Dardel CPU Blades (EX425)

4 nodes

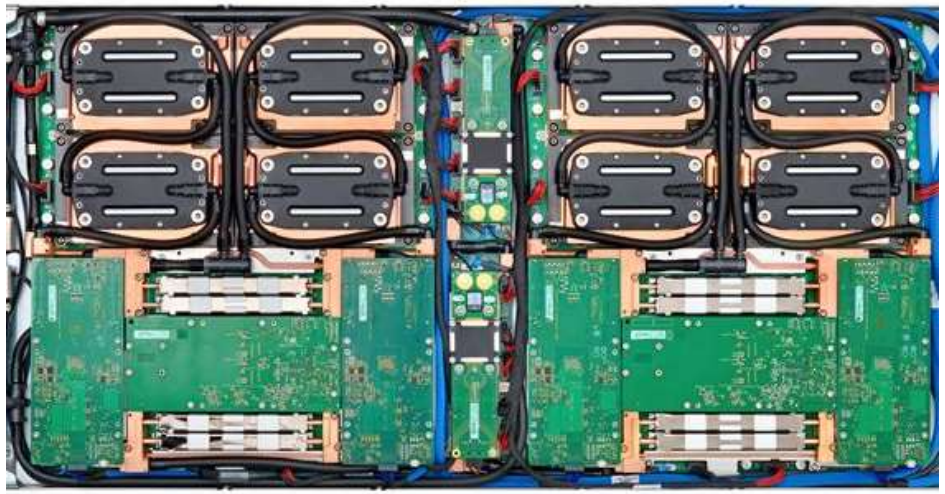


Each node:

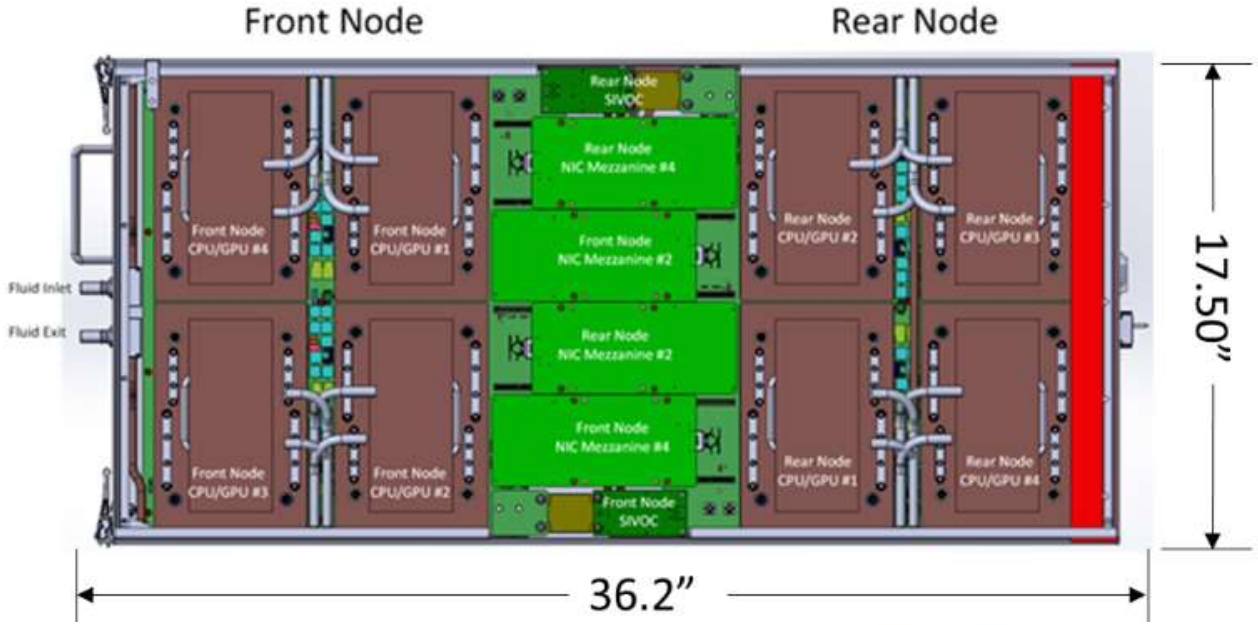
- 2 x AMD EPYC 7742 64 core 2.25GHz
- 16 x 16/32/64 GB DDR4 3200
- 256/512/1024 GB per node 2-8GB per core
- 1 x 200Gb/s injection port per node

Accelerator Blades

- Home PC and Enterprise GPUs are most likely PCI devices attached to your motherboard
- Supercomputers have custom blades, typically with GPU in sockets.
- EX: 2 nodes per blade, liquid cooled, multiple vendor architectures

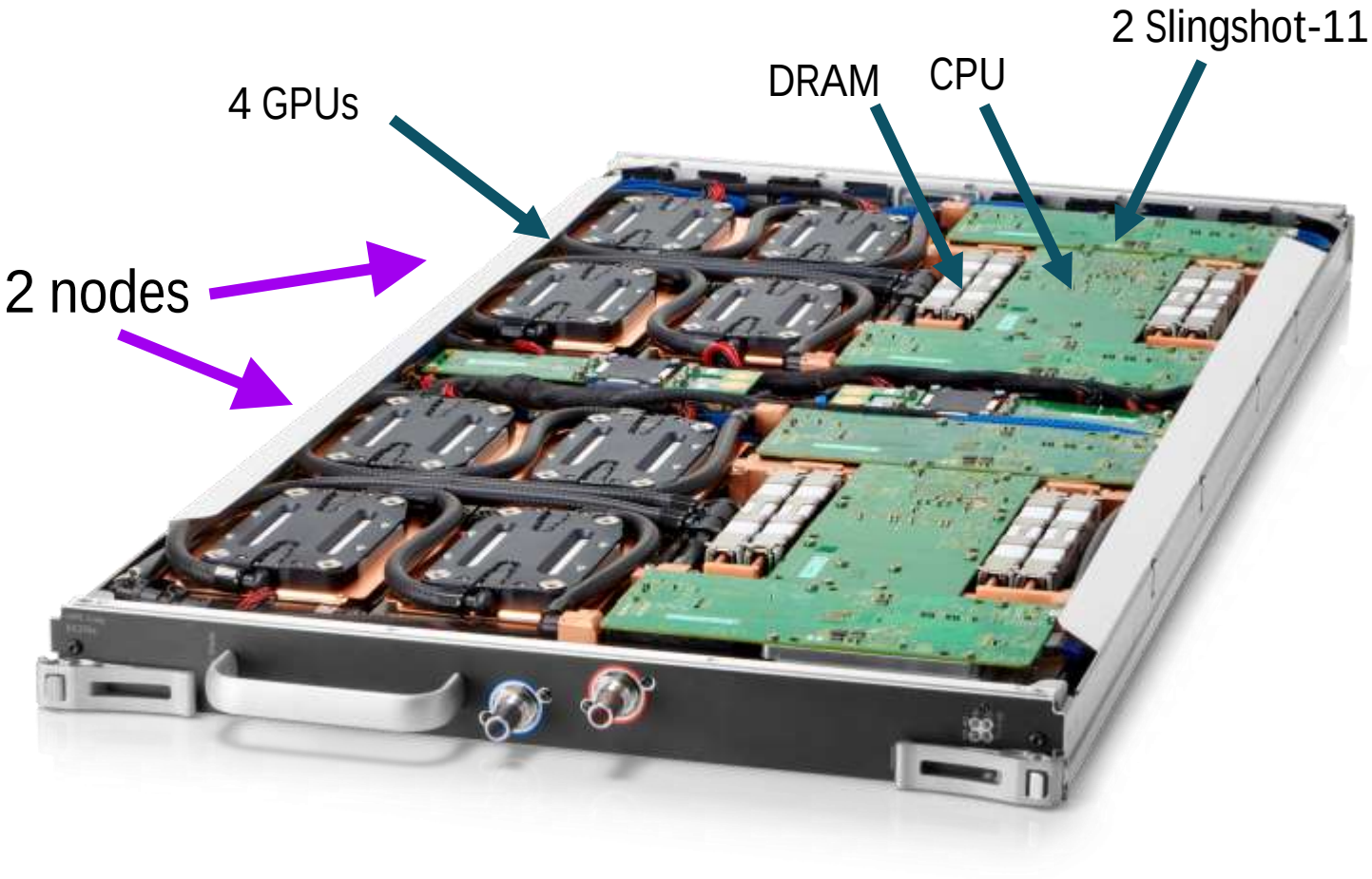


Frontier blade: AMD Mi250X GPUs



Isambard-AI blade: NVIDIA GH200 GPUs

Compute Blade Architecture for Dardel GPU Blades (EX235A)

