

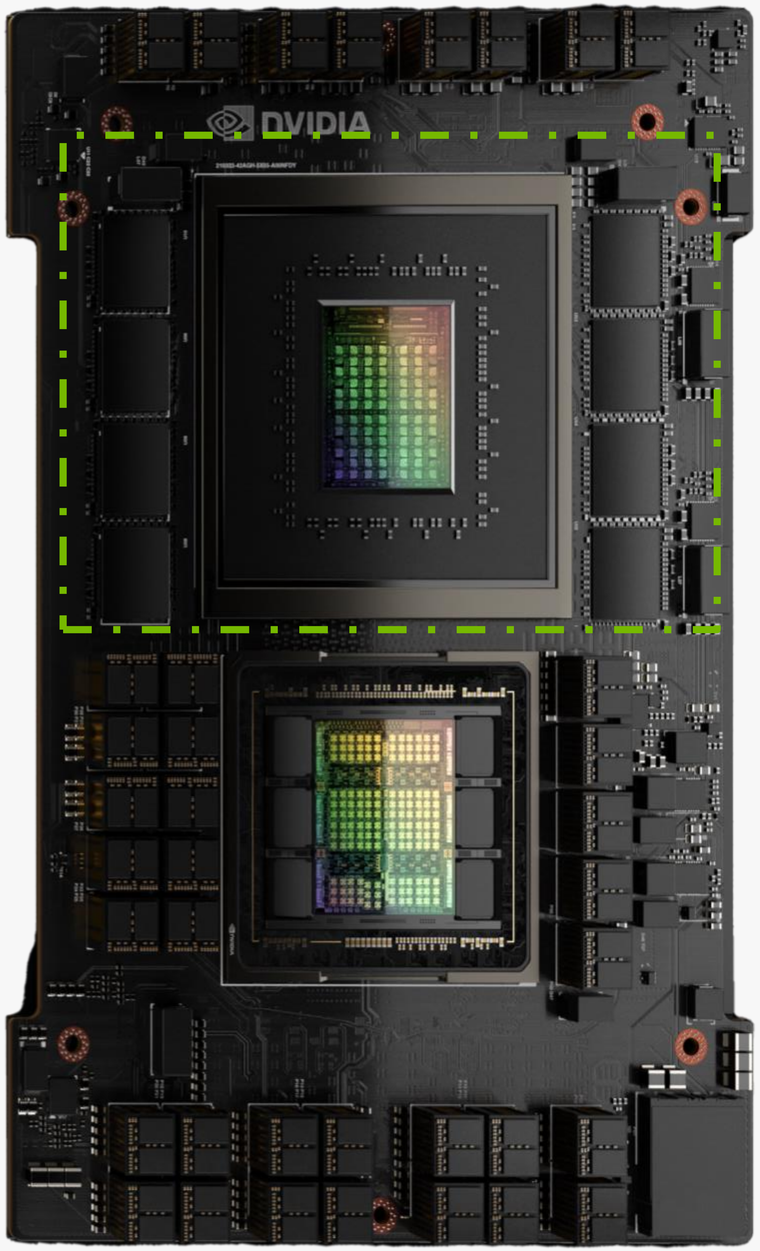
Accelerating Scientific Workflows with the NVIDIA Grace Hopper Platform

Mathias Wagner, Senior Developer Technology Engineer | SC23



Prelimiary Slides

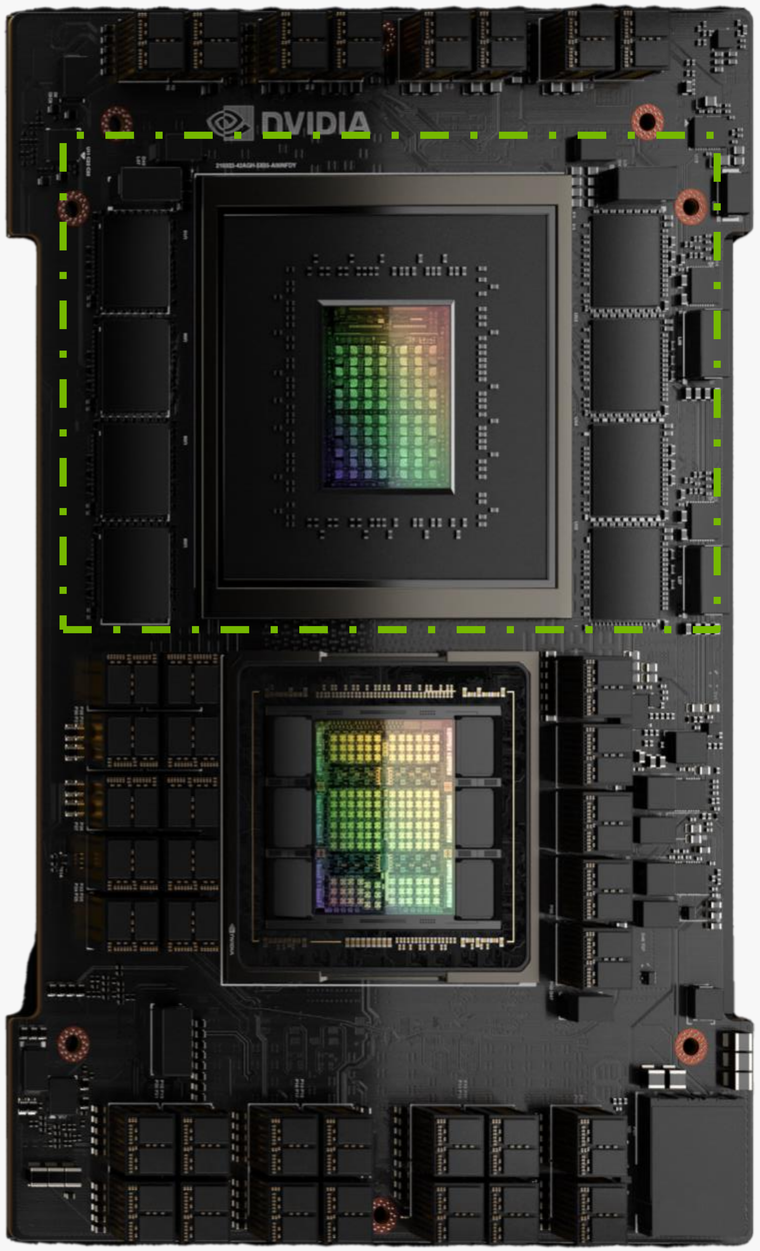
Mostly final content



NVIDIA Grace Hopper Superchip

"super" - more than a "chip"

NVIDIA CPU + NVIDIA GPU w/o compromises



NVIDIA Grace Hopper Superchip

"super" - more than a "chip"

NVIDIA CPU + NVIDIA GPU w/o compromises

- **NVIDIA Grace CPU**
  - 72 Arm-v9 Neoverse V2 CPU cores with SVE2.
    - → Throughput: 3.6 TFLOP/s
  - Memory:
    - → High capacity: ≤ 480 GB LPDDR5X
    - → High System Memory bandwidth: ≤ 500 GB/s



NVIDIA Grace Hopper Superchip

"super" - more than a "chip"

NVIDIA CPU + NVIDIA GPU w/o compromises

- **NVIDIA Grace CPU**
  - 72 Arm-v9 Neoverse V2 CPU cores with SVE2.
    - → Throughput: 3.6 TFLOP/s
  - Memory:
    - → High capacity: ≤ 480 GB LPDDR5X
    - → High System Memory bandwidth: ≤ 500 GB/s

- **NVIDIA Hopper GPU**
  - → High throughput: 60 TFLOP/s
  - Memory:
    - → Medium capacity: 96 GB HBM3
    - → Extreme bandwidth ≤ 4000 GB/s
  - ≤ 18x NVLink 4 → 900 GB/s
  - → Threads are threads



NVIDIA Grace Hopper Superchip

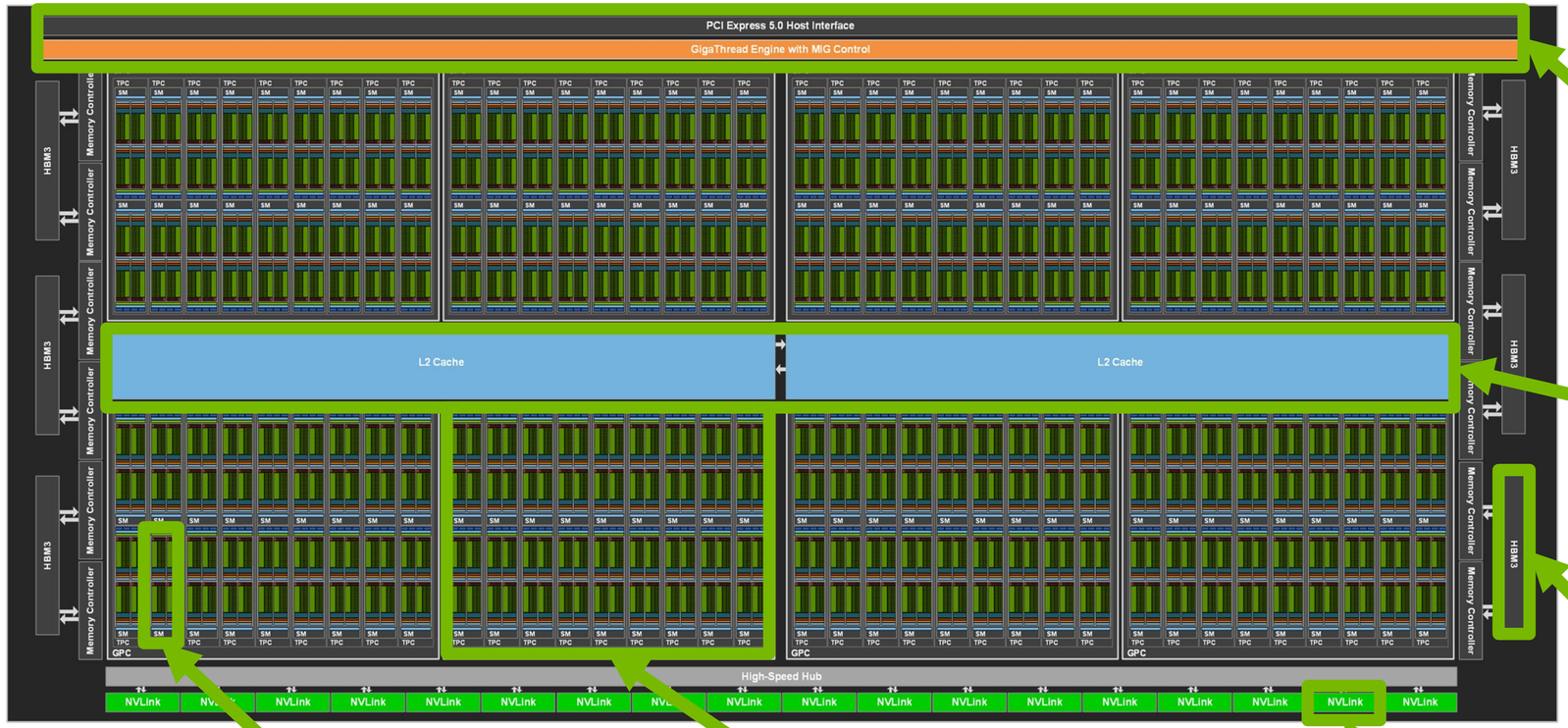
Soul is the new **NVLink-C2C** CPU ←→ GPU interconnect

- **Memory consistency**: ease of use
  - →*All* threads – GPU and CPU – access system memory: C++ `new`, `malloc`, `mmap`'ed files, atomics, …
  - → Fast automatic page migrations HBM3←→LPDDR5X.
  - → Threads cache peer memory → Less migrations.
- **High-bandwidth**: 900 GB/s (same as peer NVLink 4)
  - → GPU reads or writes local/peer LPDDR5X at ~peak BW
- **Low-latency**: GPU→HBM latency
  - →GPU reads or writes LPDDR5X at ~HBM3 latency

For all threads in the system
memory tastes like memory
expected behavior + latency + bandwidth.