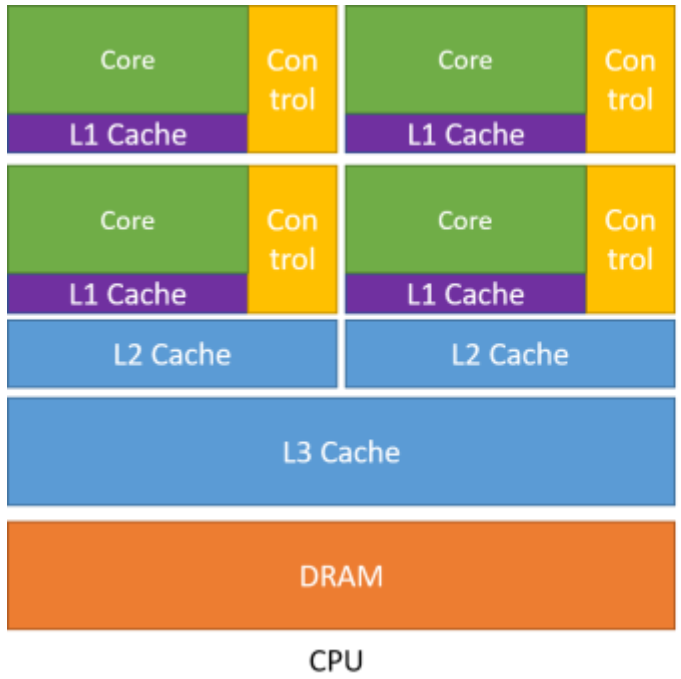


Each node:

- AMD EPYC 7A53 “Optimized 3rd Gen EPYC” 64-Core Processor, 2.00 GHz
- 512 GB DDR4 memory
- 4x AMD MI-250X GPU each with 128GB HBM2e
- Each GPU connected to a Slingshot 200Gb/s NIC

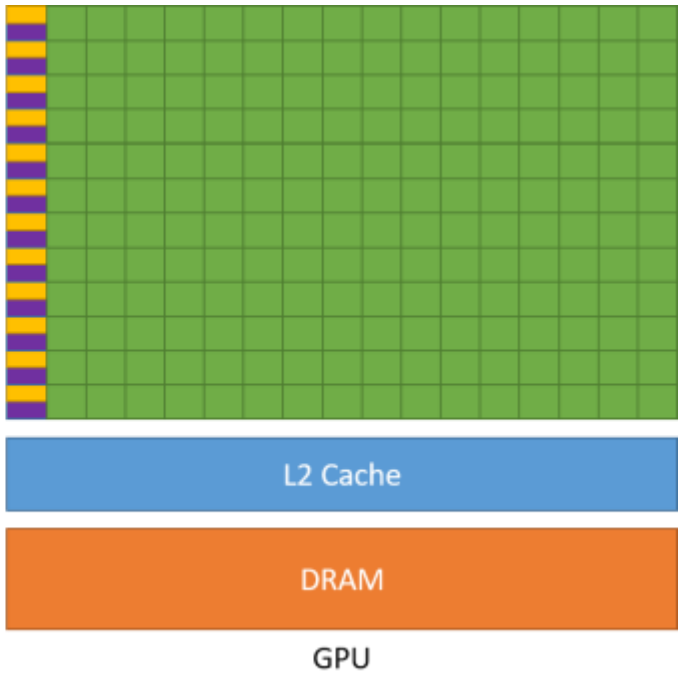
Accelerators – why do we need them in our blades?

- Specialized parallel hardware for floating point operations
  - Based on massive parallel architectures
  - Co-processors for traditional CPUs (which are used for generic workloads)



CPU

- Low compute density
- Large caches
- Optimized for serial operations
- Latency optimized

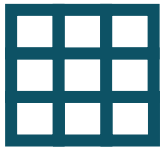


GPU

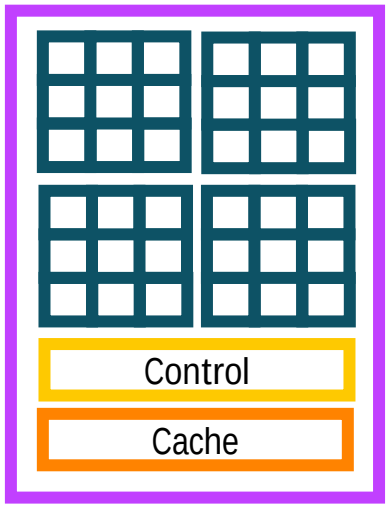
- High compute density
- High computations per Memory Access
- Built for parallel operations
- Throughput optimized

Source: CUDA C++ Programming Guide

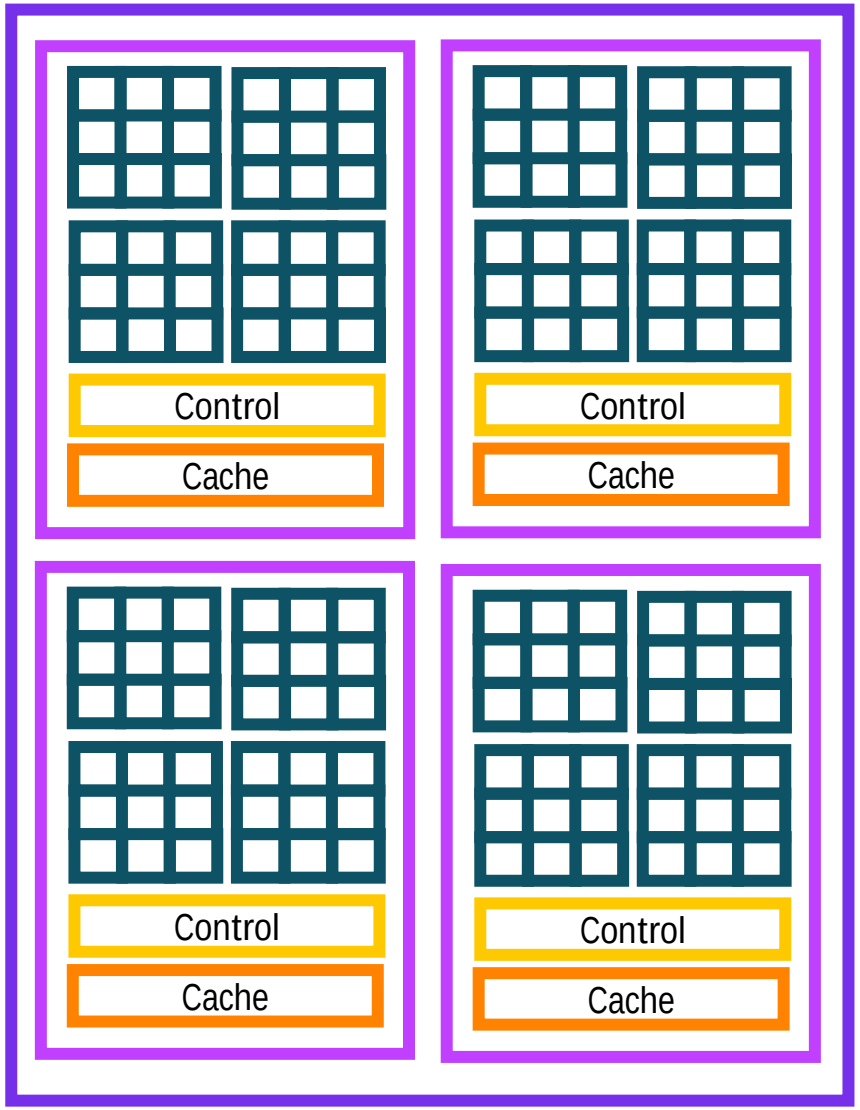
Building Blocks for GPU Architecture



Build some ALUs
AMD: Shader Cores / Matrix Cores
Nvidia: Cuda Cores / Tensor Cores
Intel: Vector Engine / Matrix Engine