Hopper Architecture

H100 GPU Key features



2nd Gen Multi-Instance GPU
Confidential Computing
PCIe Gen5

Larger 60 MB L2

96GB HBM3, 4 TB/s bandwidth

132 SMs
4th Gen Tensor Core

Thread Block Clusters

4th Gen NVLink
900 GB/s total bandwidth

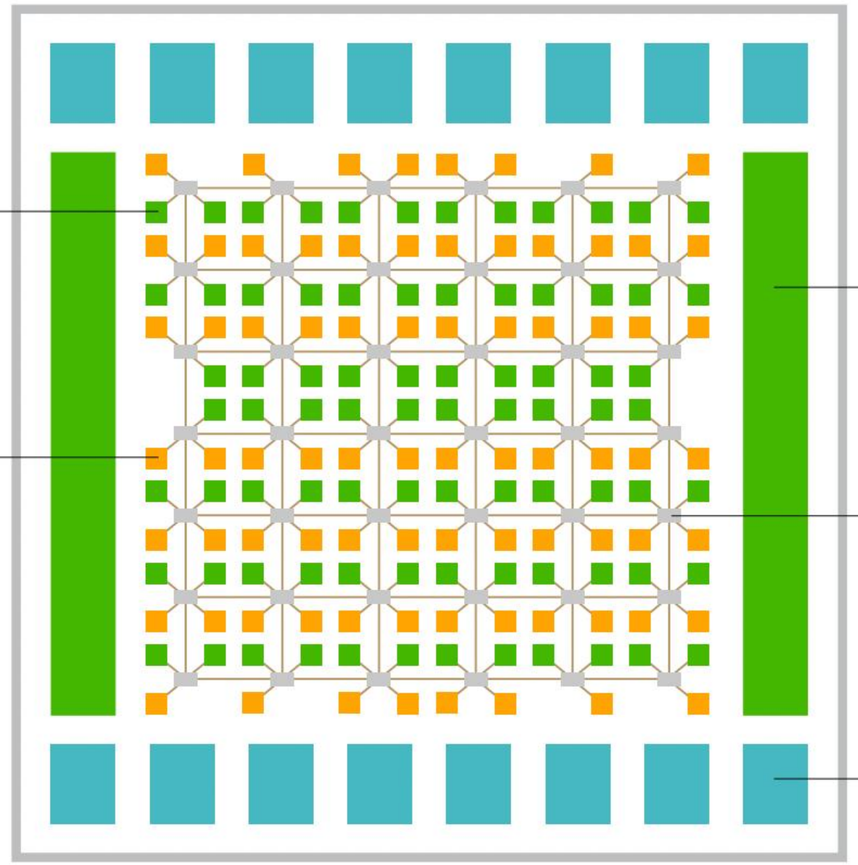
CUDA Programming Model for Hopper Architecture
Optimizing Applications for Hopper Architecture

GRACE Architecture

ARM Neoverse V2 with NVIDIA SCALABLE COHERENCY FABRIC

72 Neoverse V2 cores
Armv9
SVE2 with 4x128b

L1 per core:
64KB i-cache
64KB d-cache per core
L2 per core:
1 MB per core
L3 SCF117 MB



900 GB/s NVLink C2C
Up to 4x PCIe Gen5 x16

NVIDIA Scalable Coherency Fabric
3,225.6 GB/s Bi-section BW
117MB of L3 cache
Local caching of remote CPU/GPU memory
Supports up to 4 chip coherency over Coherent NVLINK

LPDDR5X 500 GB/s Bandwidth
240 GB per CPU

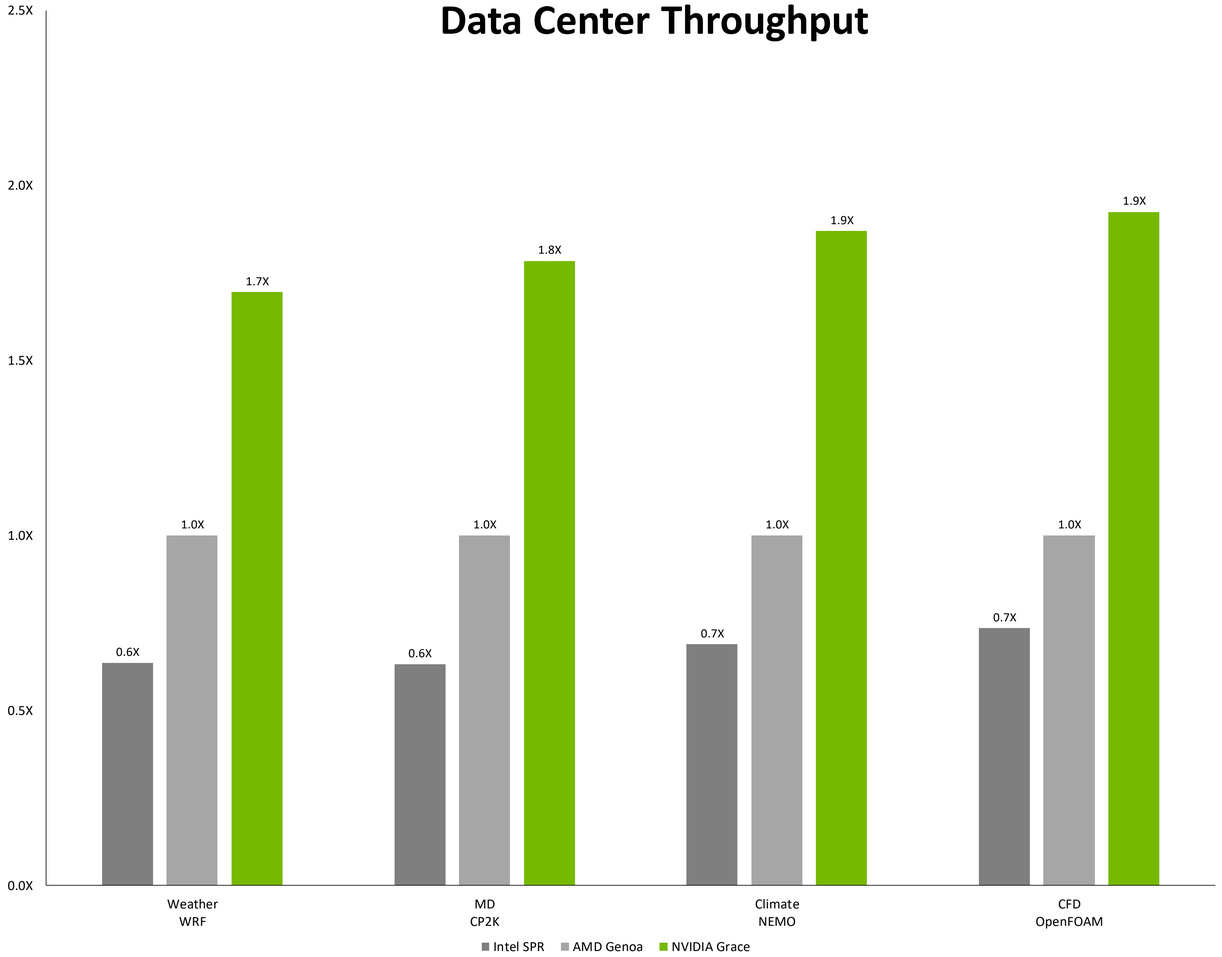
NVIDIA Grace CPU Delivers 1.9X HPC Data Center Throughput at the Same Power

Breakthrough Performance and Efficiency

Server Performance



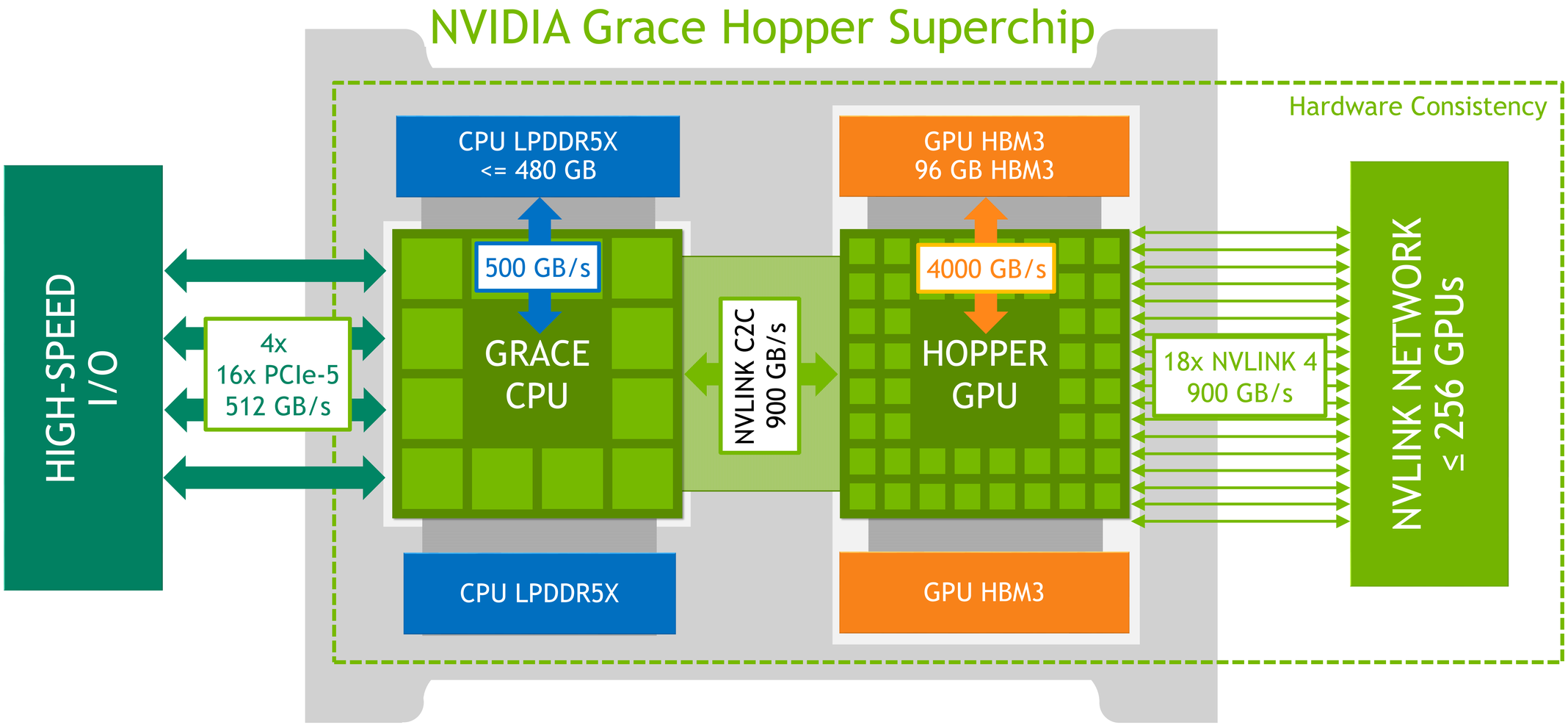
Data Center Throughput



Data Center level projection of NVIDIA Grace Superchip vs x86 flagship 2-socket data center systems (AMD Epyc 9654 and Intel Xeon 8480+). MD: CP2K RPA 2023.1 Climate: NEMO Gyre_Pisces v4.2.0 Weather: CONUS12, 24 hr simulation 4.4.2
CFD: OpenFOAM Motorbike | Large v2212
NVIDIA Grace Superchip performance based on engineering measurements. Results subject to change.

Grace Hopper Superchip

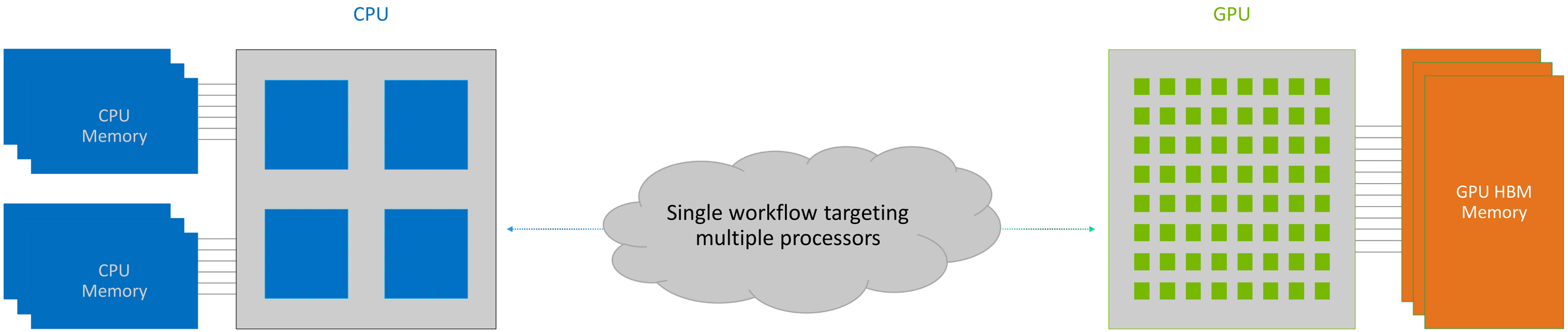
GPU can access CPU memory at CPU memory speeds



https://resources.nvidia.com/en-us-grace-cpu/nvidia-grace-hopper

Heterogeneous Computing

Combining processors of different types, each specializing in different types of execution

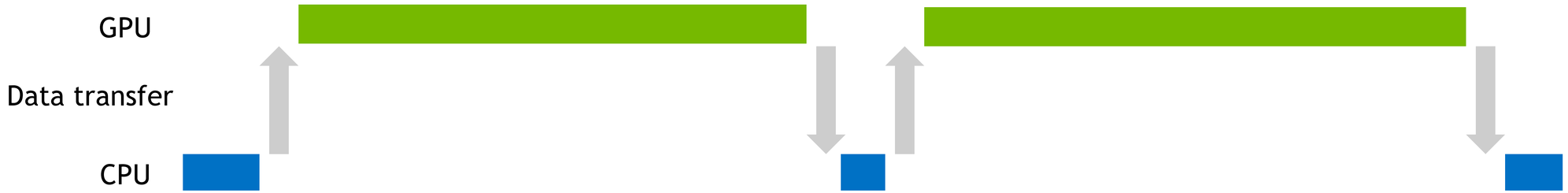


- Categorize applications into 3 categories
  - *Fully GPU Accelerated*
  - *Partially GPU Accelerated*
  - *Coherently GPU Accelerated*

Application on Accelerated Systems

Fully GPU Accelerated

- Compute almost fully on the GPU with data in GPU memory

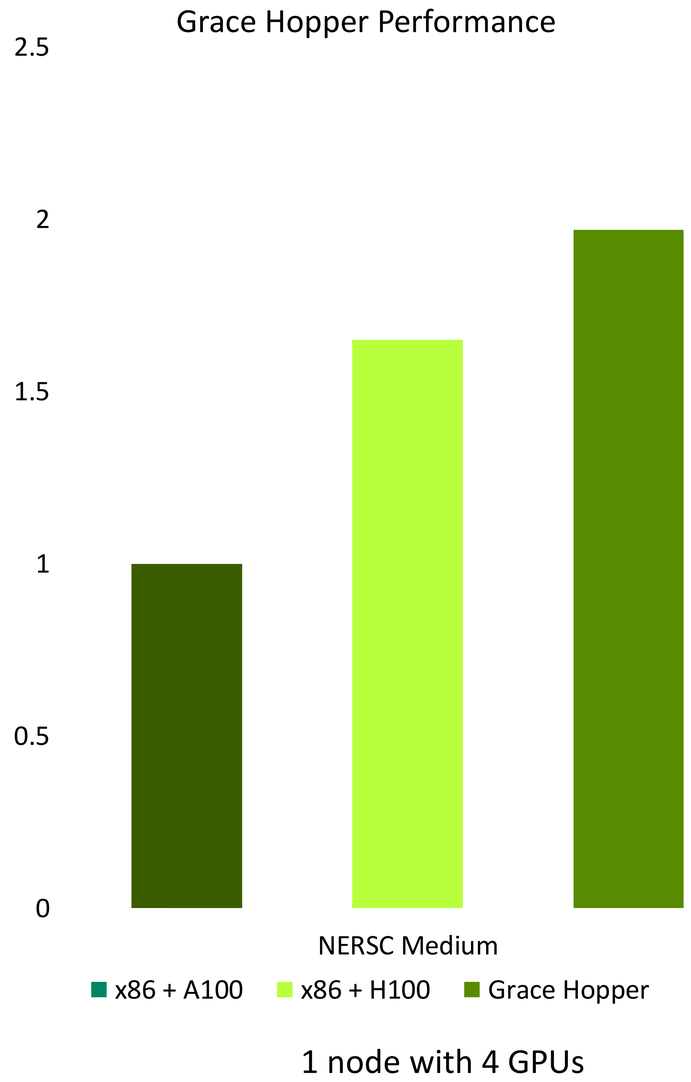


- Little to no limitation from CPU and data transfers

MILC

Fully accelerated using QUDA

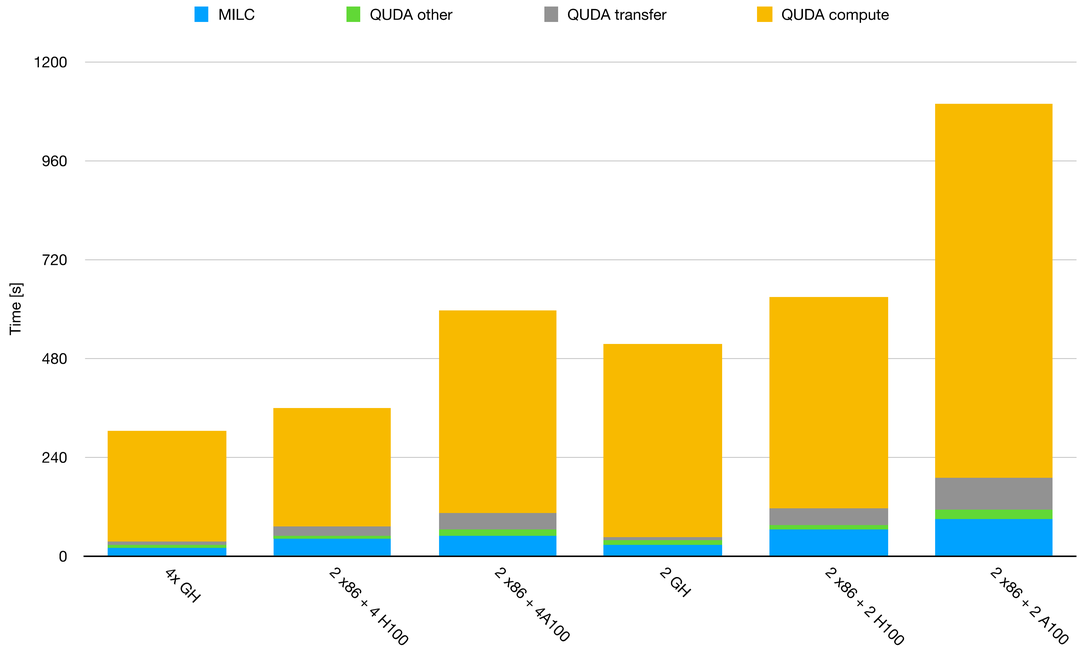
- Lattice Quantum Chromodynamics Application
  - Widely used by multiple research collaborations
  - GPU offload acceleration through QUDA with partial GPU data residency
  - Primarily bound by memory bandwidth
  - Strong-scaling required for time-to-solution
- NERSC Medium benchmark
- Performance on Grace-Hopper
  - C2C drastically reduces data-transfer time
  - Grace CPU memory bandwidth accelerates remaining CPU parts
  - Both combined restore scaling between generations

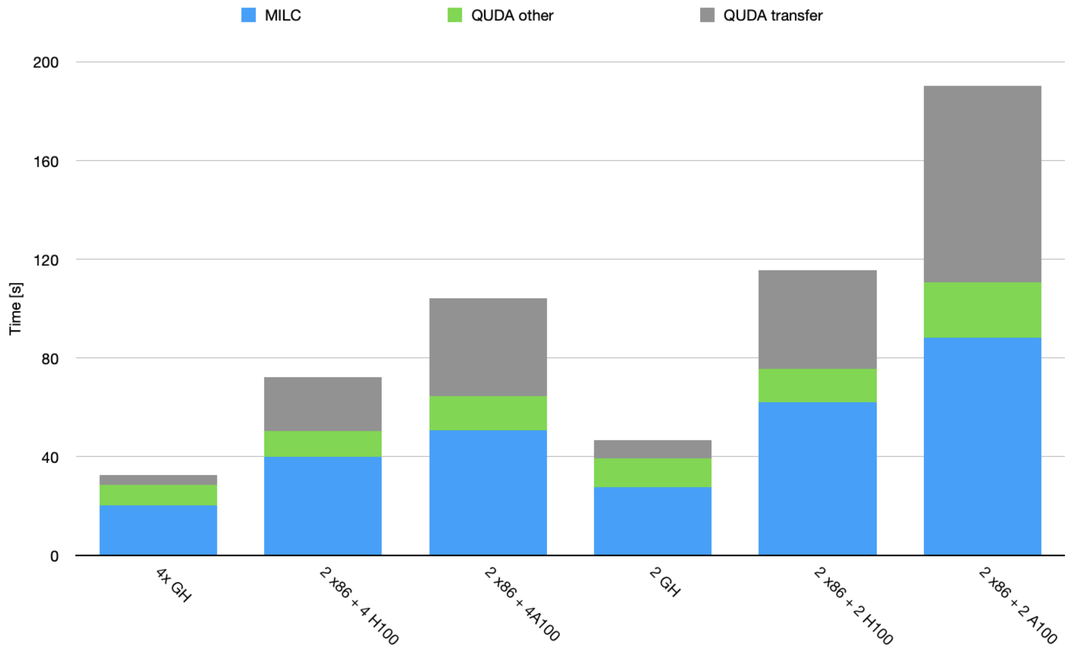


1 node with 4 GPUs

MILC

Breakdown





Application on Accelerated Systems

Partially GPU Accelerated

- As GPUs become faster applications become increasingly limited by non-GPU factors, e.g.
- mostly data transfer (PCIe) limited

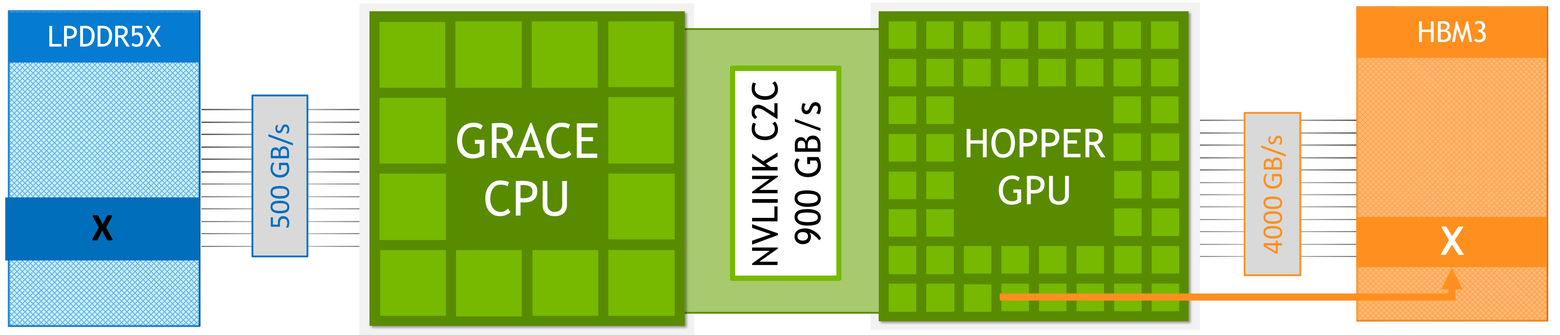


- mostly CPU limited

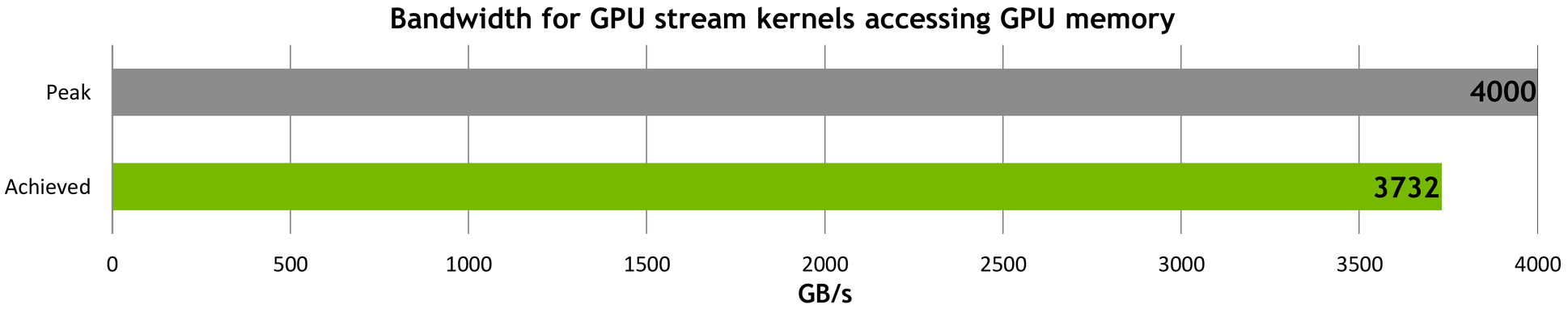


NVIDIA.