Group into a functional unit
add some control logic and cache
AMD: Compute Unit
Nvidia: Streaming Multiprocessor
Intel: Xe Core

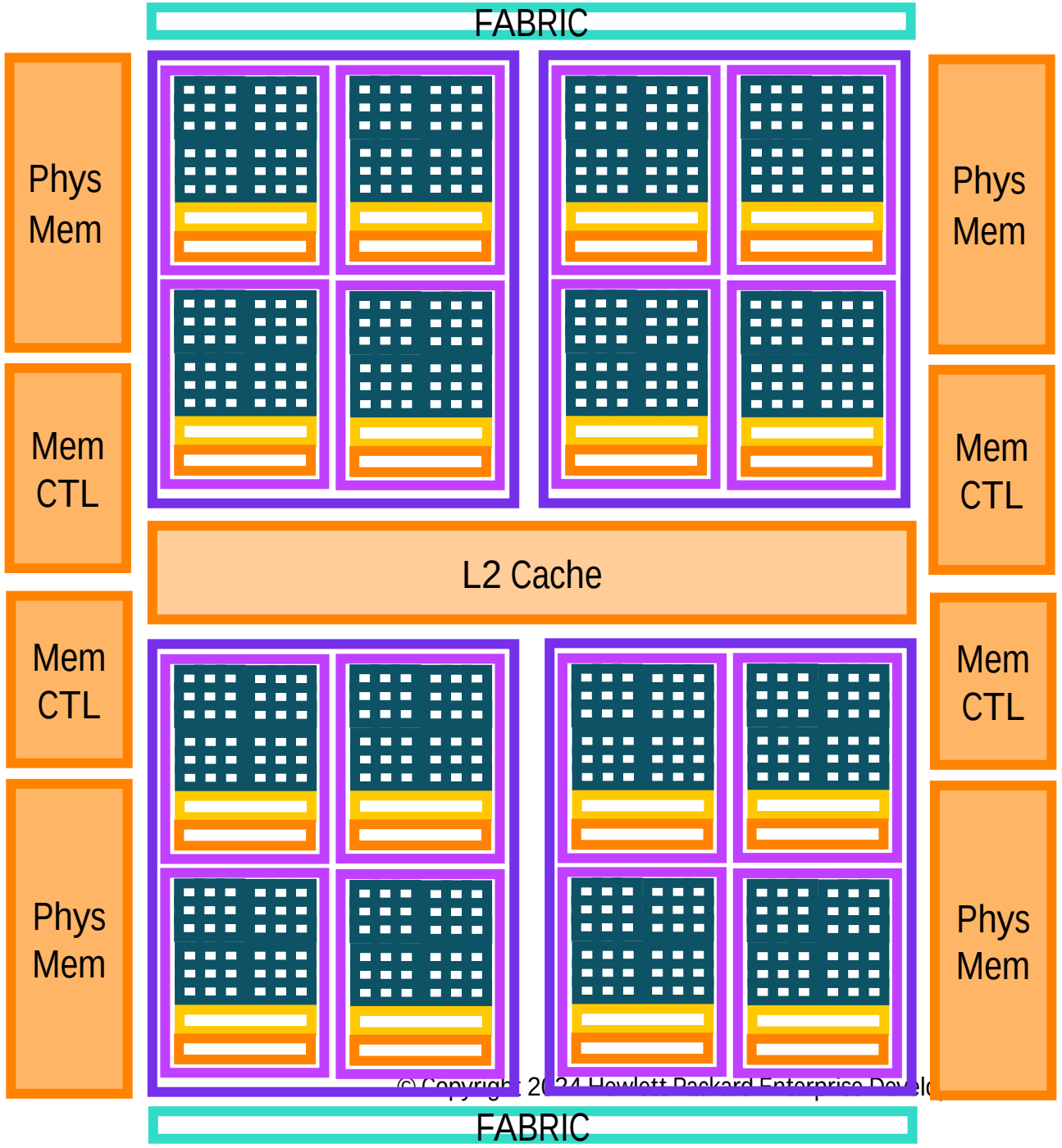


Group into a functional unit and add some special sauce
AMD: Compute Engine. NVIDIA: GPU Processing Cluster Intel: Xe Slice

From Functional units to GPUs

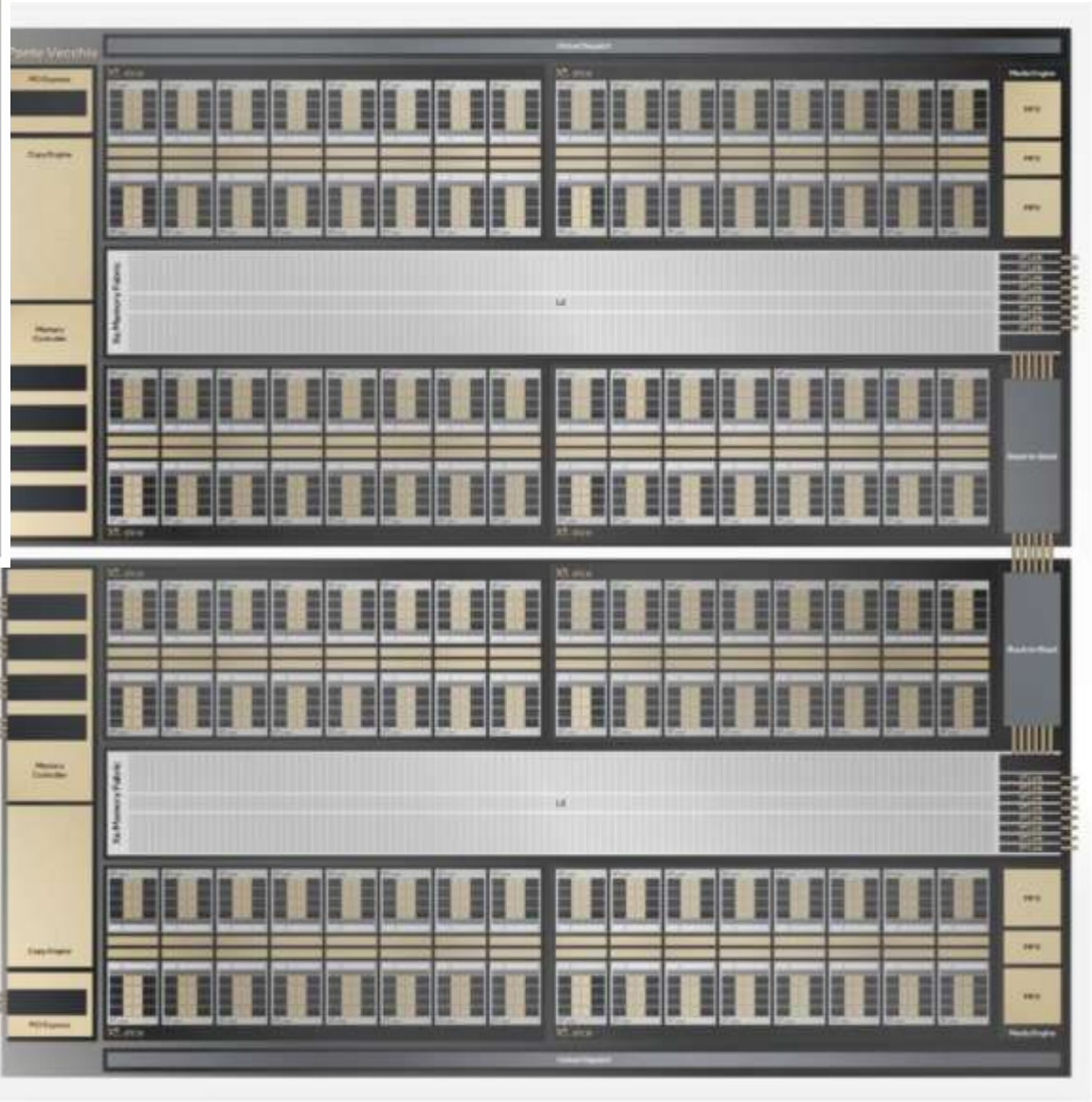
Group into a functional unit
add more cache, memory controllers,
physical memory, fabric interfaces
AMD: Graphics Compute Die (GCD)
Nvidia: GPU
Intel: Xe Stack

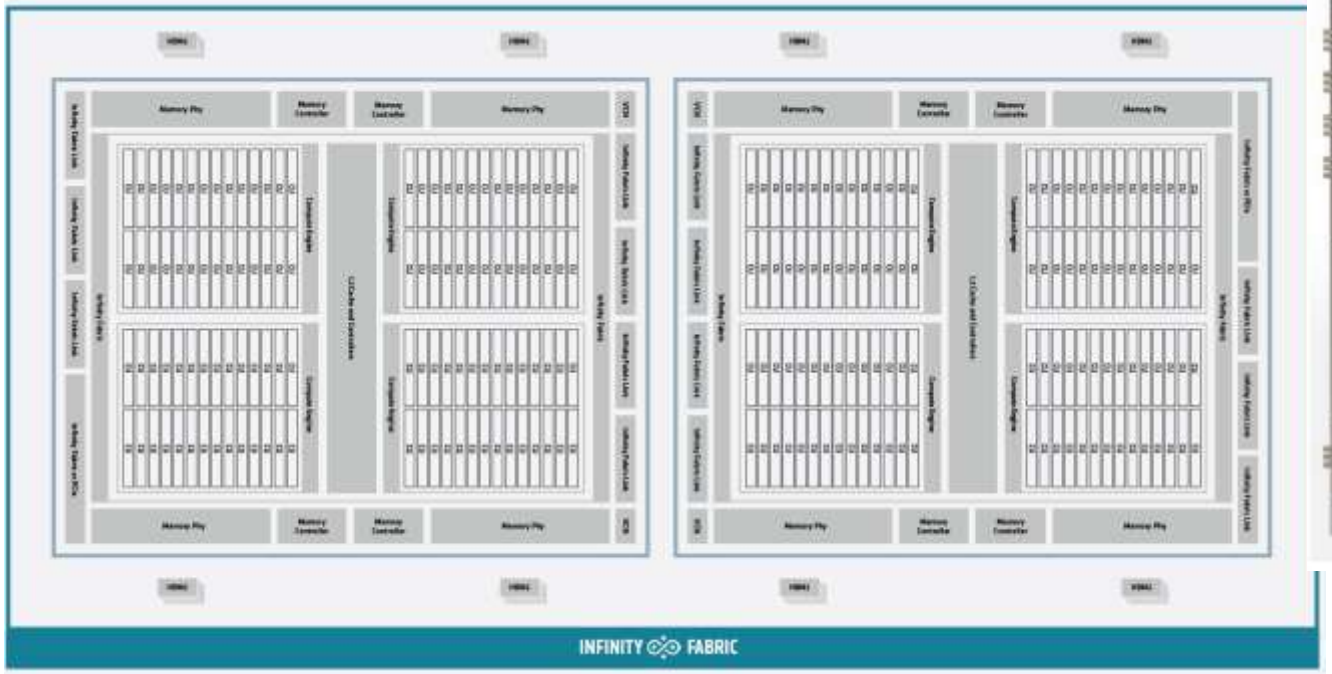
Stick one or more of these on a backplane
And interconnect them
et voila: A GPU!