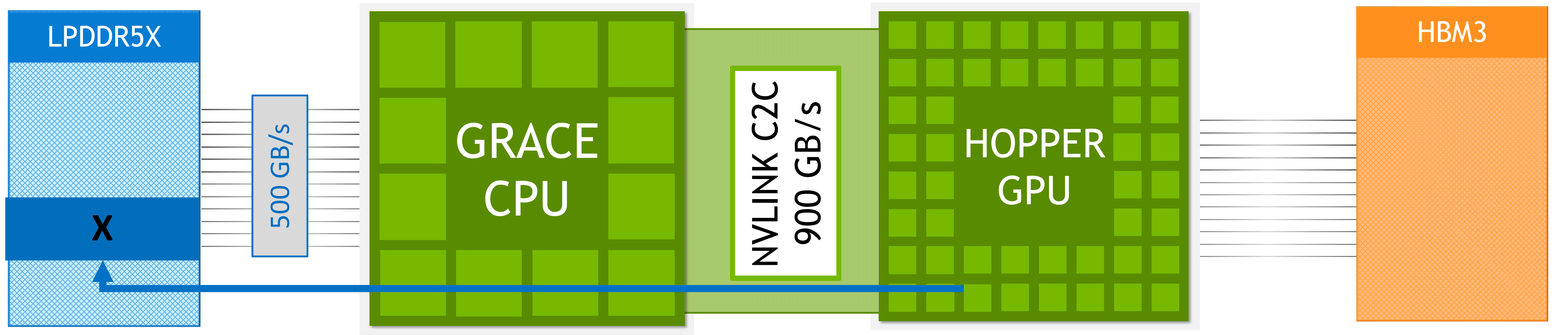
High Bandwidth Memory Access & Automatic Data Migration



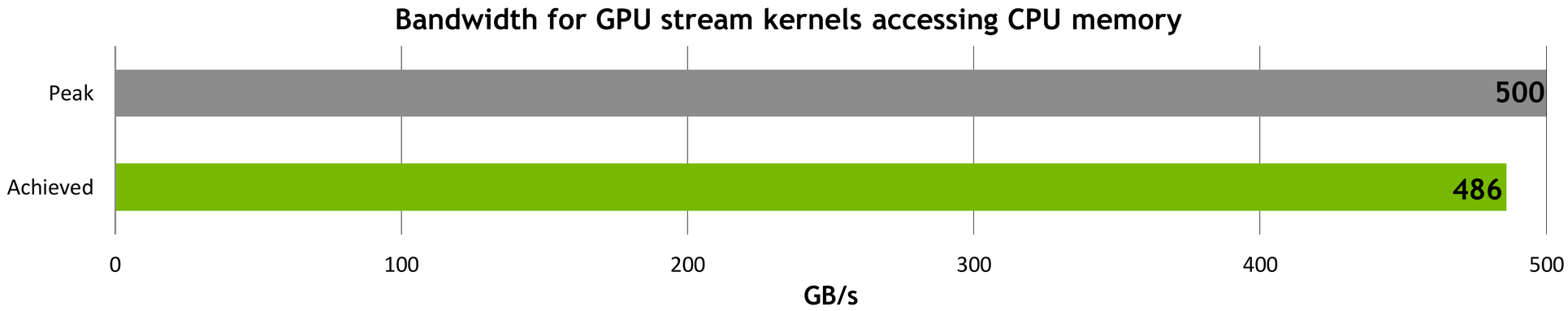
But Hopper can access its own memory
at **full HBM speed** of 4000 GB/sec



High Bandwidth Memory Access & Automatic Data Migration



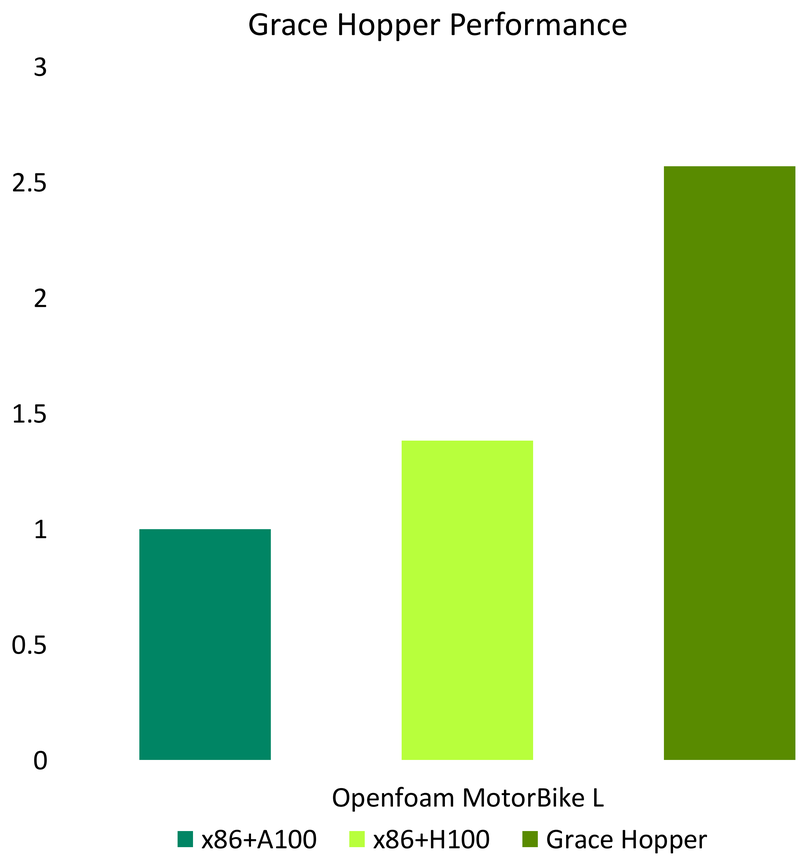
Hopper can access Grace memory
at **full CPU memory speed** of 500 GB/sec



OpenFoam

Partially GPU Accelerated – mostly CPU limited

- Computational fluid dynamics (CFD) toolbox developed by OpenCFD
  - Popular in automotive and other engineering sectors
  - Highly configurable fluid flow solvers with turbulence / heat transfer / etc.
  - Leverage GPU accelerated AMGX linear solvers
- HPC motorbike problem (Large)
  - Around 30% of CPU-only execution is spent in linear solves
- Performance on Grace Hopper
  - High CPU and GPU memory bandwidth improve compute performance
  - C2C bandwidth minimises the cost of migrating CPU matrix data

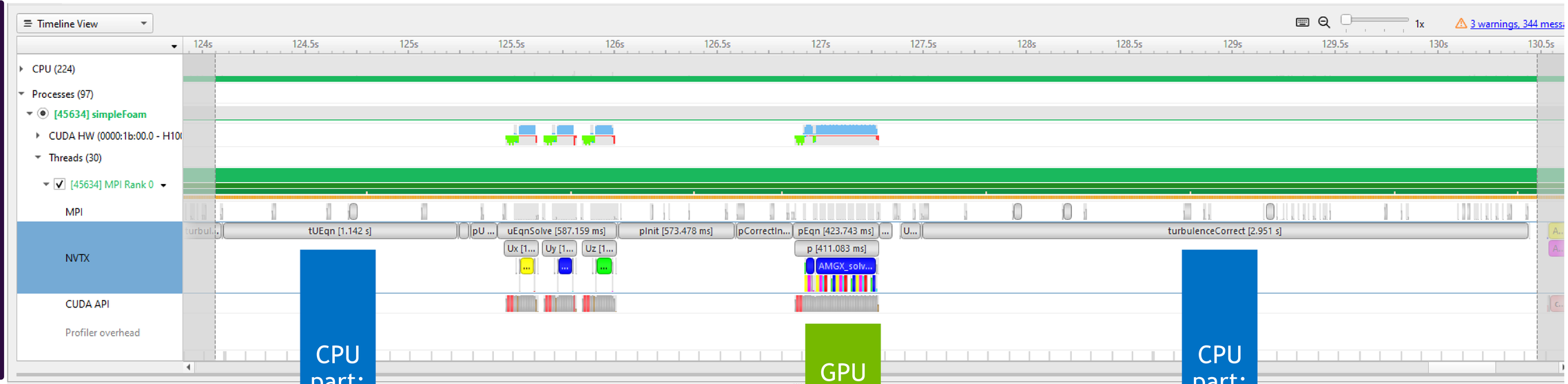


~35M cells benchmark designed by OpenFOAM HPC technical committee

OpenFoam

Nsight Systems Profile

x86 + H100 (H100 80GB HBM3)

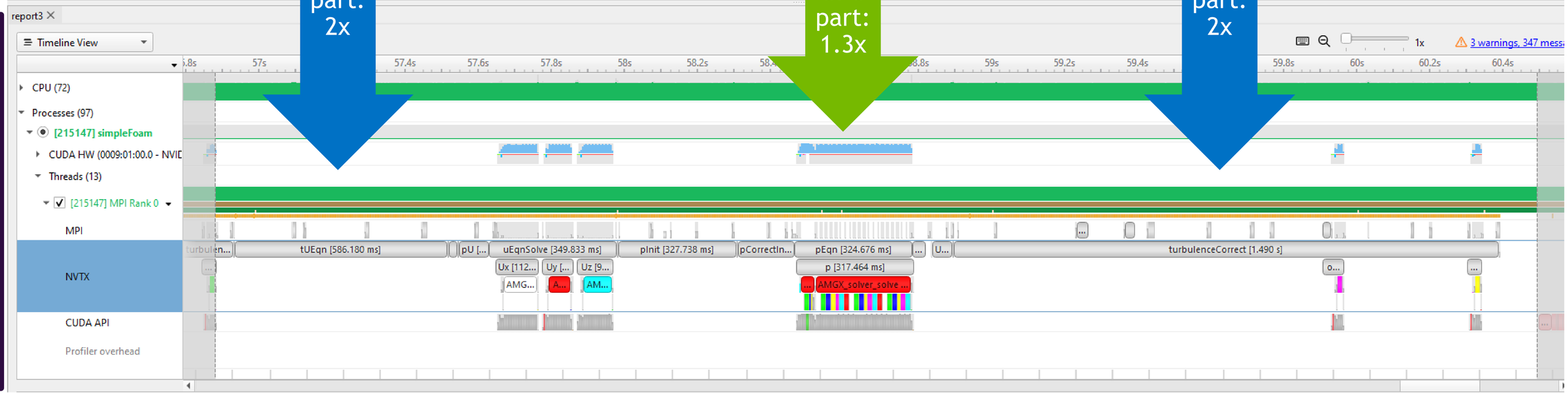


CPU part: 2x

GPU part: 1.3x

CPU part: 2x

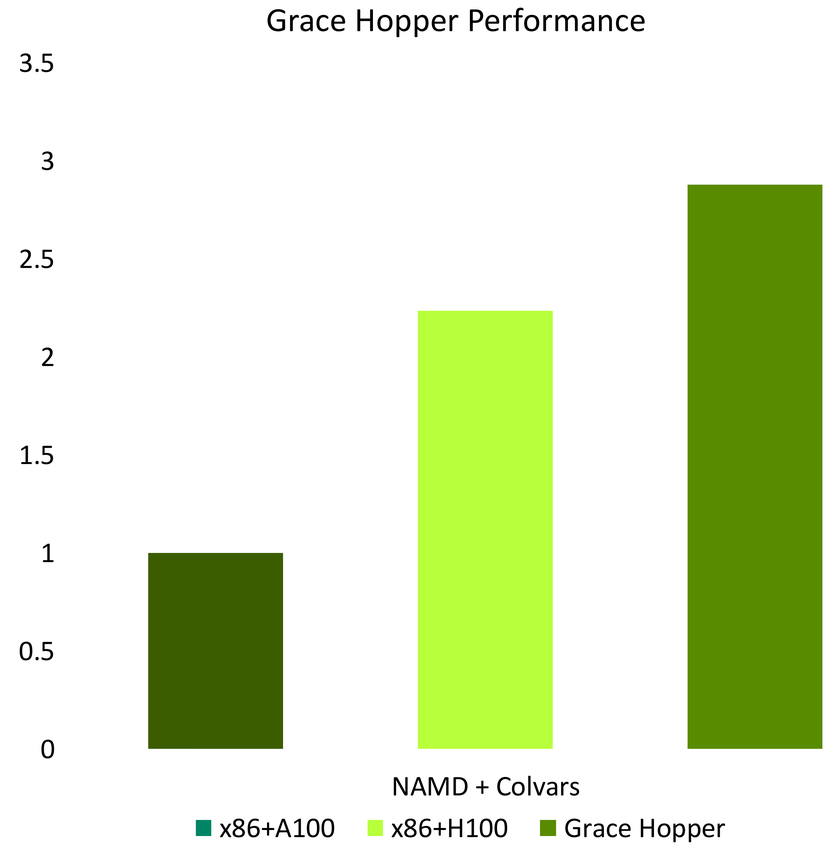
Grace Hopper (H100 96GB HBM3)