Nvidia H100 Whitepaper

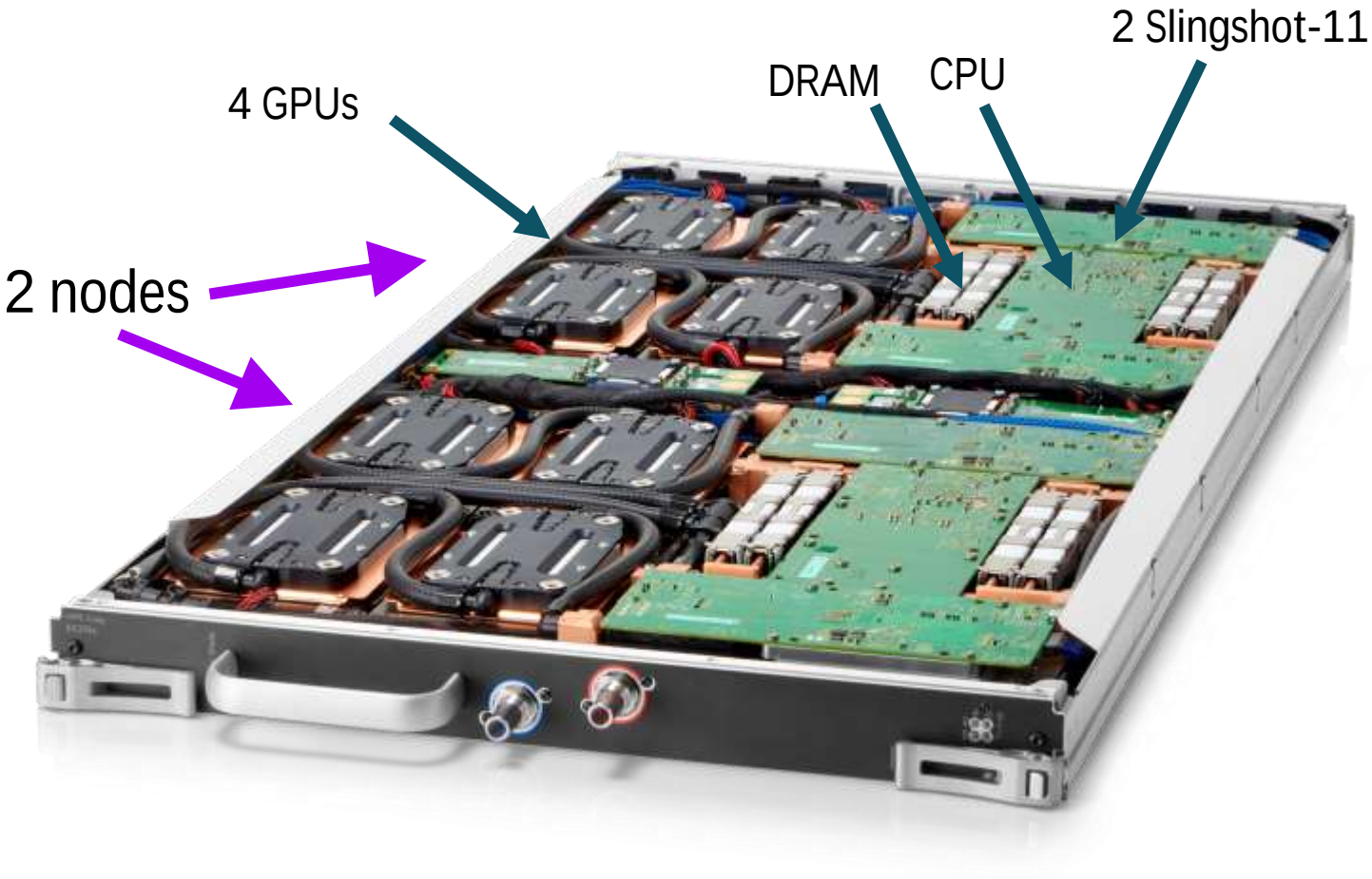
Intel Ponte Vecchio Whitepaper





AMD MI250x Whitepaper

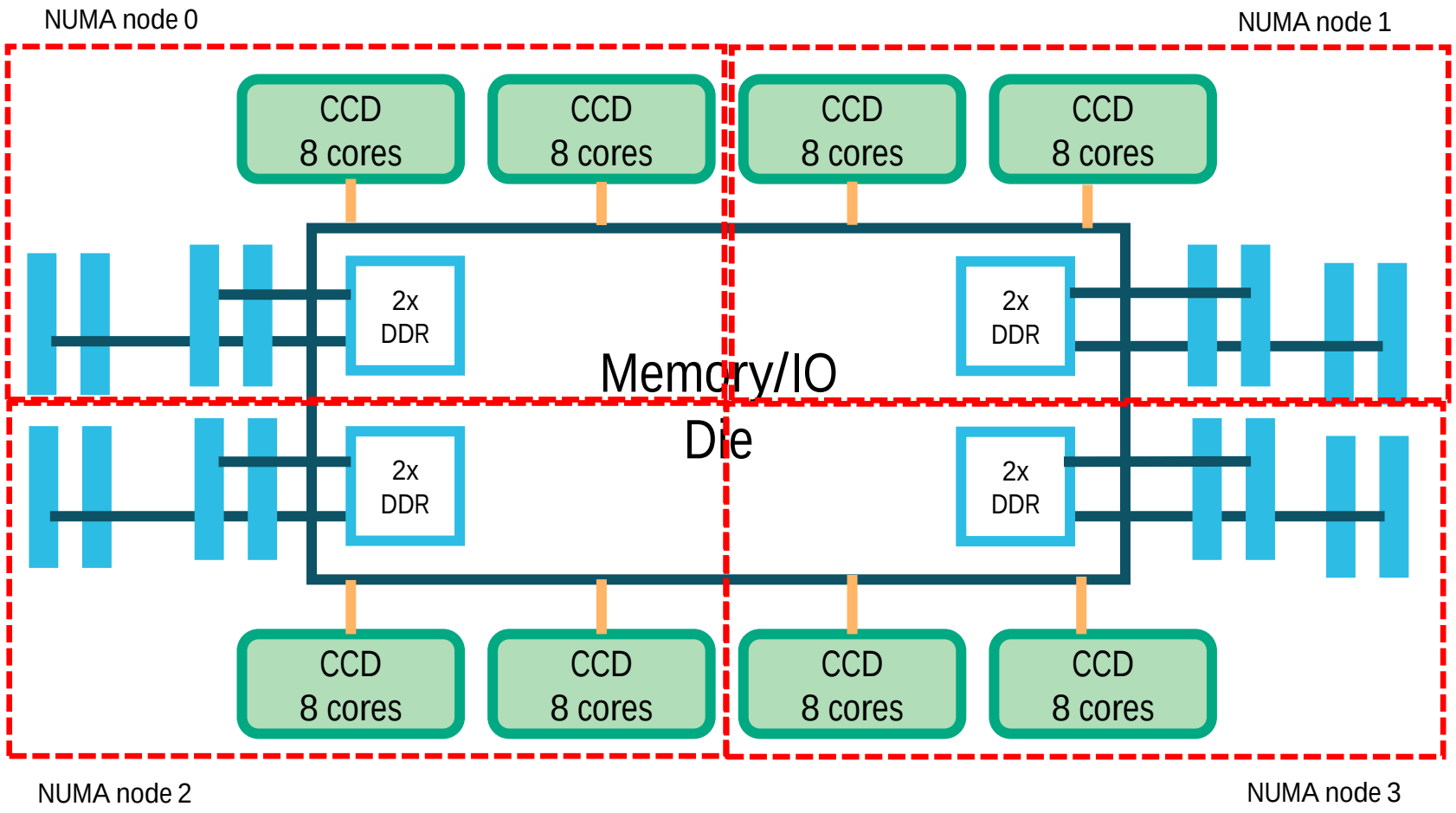
Coming back to our GPU compute blade…



Each node:

- AMD EPYC 7A53 “Optimized 3rd Gen EPYC” 64-Core Processor, 2.00 GHz
- 512 GB DDR4 memory
- 4x AMD MI-250X GPU each with 128GB HBM2e
- Each GPU connected to a Slingshot 200Gb/s NIC

AMD EPYC 7A53 processor

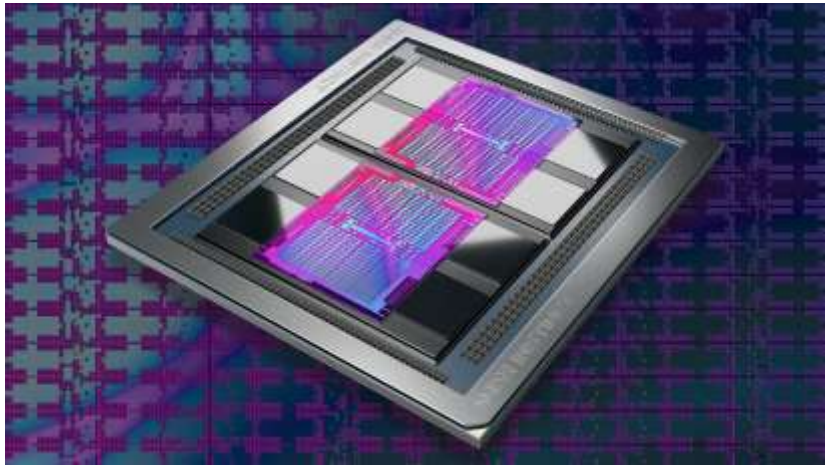


AMD EPYC 7A53

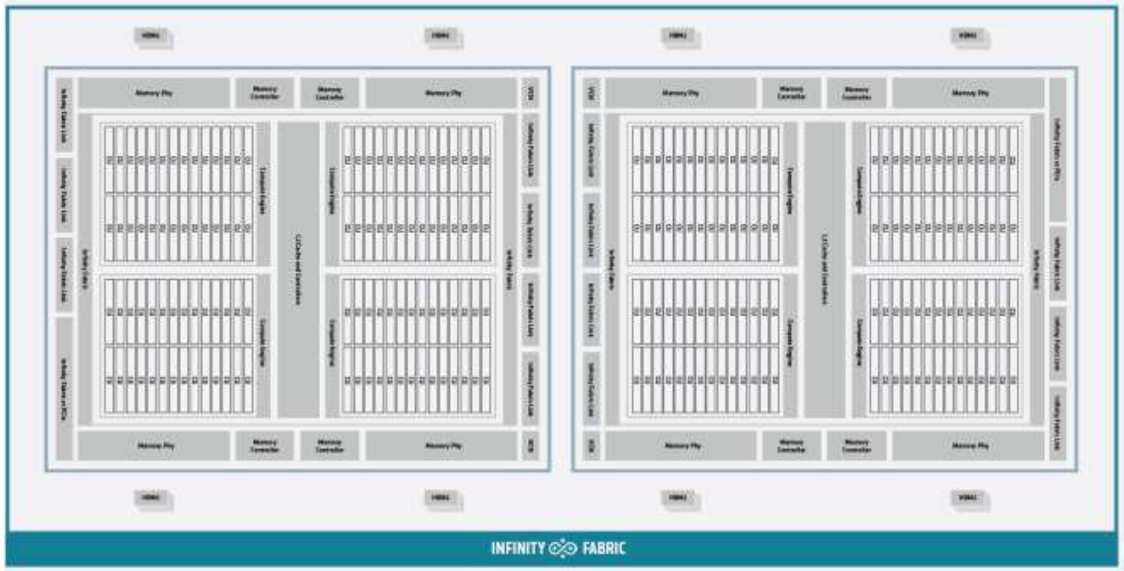
- Base clock 2.00 GHz
- 64 cores,
  128 hardware threads
- L1 cache 32kB / core
- L2 cache 512kB / core
- L3 cache 32MB / 8-cores
  256MB L3 cache in total
- 128 PCIe 4.0 lanes
- 8 channel DDR 3200MHz,
  204.8 GB/s peak b/w
- Chiplet based architecture
- Configured as 4
  NUMA nodes
- Vector support: AVX2

Mi250X GPU Architecture

- Two compute dies (Graphics Compute Dies (GCDs))
  - Interconnected with 200 GB/s per direction
- Dedicated Memory (HBM2e) Size: 128 GB
  - High bandwidth device memory (up to 3.2 TB/s)
  - Memory Clock: 1.6 GHz
- AMD CDNA2 Architecture
  - 110 Compute Units (CU) per each die = 220 CUs
  - 64 SIMD threads per each CU = 14080 Stream Processors
  - Peak FP64/FP32 Vector: 47.9 TFLOPS
  - Peak FP64/FP32 Matrix: 95.7 TFLOPS
  - Total L2 cache: 8 MB per each die (64kB per CU)
  - Frequency: up to 1700 MHz
  - Max power: up to 560 Watts
- Memory coherency with CPU



Source: https://www.amd.com/en/products/server-accelerators/instinct-mi250

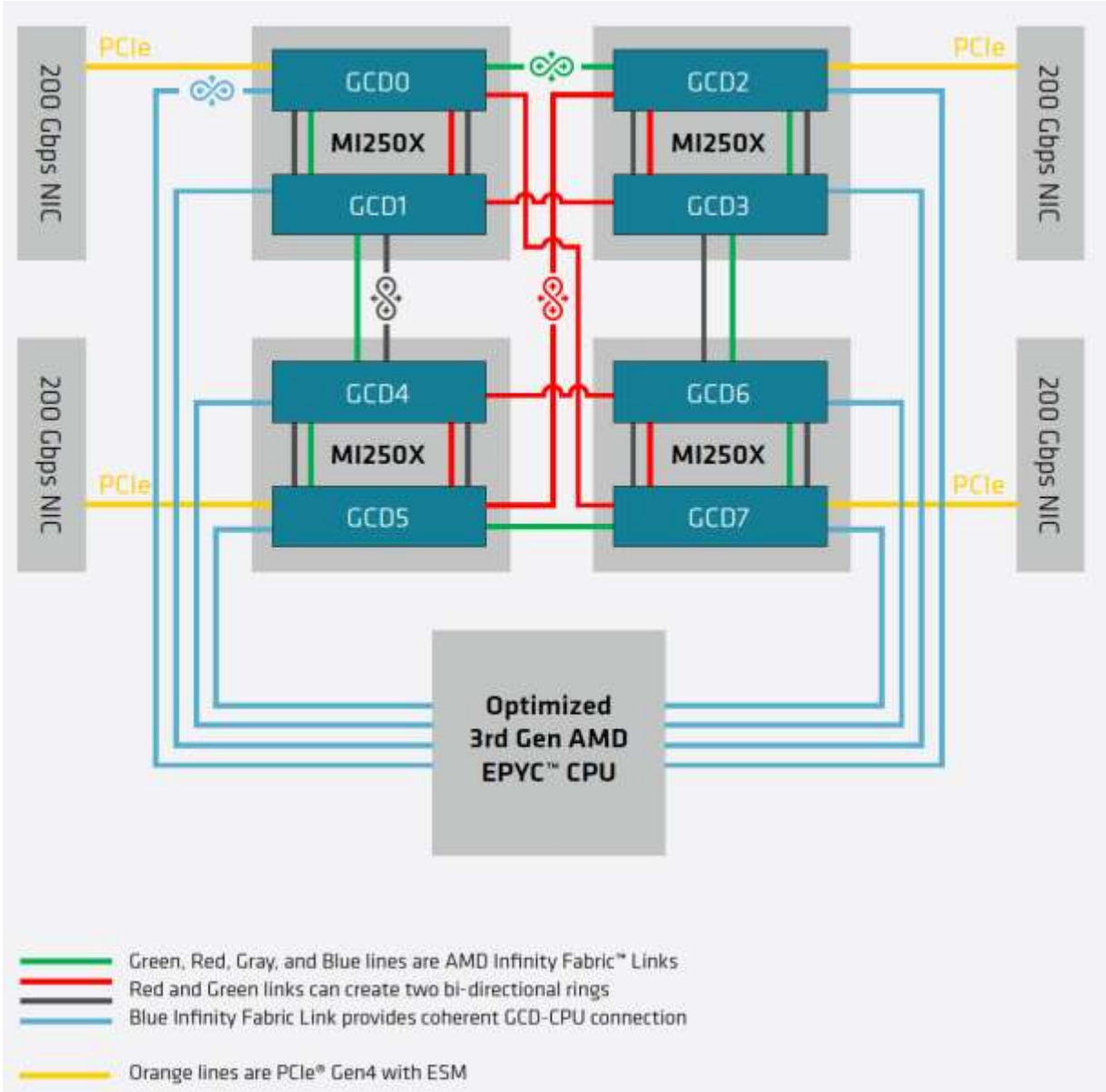


Source: (https://www.amd.com/system/files/documents/amd-cdna2-white-paper.pdf)

Node Architecture

- The programmer can think of the 8 GCDs as 8 separate GPUs, each having 64 GB of high-bandwidth memory (HBM2E)
- The CPU is connected to each GCD via Infinity Fabric CPU-GPU, allowing a peak host-to-device (H2D) and device-to-host (D2H) bandwidth of 36+36 GB/s
  - Coherent memory CPU-GPU
- The 2 GCDs on the same MI250X are connected with Infinity Fabric GPU-GPU
- The GCDs on different MI250X are connected with Infinity Fabric GPU-GPU in the arrangement shown in the diagram on the right, where the peak bandwidth ranges from 50-100 GB/s based on the number of Infinity Fabric connections between individual GCDs

Source: https://www.amd.com/system/files/documents/amd-cdna2-white-paper.pdf

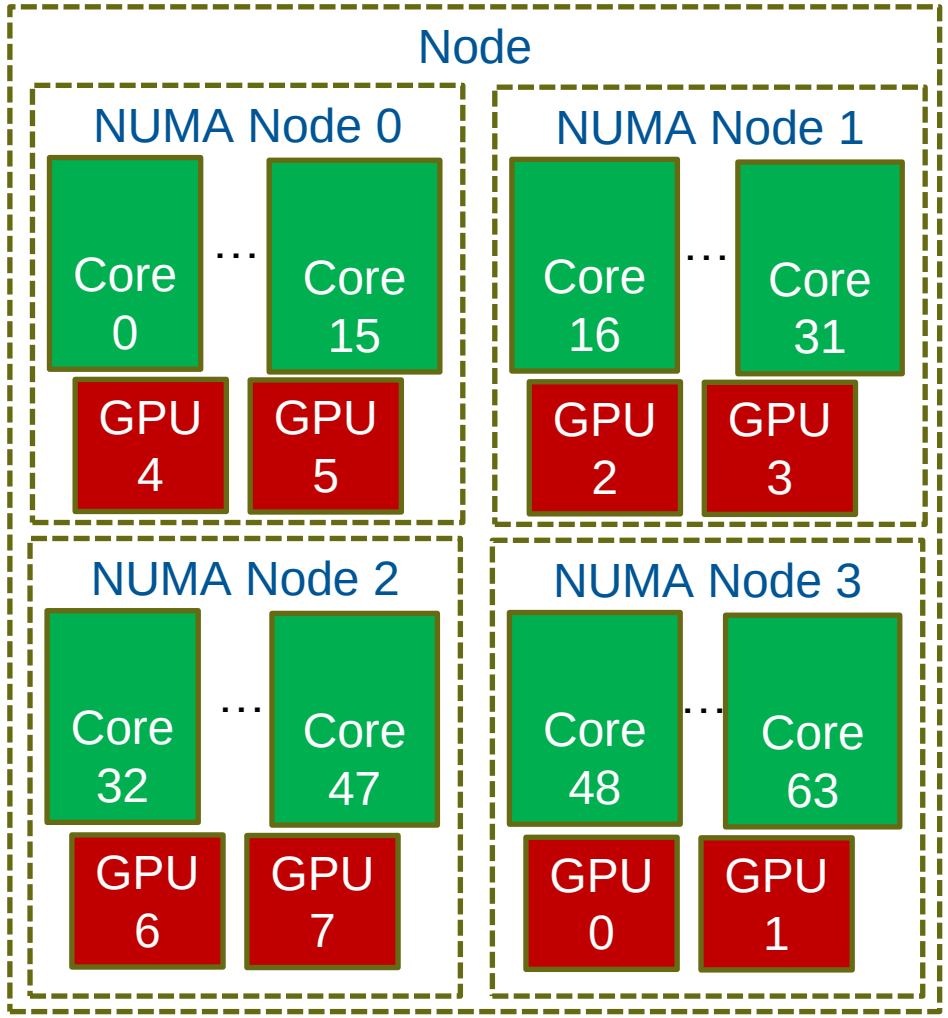


GPUs to NUMA Domains Mapping

- GPUs are associated to NUMA nodes
- Can use rocm-smi to see the topology

```
> srun --nodes=1 -p gpu -A edu24.summer -t "00:02:00" --ntasks=1 --gres=gpu:8 rocm-smi --showtopo
…
========================================= Numa Nodes =========================================
GPU[0] : (Topology) Numa Node: 3
GPU[0] : (Topology) Numa Affinity: 3
GPU[1] : (Topology) Numa Node: 3
GPU[1] : (Topology) Numa Affinity: 3
GPU[2] : (Topology) Numa Node: 1
GPU[2] : (Topology) Numa Affinity: 1
GPU[3] : (Topology) Numa Node: 1
GPU[3] : (Topology) Numa Affinity: 1
GPU[4] : (Topology) Numa Node: 0
GPU[4] : (Topology) Numa Affinity: 0
GPU[5] : (Topology) Numa Node: 0
GPU[5] : (Topology) Numa Affinity: 0
GPU[6] : (Topology) Numa Node: 2
GPU[6] : (Topology) Numa Affinity: 2
GPU[7] : (Topology) Numa Node: 2
GPU[7] : (Topology) Numa Affinity: 2
================================== End of ROCm SMI Log ==================================
```