NAMD Collective Variables

Partially GPU Accelerated – mostly CPU limited

- Molecular Dynamics Simulation
  - Collective Variables provided by 3rd party Colvars[1] module (CPU-only)
    - Popular extension also used by e.g., Gromacs, LAMMPS
  - Colvars currently only compatible with NAMD's GPU-offload mode
- Glucose transporter 3 system (143k atoms)
  - Majority of forces are GPU accelerated
  - Additional Colvars force computed on CPU
  - Major bottleneck is integration step bound by CPU memory bandwidth
- Performance on Grace Hopper
  - Grace CPU memory bandwidth accelerates integration step
  - NVLink C2C bandwidth prevents data transfer bottleneck

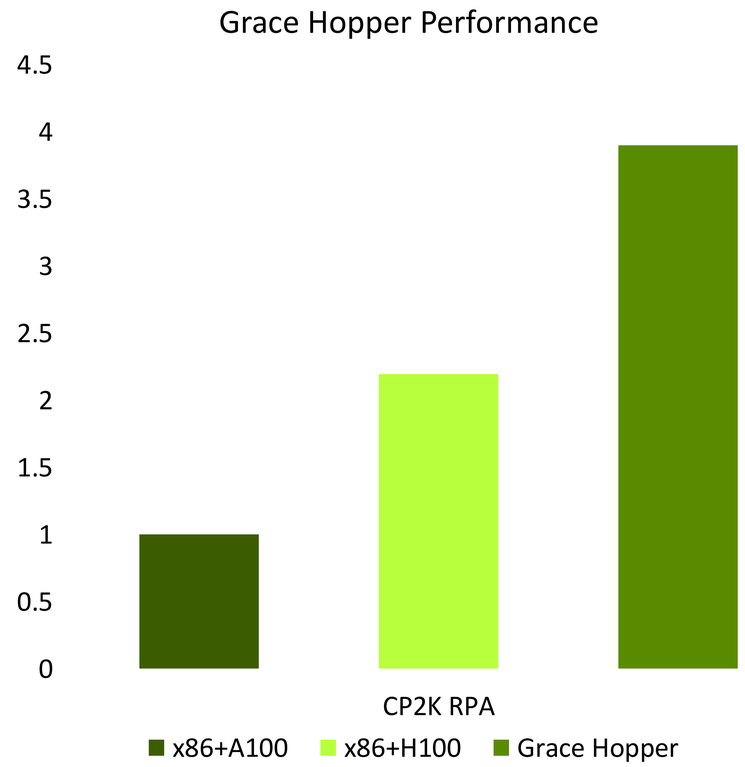


[1]) https://colvars.github.io/

CP2K

Partially GPU Accelerated – mostly data transfer limited

- Quantum Chemistry application
  - Implements a suite of different methods - many not yet GPU accelerated
  - Memory capacity constraints often require to keep some data in system memory
- Dataset "128-H2O" with random-phase approximation (RPA) method
  - PDGEMM dominates CPU runtime → use GPU accelerated PDGEMM
  - On x86 + current GPU: PDGEMM is hidden behind PCIe data transfer
    - Performance bound by CPU Memory Bandwidth, data transfer and MPI communication
- Performance on Grace Hopper
  - Grace CPU memory bandwidth and NVLink C2C greatly accelerates CP2k RPA
    - 6.2x faster than PCIe
  - C2C accelerated transfers hidden behind GPU-accelerated PDGEMM



Dataset "128-H2O" with random-phase approximation (RPA) method

Application on Accelerated Systems

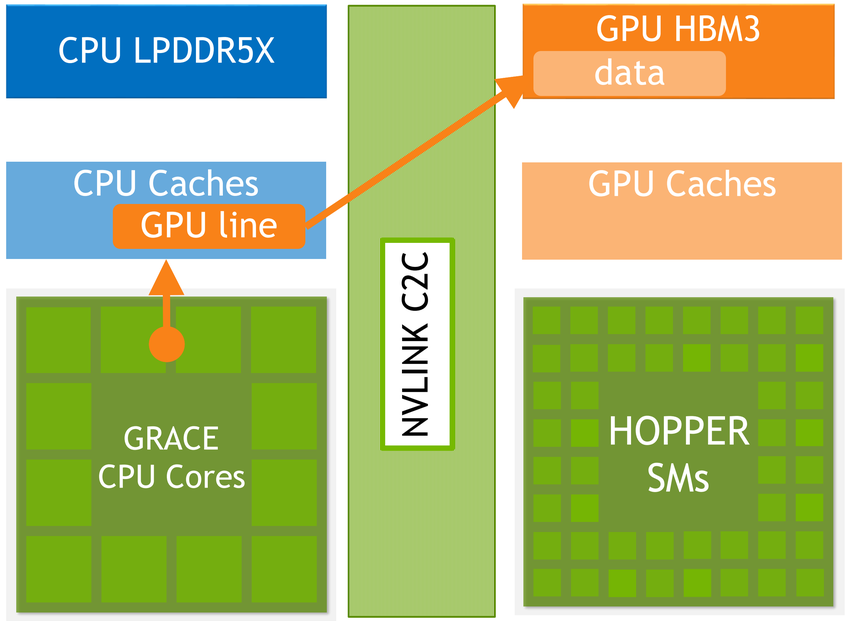
Coherently GPU Accelerated

- Exploit GPU / CPU coherency
- Use all available system features
  - not necessarily clean distinction between phases



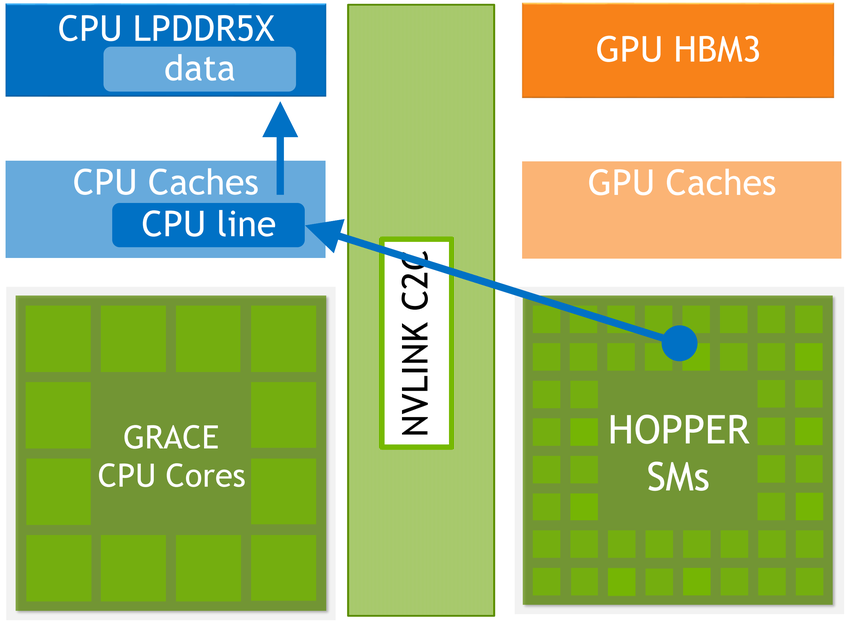
Global Access to All Data

Cache-coherent access via NVLink C2C from either processor to either physical memory



Grace directly reading Hopper's memory

CPU fetches GPU data into CPU L3 cache
Cache remains **coherent** with GPU memory
Changes to GPU memory **evict** cache line

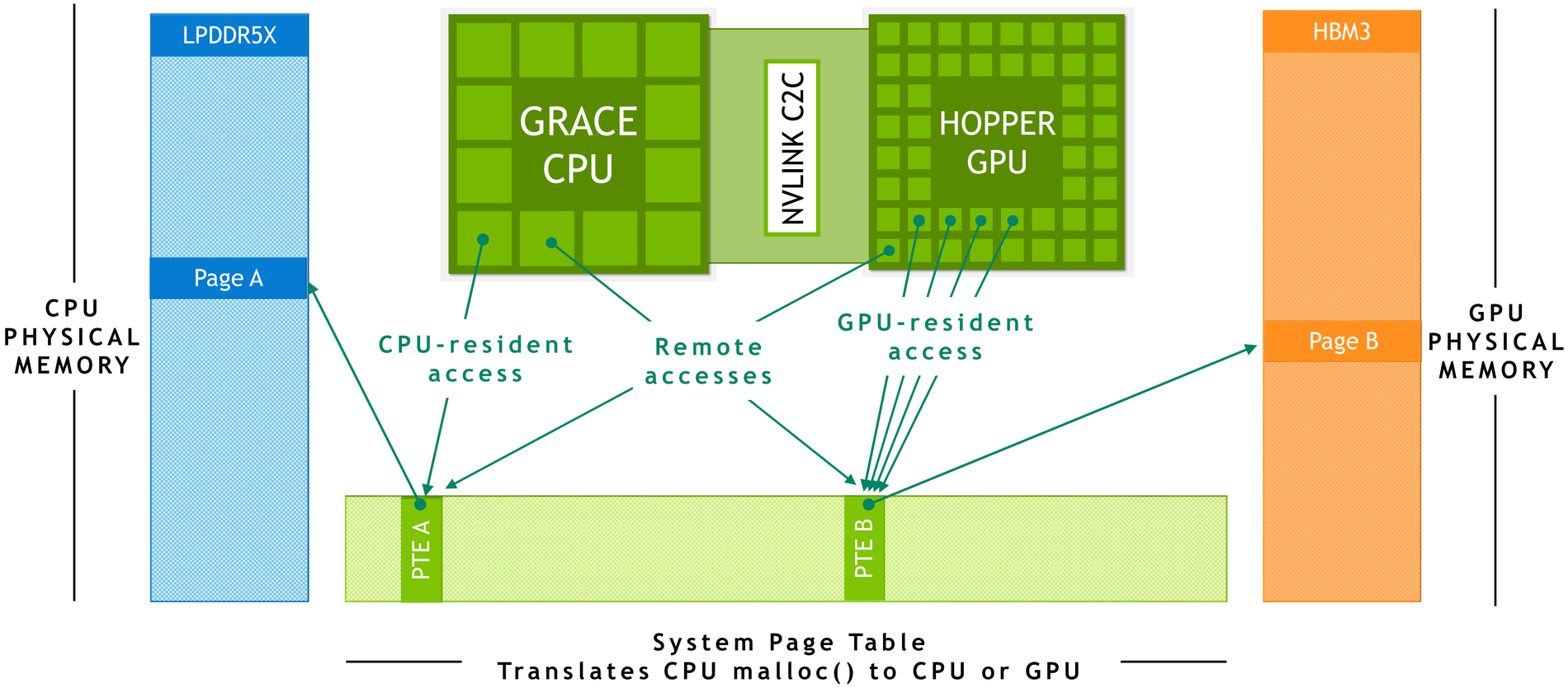


Hopper directly reading Grace's memory

GPU loads CPU data via CPU L3 cache
CPU and GPU **can both hit** on cached data
Changes to CPU memory **update** cache line