GPUs to NUMA Domains Mapping



NUMA ID	0	0	1	1	2	2	3	3
Optimal GPU ID	4	5	2	3	6	7	0	1

GPU & NUMA Node Weights/Distance – affinity affects performance!

```
tdykes@login1:~> srun -N 1 -A edu24.summer -p gpu rocm-smi --showtopoweight
========================= ROCm System Management Interface =========================
============================= Weight between two GPUs =============================
       GPU0       GPU1       GPU2       GPU3       GPU4       GPU5       GPU6       GPU7
GPU0   0          15         15         30         30         30         15         30
GPU1   15         0          30         15         30         15         30         45
GPU2   15         30         0          15         15         30         30         30
GPU3   30         15         15         0          30         45         30         15
GPU4   30         30         15         30         0          15         15         30
GPU5   30         15         30         45         15         0          30         15
GPU6   15         30         30         30         15         30         0          15
GPU7   30         45         30         15         30         15         15         0
============================== End of ROCm SMI Log ==============================
```

```
tdykes@login1:~> srun -N 1 -A edu24.summer -p gpu numactl --hardware
available: 4 nodes (0-3)
...
node distances:
node   0   1   2   3
  0:  10  12  12  12
  1:  12  10  12  12
  2:  12  12  10  12
  3:  12  12  12  10
```

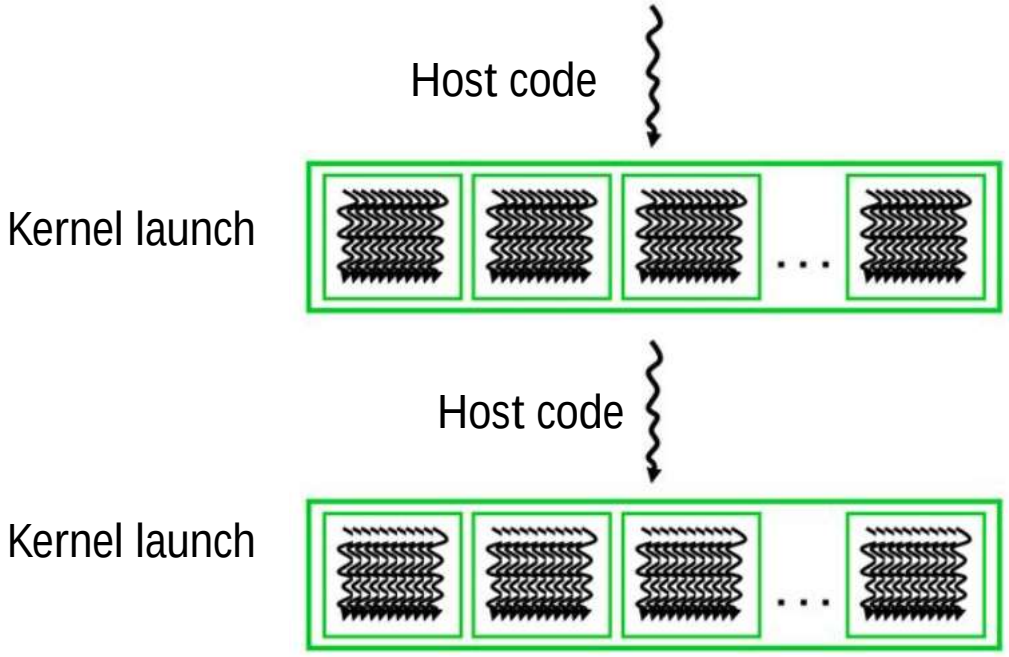
GPU Programming Overview

- So we now understand a bit more about the hardware we have…
  - The supercomputer as a whole
  - The compute blades holding the hardware
  - The architecture of the CPU and GPU
  - Connectivity and affinity between CPU and GPU

- Lets start thinking about how we might use them!

GPU Offload execution

- Code regions are offloaded from a host CPU to be computed on an accelerator
  - Host-device model
  - Kernel execution
- Keep latency sensitive and serial work on the CPU (host)
- Memory management traditionally via copies (host-to-device, device-to-host, device-to-device)



High Level Programming considerations

- GPU model - lots of small cores - requires many small tasks executing a kernel
  - Can replace iterations of a CPU loop with a GPU kernel call
    - Would it be better to port 10 iterations with 1M instructions each or 1M iterations with 10 instructions each?
- Need to adapt CPU code to run on the GPU
  - Programming patterns, eg. reduction
  - Decompose your problem in hierarchical thread/grid model
  - Manage data transfers between CPU and GPU memories
    - Data movement can easily become the bottleneck for performance
    - Try to reduce data movement, eg. reusing data
    - Hide latency by overlapping data movement with communications
  - Keep reusing data on the GPU to reach high an efficient use of the hardware (occupancy)
- Usually, after a first porting of the code to GPU, performance tuning is required to reach good performance

Programming Models for GPU: Different Approaches

Accelerated Libraries

- The ‘easiest’ solution, just link the library to your application without in-depth knowledge of GPU programming
- Many libraries are optimized by GPU vendors, eg. algebra libraries

Directive based methods

- Add acceleration to your existing code (C, C++, Fortran)
- Can reach good performance with relatively minimal code changes
- OpenACC, OpenMP

Programming Languages

- Maximum flexibility, requires in-depth knowledge of GPU programming and code rewriting (especially for Fortran)
- Kokkos, RAJA, CUDA, HIP, OpenCL, SYCL

Accelerated Libraries

- The easiest way to write GPU-accelerated code… is not to!
- The library based approach: you don't write the GPU code yourself, but use a library
- Some else has done the heavy lifting

- AMD provides several accelerated libraries as part of ROCM (see https://rocm.docs.amd.com/en/latest/reference/api-libraries.html)
  - Math (rocXXX, hipXXX): BLAS, FFT, SOLVER, SPARSE
    - Anything prefixed with `roc` (e.g. rocBLAS) is only for AMD GPUs
    - Anything prefixed with `hip` (e.g. hipBLAS) is portable
  - Communication: RCCL
    - RCCL (pronounced "Rickle") is a stand-alone library of standard collective communication routines for GPUs, implementing all-reduce, all-gather, reduce, broadcast, reduce-scatter, gather, scatter, and all-to-all
  - C++ primitives: hipCUB, hipTensor, rocPRIM, rocThrust
  - Random numbers: hipRAND, rocRAND
  - AI: MIOpen, MIVisionX, rocAL
- Many of these hidden behind python libraries too for tensorflow / pytorch and friends, more on that tomorrow

Directive based methods

- Another approach is… Let the compiler do it for you!