Grace Hopper

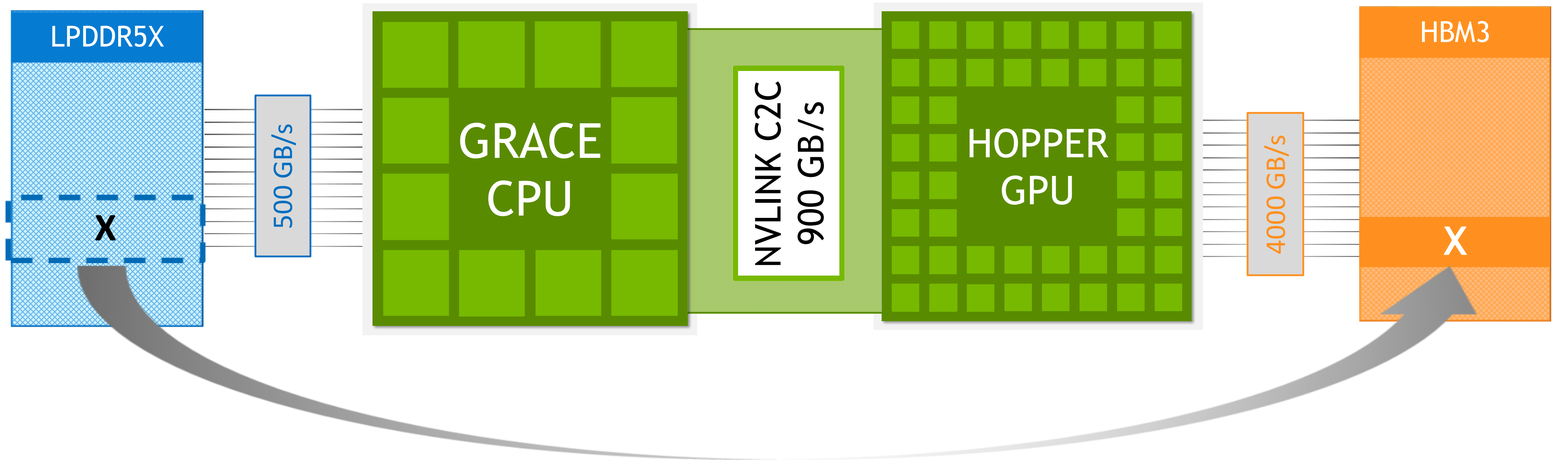
Address Translation Service (ATS) enables full access to all CPU & GPU allocations
Migrations are not required: Fewer Migrations



ATS creates a single page table for the whole system

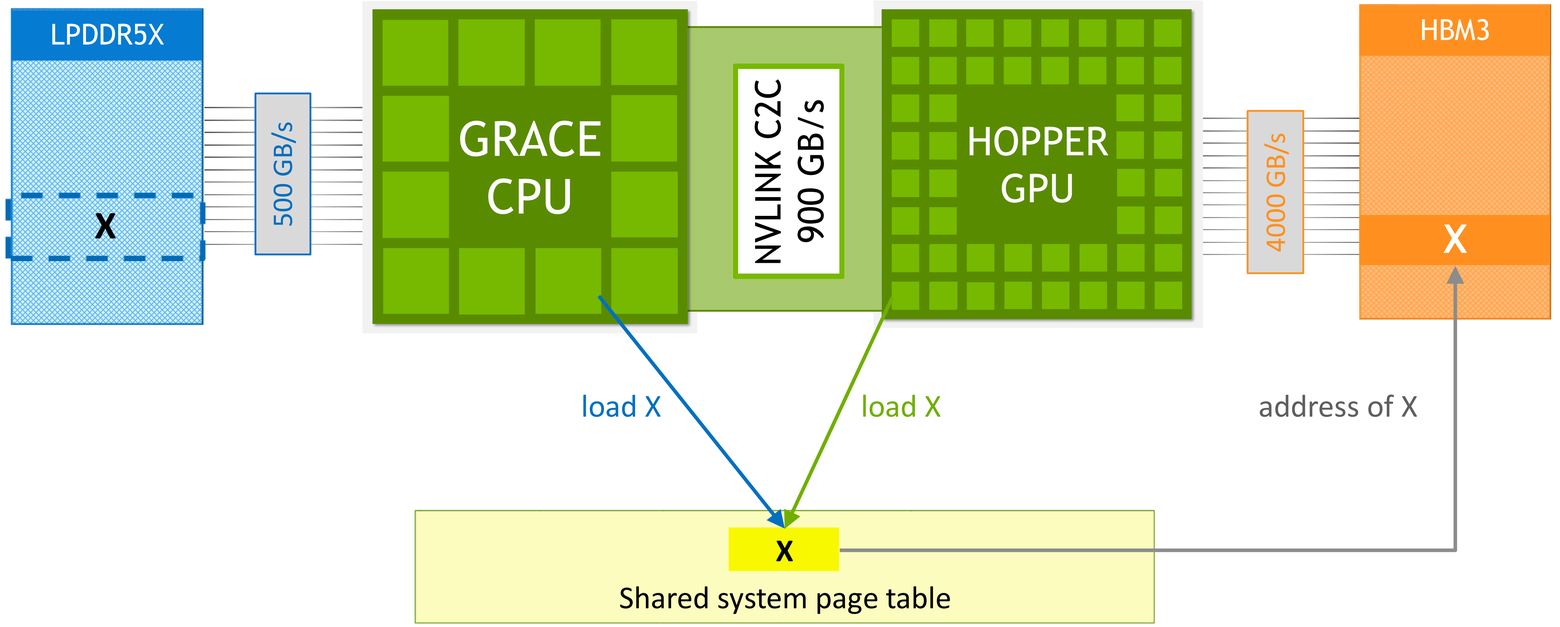
NVLink C2C allows access to all physical memory **without migration**

High Bandwidth Memory Access & Automatic Data Migration



The system can **automatically** migrate
both managed and CPU-allocated memory
in order to optimize access speed

High Bandwidth Memory Access & Automatic Data Migration

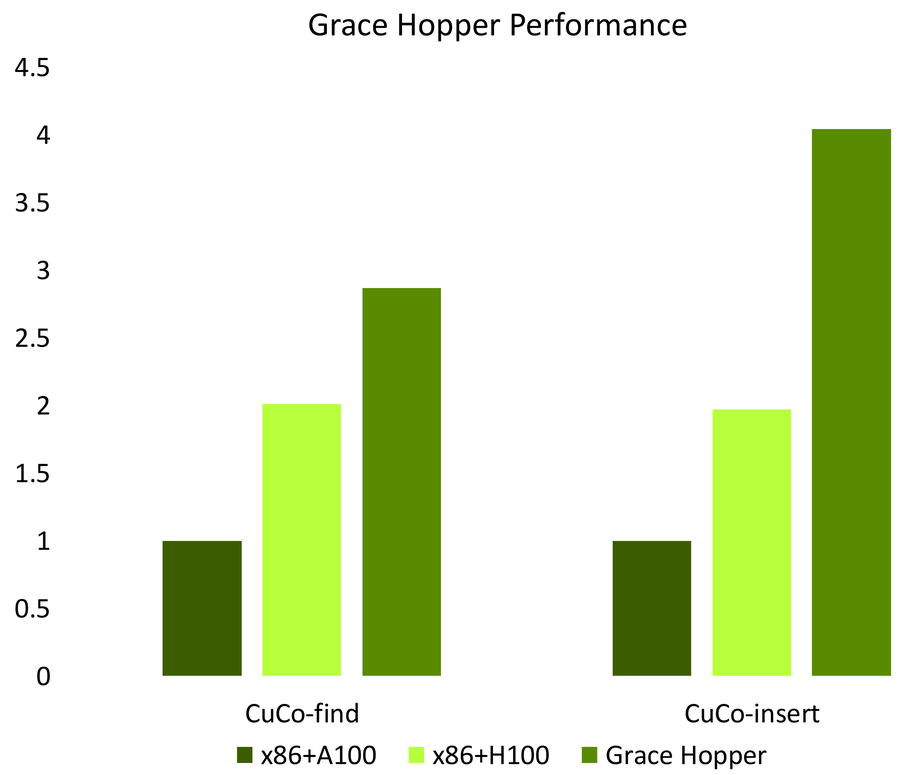


ATS shared page table means that both CPU and GPU automatically access X in its new location after migration

Applications using CPU/GPU coherency

cuCollections Hash Map benchmarks

- cuCollections enables you to build and query hash tables at the speed of your CPU GPU link bandwidth when the data is not located in GPU memory.
- Building Hash Table on GPU with dataset resident on CPU.
  - Convenient as data does not need to be moved explicitly
- Coherent access from GPU to CPU Memory.



Applications using CPU/GPU coherency

Performance regardless of memory location

Page: Details | Result: 5 - 943 - insert | Add Baseline | Apply Rules | Occupancy C

	Result	Time	Cycles	Regs	GPU	SM Fre
Data in CPU Memory	943 - insert (4194304, 1, 1)x(128, 1, 1)	27.36 msecond	43,640,489	26	0 - H100 80GB HBM3	1.59 cyc
Data in GPU Memory	897 - insert (4194304, 1, 1)x(128, 1, 1)	12.29 msecond	19,529,593	26	0 - H100 80GB HBM3	1.59 cyc

The report contains imported source files.

GPU Speed Of Light Throughput

High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved p
Compute and Memory to clearly identify the highest contributor. High-level overview of the utilization for compute and memory resources

Compute (SM) Throughput [%]	19.31 (-55.31%)
Memory Throughput [%]	17.49 (-58.06%)
L1/TEX Cache Throughput [%]	9.54 (-56.37%)
L2 Cache Throughput [%]	22.91 (-54.11%)
DRAM Throughput [%]	17.49 (-58.06%)

Page: Details | Result: 5 - 599 - insert | Add Baseline | Apply Rules | Occupancy Calculator

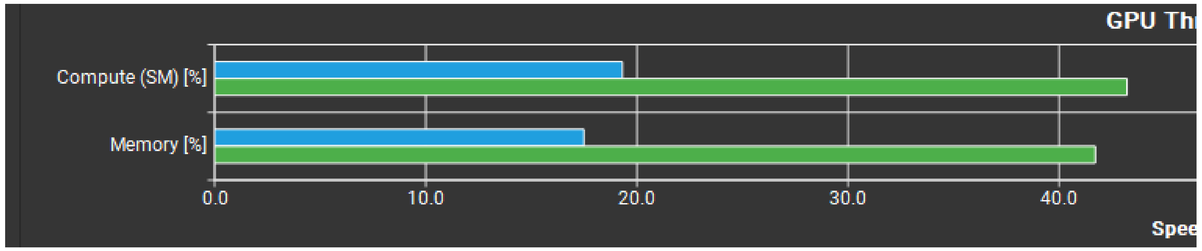
	Result	Time	Cycles	Regs	GPU	SM Frequency	CC	Proces
Data in CPU Memory	nsert (4194304, 1, 1)x(128, 1, 1)	10.22 msecond	16,243,946	26	0 - H100 96GB HBM3	1.59 cycle/nsecond	9.0	[51220]
Data in GPU Memory	nsert (4194304, 1, 1)x(128, 1, 1)	10.38 msecond	16,489,461	26	0 - H100 96GB HBM3	1.59 cycle/nsecond	9.0	[51220]

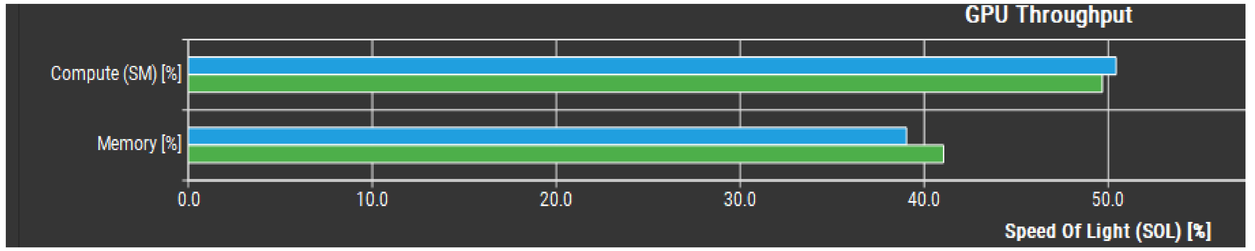
GPU Speed Of Light Throughput

High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with
Compute and Memory to clearly identify the highest contributor. High-level overview of the utilization for compute and memory resources of the GPU presented as a r

Compute (SM) Throughput [%]	50.43 (+1.51%)	Duration [msecond]
Memory Throughput [%]	39.03 (-4.93%)	Elapsed Cycles [cycle]
L1/TEX Cache Throughput [%]	25.88 (+1.71%)	SM Active Cycles [cycle]
L2 Cache Throughput [%]	49.86 (+1.06%)	SM Frequency [cycle/nsec
DRAM Throughput [%]	39.03 (-4.93%)	DRAM Frequency [cycle/ns

2.7 x





NVIDIA.

Heterogenous Programming Example

Works on both x86 and Grace Hopper

cudaMalloc continues to behave as x86 platform and cannot be accessed on CPU.

```
__global__ void kernel(int *data)
{
    data[threadIdx.x] += 2;
}
int main()
{
    int N = 128;
    int *data = (int *)malloc(N * sizeof(int));
    int *d_data;

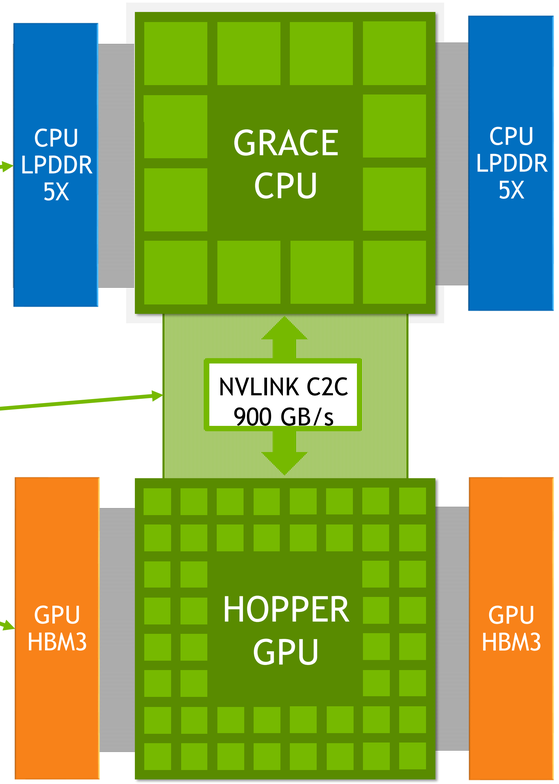
    cudaMalloc((void **)&d_data, N * sizeof(int));

    cudaMemcpy(d_data, data, N * sizeof(int), cudaMemcpyHostToDevice);

    kernel<<<1, N>>>(d_data);

    cudaMemcpy(d_data, data, N * sizeof(int), cudaMemcpyDeviceToHost);
    cudaDeviceSynchronize();

    cudaFree(d_data);
    free(data);
}
```



CUDA Unified Memory or Managed Memory

Portable codes across x86 and Grace Hopper by direct access or moving memory pages

```
__global__ void kernel(int *data)
{
    data[threadIdx.x] += 2;
}
int main()
{
    int N = 128;

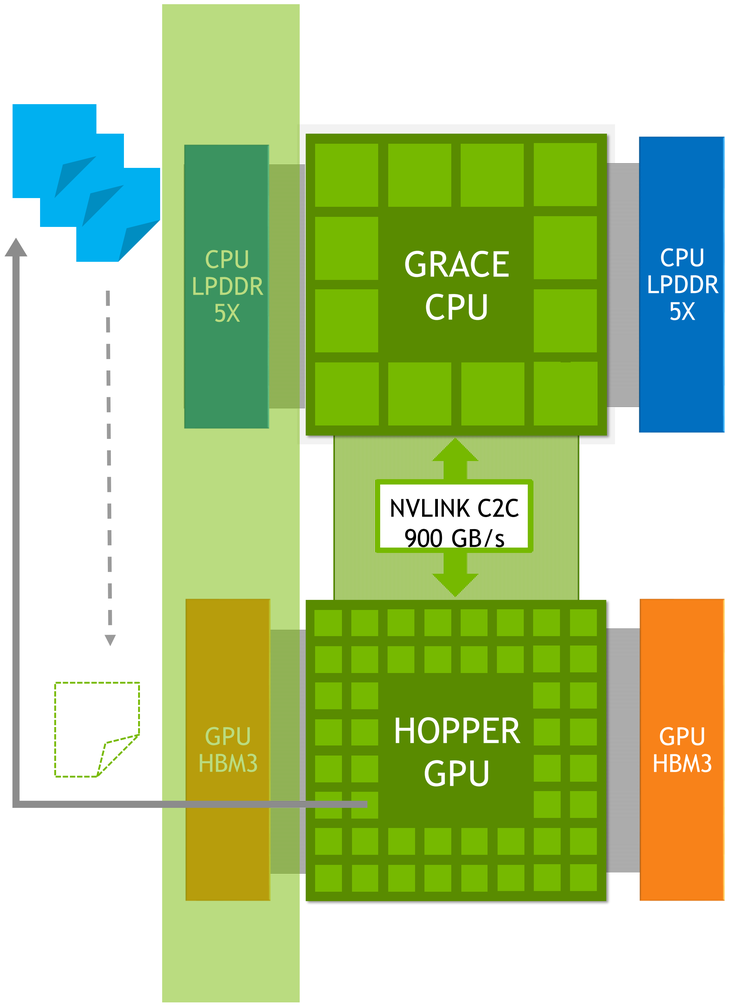
    int *data;
    cudaMallocManaged((void **)&data, N * sizeof(int));

    for (int i = 0; i < N; i++)
    {
        data[i] = i;
    }

    kernel<<<1, N>>>(data);

    cudaDeviceSynchronize();

    cudaFree(data);
}
```



System allocator

Easy to move codes to GPU. Uses Address Translation Service (ATS)

```
__global__ void kernel(int *data1, int *data2)
{
  data1[threadIdx.x] = threadIdx.x;
  data2[threadIdx.x] = threadIdx.x;
}

int main()
{
  int *data = (int *)malloc(128 * sizeof(int));

  int data2[128];

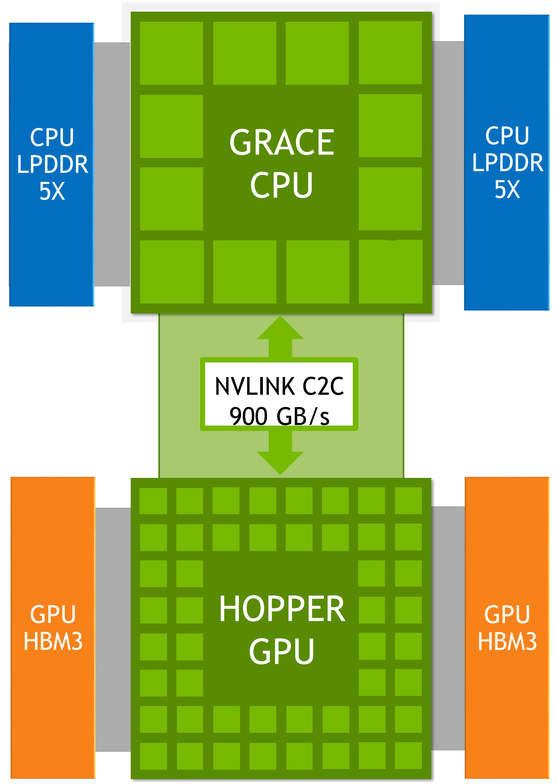
  kernel<<<1, 128>>>(data, data2);
  cudaDeviceSynchronize();

  for (int i = 0; i < 128; i++)
  {
    printf("(%d, %d)  ", data[i], data2[i]);
  }

  free(data);
  return 0;
}
```

Physical pages are allocated on first touch





System allocator

Easy to move codes to GPU. Uses Address Translation Service (ATS)

```
__global__ void kernel(int *data1, int *data2)
{
  data1[threadIdx.x] = threadIdx.x;
  data2[threadIdx.x] = threadIdx.x;
}

int main()
{
  int *data = (int *)malloc(128 * sizeof(int));

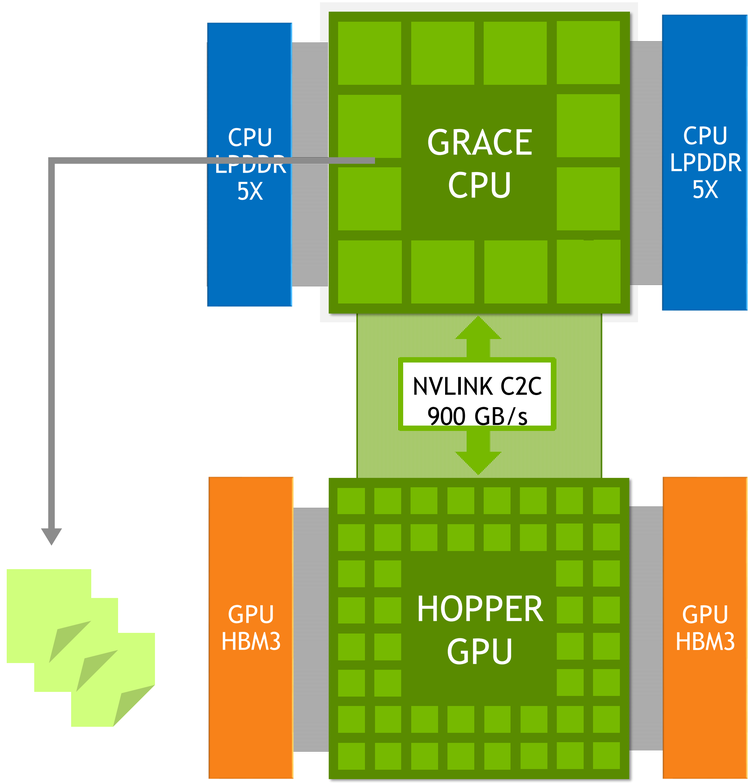
  int data2[128];

  kernel<<<1, 128>>>(data, data2);
  cudaDeviceSynchronize();

  for (int i = 0; i < 128; i++)
  {
    printf("(%d, %d)  ", data[i], data2[i]);
  }

  free(data);
  return 0;
}
```

Direct access from CPU to GPU data



Programming Models for Grace Hopper

Unmatched Developer Flexibility

Accelerated Standard Languages	Incremental Optimization	Platform Specialization
C++ python™ Fortran	OpenACC More Science, Less Programming OpenMP	NVIDIA CUDA C++ | Fortran | Python

Performance Libraries

Core	Math	Communication	Data Analytics	AI	DSLs

GPU	CPU	Interconnect

There is no one programming model for Grace Hopper, developers can choose what works best for their application.

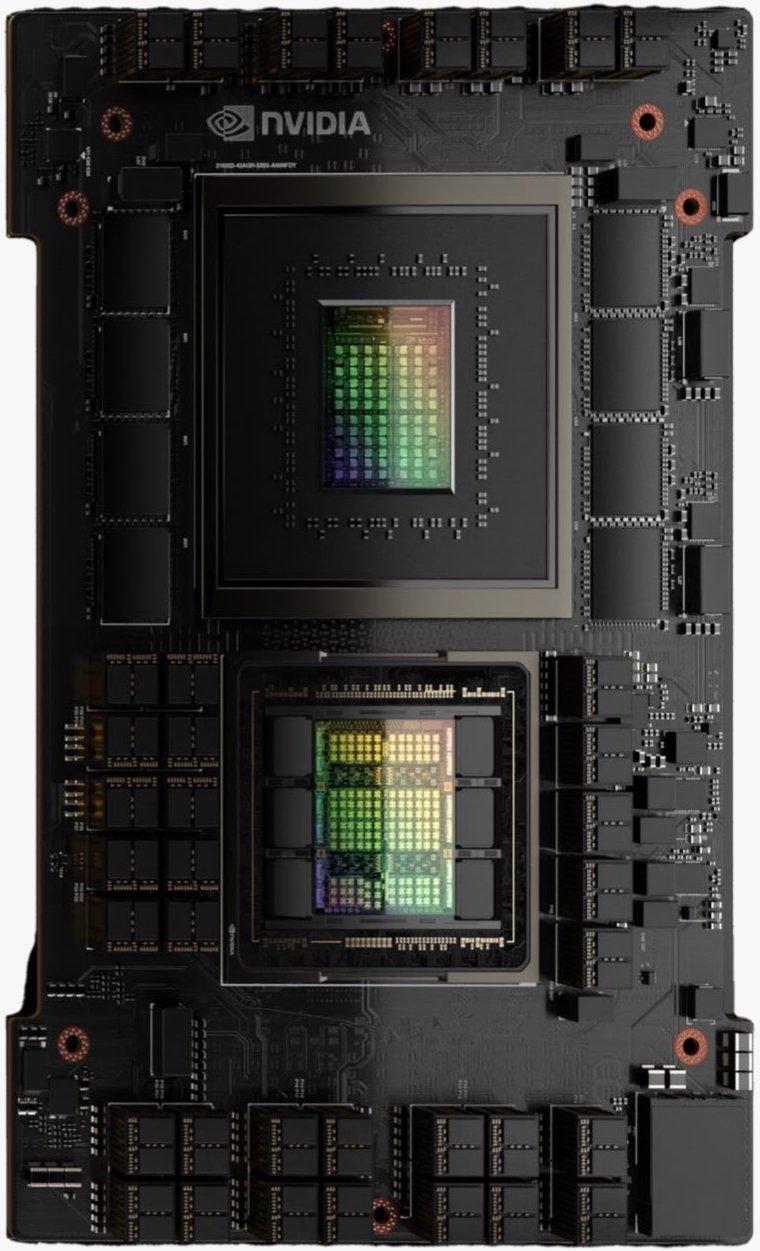
Choosing A Programming Model

There can be ~~only~~ ***more than*** one.