- The two most used directive approaches are:
  - OpenMP device model
  - OpenACC industry standard
  - These support Fortran natively

- The basic idea is to add pragmas/directives to a serial program to address GPU offload, typically addressing a loop nest.
  - OpenACC is known to be more descriptive
    - The programmer uses directives to tell the compiler where it can parallelize the code and whether it should manage data between potentially separate host and accelerator memories, but the compiler decides
  - OpenMP offloading approach, on the other hand, is known to be more prescriptive
    - The programmer uses directives to tell the compiler explicitly how/where to parallelize the code, instead of letting the compiler decide, with more explicit memory management

OpenMP/OpenACC offload example

- OpenACC (only Fortran support in CCE)

```
!$acc data copyin(B[0:n], C[0:n]) copyout(A[0:n])
!$acc parallel loop
do i = 1, n
  A(i) = B(i) + scalar * C(i)
end do
!$acc end parallel loop
!$acc end data
```

- It schedules the loop to be executed in parallel on the GPU
- Levels of parallelism are automatically decided and applied by the compiler to map the hardware resources

- OpenMP (C/C++/Fortran support in CCE)

```
#pragma omp target data map(to: B[0:n], C[0:n]) map(from: A[0:n])
{
  #pragma omp target \
              teams \
              distribute \
              parallel for simd
  for (size_t i = 0; i < n; i++) {
    A[i] = B[i] + scalar * C[i];
  }
}
```

- The `target` directive offloads the execution, no parallelization
- `teams` creates a league of teams and one master thread in each team, but no worksharing among the teams
- `distribute` distributes the iterations across the master threads in the teams, but no worksharing among the threads within one team
- `parallel do/for`: threads are activated within one team and worksharing among them
- `simd` enables SIMD instructions

Programming Languages

- Widest array of options
- Integrated into your language, usually via API but also language extensions

- Vendor provided APIs for device control and offload
  - AMD: HIP
    - Support of AMD and NVIDIA GPUs (wrappers around CUDA calls)
    - Integrated in LLVM, available as part of the ROCm https://github.com/ROCm/ROCm
  - Nvidia: CUDA
    - Nvidia proprietary software, it can run (*in principle*) only on Nvidia GPUs
    - NVCC compiler
  - Intel: oneAPI Level Zero
    - Provide direct-to-metal interfaces to offload accelerator devices (e.g. GPU and FPGA)
    - Part of the Intel oneAPI project (https://github.com/oneapi-src)
    - More low-level control than SYCL

Programming Languages

- Frameworks that abstract parallel execution from index space and memory location, based on C++:
  - RAJA: https://github.com/LLNL/RAJA
  - Kokkos: https://github.com/kokkos/kokkos
- Well supported
- They provide portability across multiple devices

- CPU code

```
for (int i = 0; i < N; ++i)
{
  y[i] = a * x[i] + y[i];
}
```

- RAJA

```
RAJA::RangeSegment it_space(0, N);
RAJA::forall< exec_policy >
(it_space, [=](int i)
{
  y[i] = a * x[i] + y[i];
}
```

- KOKKOS

```
Kokkos::parallel_for(N,
[=](const int i)
{
  y[i] = a * x[i] + y[i];
}
```

Programming Languages

- SYCL uses generic programming with templates and generic lambda functions to enable higher-level application software to be cleanly coded with optimized acceleration of kernel code across an extensive range of acceleration backend APIs
  - No language extension, no pragmas, no attribute
  - Single source, two compilation pass
  - CUDA (NVIDIA), ROCm (AMD), oneAPI (Intel)
  - https://www.khronos.org/sycl/

```
#include <CL/sycl.hpp>
#include <numeric>
namespace sycl = cl::sycl;
int main() {
  const int N{1729};
  sycl::queue Q{sycl::gpu_selector{}}; // specify a queue for the execution (select a GPU)
  int *A = sycl::malloc_shared<int>(N,Q); // shared host/device memory allocation
  Q.parallel_for(N, [=](sycl::item<1> id) { A[id] = id; }).wait();
  assert(std::accumulate(A, A+N, 0.) == N*(N-1)/2);
}
```

Programming Languages

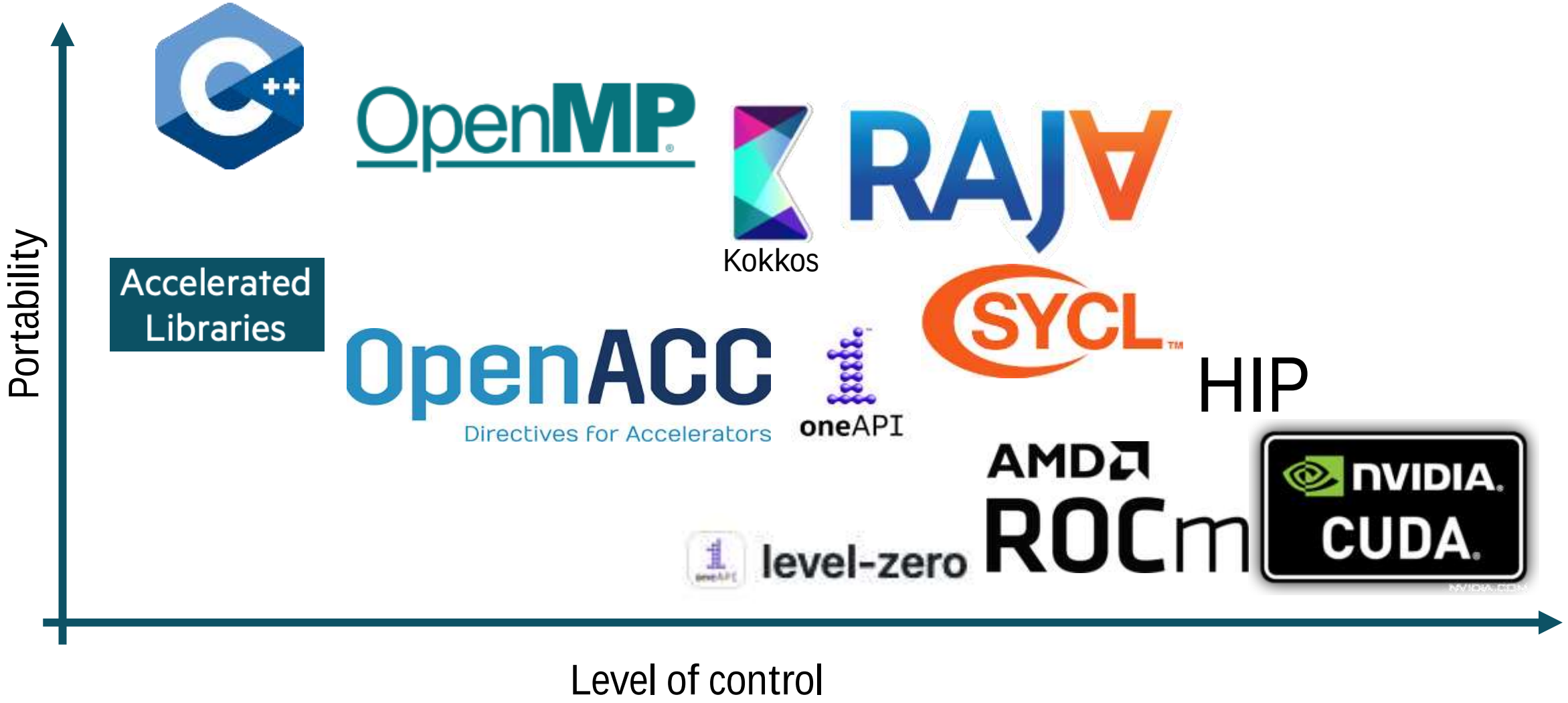
- With the C++ 17 standard, support for parallelism was introduced
  - The application developer specifies parallelism as the first parameter to a C++ algorithm:

```
std::transform(std::execution::par_unseq,
 x.begin(), x.end(), x.begin(), [](double elem_x) {
   return 5.0*elem_x;
 }
);
```

  - `std::execution::par_unseq` – Parallel multithreaded and vectorized execution: The various operations can be interleaved with each other on the same thread. Any given operation may start on a thread and end on a different thread

- With the release of ROCm 6.1, C++ standard parallelism is available for AMD GPUs
  - The previous code will run on the GPU if compiled with the flag --hipstdpar compile flag
- More info at https://rocm.blogs.amd.com/software-tools-optimization/hipstdpar/README.html

Evaluating Major GPU Programming Models



In summary...

- In this lecture we covered:
  - A very brief GPU history
  - A general overview of HPC GPU architectures
  - A zoom in on the architectures in use in Dardel
  - A general overview of GPU programming models

- The following lectures will delve further into
  - Programming GPUs: specifically AMD GPUs with HIP
  - Debugging and Profiling GPU code with rocgdb and rocprof
  - Porting code between languages / GPUs

- You'll hear more from us tomorrow!