Libraries	Standard Languages	Compiler Directives	CUDA Languages
• Accelerate common operations with little/no code changes. • Expert-tuned performance. • Forward support guarantees.	• Strong cross-platform support. • Single source code for multiple platforms. • Reduced learning curve.	• High cross-platform support. • Single source code for multiple platforms. • Reduced learning curve. • Additional programmer control.	• Exposes full GPU capabilities. • Trades portability for performance. • Distinct GPU/CPU code paths. • Full programmer control.
Programmer Productivity			Programmer Control

Approaches are interoperable.
Significantly increased productivity with coherent memory space.

NVIDIA.



NVIDIA Grace Hopper Superchip

Made ♡ for any programming model

Portable ISO C++, ISO Fortran, Python

- **Simplifies parallelization:** less SW changes

  → **ISO C++, ISO Fortran, Python**: Threads are “threads” (!SIMD), memory consistency, automatic memory management, …

  → **Applications**: complex code stays on CPU, infrequently used memory stays on DDR, large GPU memory capacity (600 GB).

- **Easiest system to:**
  → teach & learn heterogeneous programming
  → parallelize applications
  → use the right HW for each algorithm

ISO C++ Parallel Algorithms

Ranges + Execution Policies + Algorithms

Goal: widespread adoption of parallelism ← portable.

Three key abstractions:

- Ranges
- Execution Policies
- Algorithms

In this talk we include following headers and abbreviations:

```
#include <ranges>
#include <execution>
#include <algorithm>

namespace stdv = std::views;
namespace stde = std::execution;
```

Sequential C++ DAXPY

```
double* x, *y; double a; int n;

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

Parallel C++ DAXPY

```
#include <???>

std::for_each(stde::par, ???,
     [](int i) { y[i] += a * x[i]; });
```

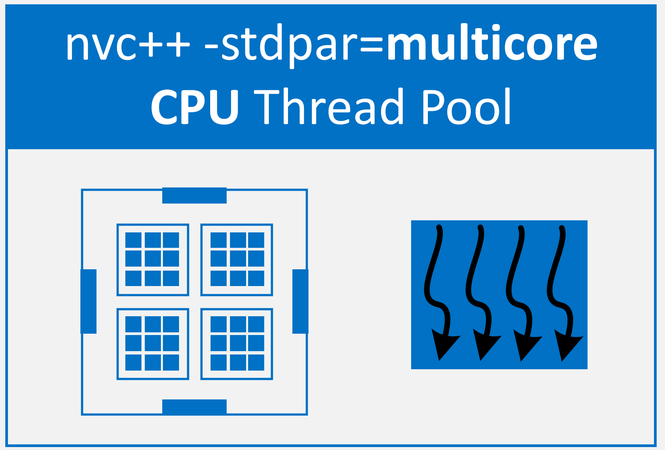
ISO C++ Parallel Algorithms

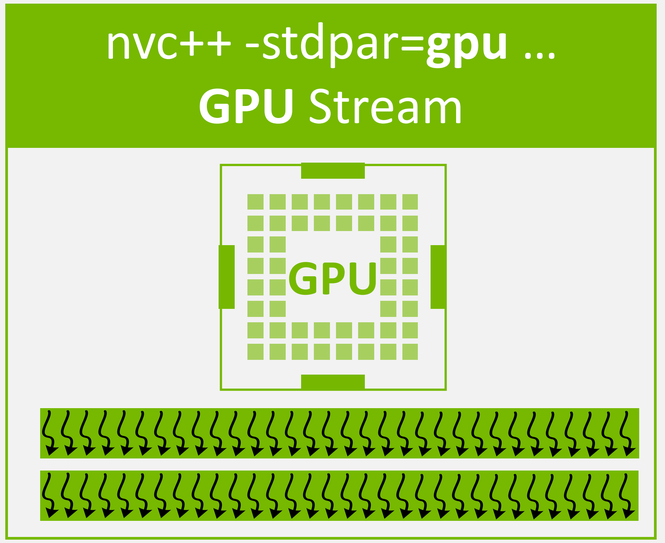
Ranges + Execution Policies + Algorithms

"How" the algorithms may be executed | Not where

- Enable the implementation to assume that work is safe to schedule:
  - `stde::seq` (sequential), `stde::unseq` (SIMD)
  - `stde::par` (threads), `stde::par_unseq` (threads+SIMD)
- *Where* the algorithms are executed is up to the implementation:
  - CPU thread pool,
  - GPU Stream,
  - …

NVIDIA C++ Compiler (nvc++)





Compressible Flow Example

Parallelizing & Optimizing HPC Applications with ISO C++ on Grace Hopper

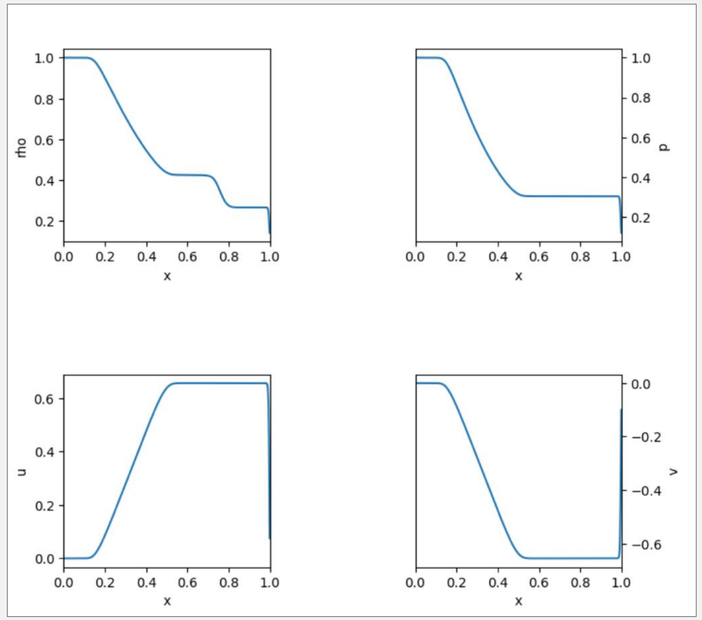
Easy and portable steps to accelerations:

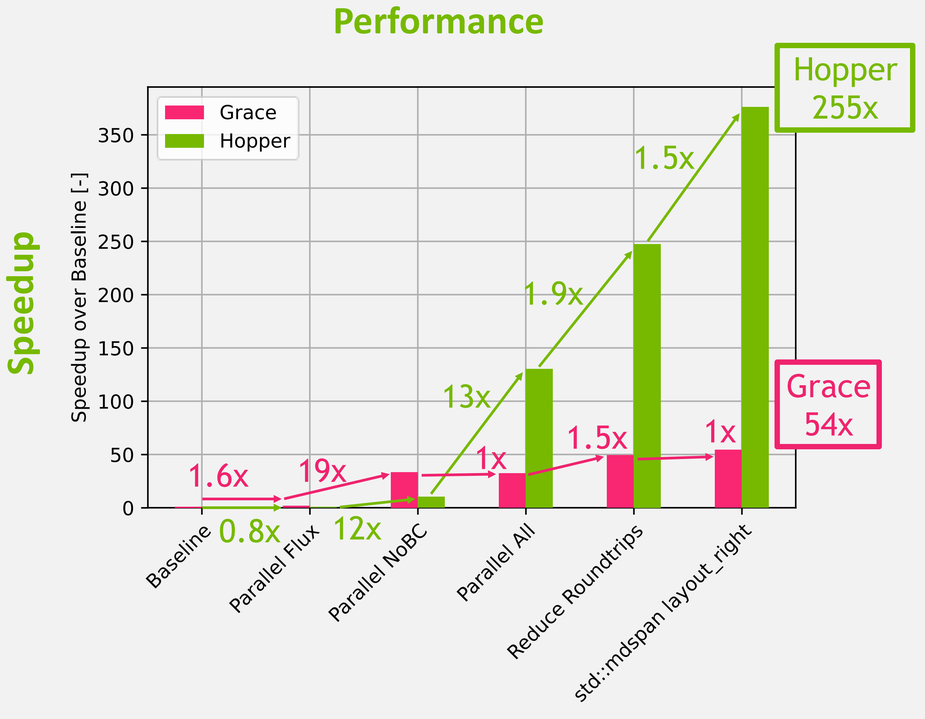
1. use the C++ standard library parallel algorithms and multi-dimensional spans,
2. Apply common optimizations, e.g. algorithm fusion and memory layout.

With these techniques we optimized an HPC application for Grace Hopper written in portable C++:

- Grace CPU Speedup: 54x .
- Hopper GPU Speedup: 255x .

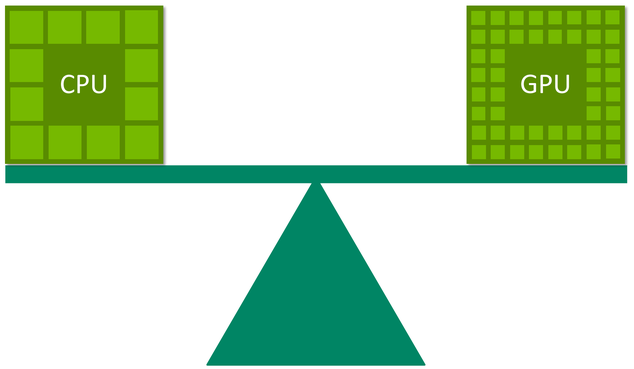
We **did not need to** learn about manual memory management or data-movement between GPUs and CPUs.



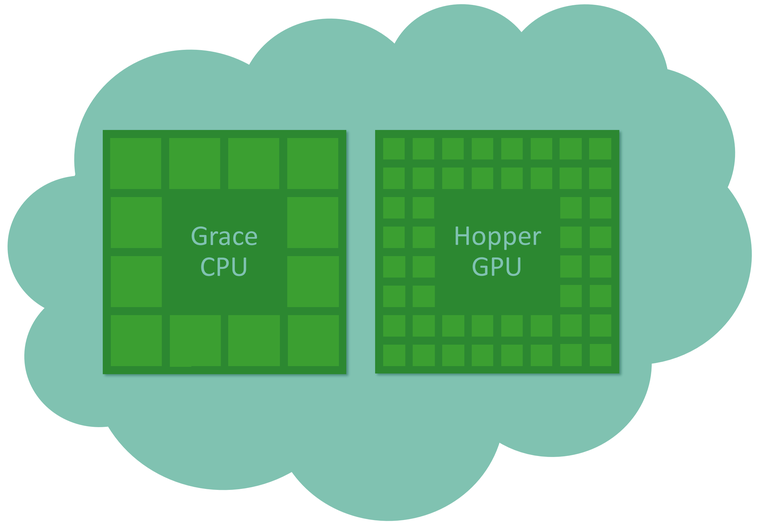


Grace Hopper Superchip

Key takeaways



Balanced & Versatile high-performance system for all existing workloads



Coherent architecture for new applications

```
void jacobi_iteration(vector<double> const& v, vector<double>& tmp) {
    double const* vptr = v.data();
    double *tmp_ptr = tmp.data();
    double l2_error = transform_reduce(execution::par_unseq
        begin(v), end(v), 0., plus<double>, [=](double&
    {
        int i = &x - vptr; // Co   te index of x fr    address.
    auto [iX, iY] = split(i)
        double avg = 0.25
        vptr[fuse(iX-1    ] + vptr[fu      iY)] +
        vptr[fuse(    -1)] + vptr[fuse(    iY+1)] );
        tmp_ptr[i    avg;
        return (avg - x) * (avg - x);
    } );
}
```

Developer velocity for accelerating applications

