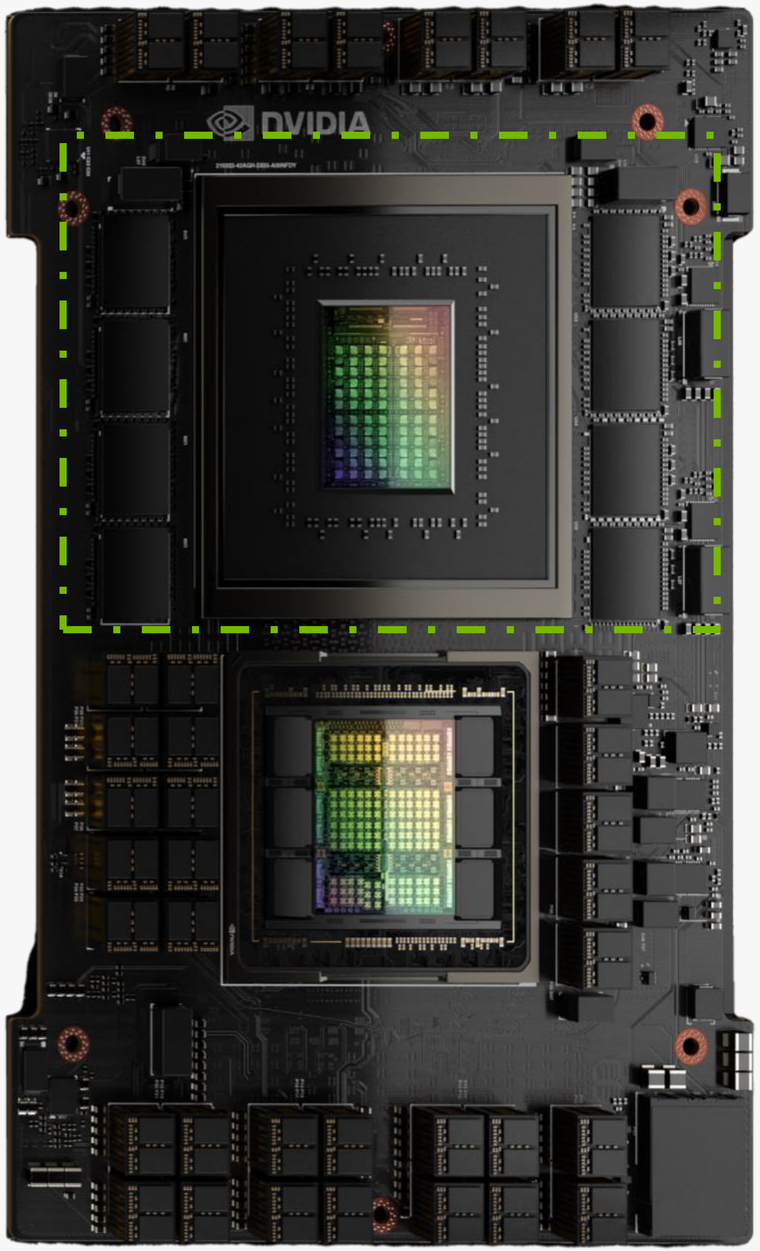
Applications on the NVIDIA Grace Hopper Superchip

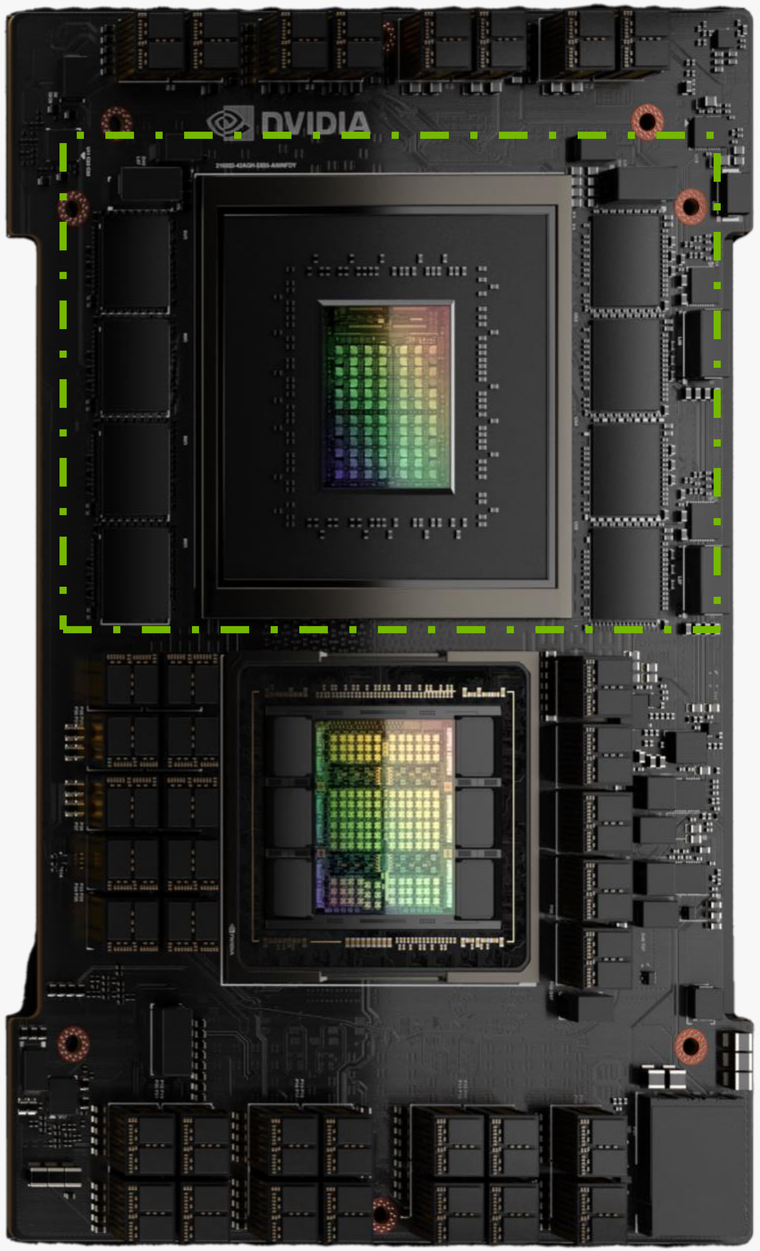
Mathias Wagner, Sr Developer Technology Engineer | CASA Workshop 2024



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:
    - → Capacity: 96 GB HBM3 / 144 GB HBM3e
    - → Extreme bandwidth ≤ 4000 GB/s / 5000 GB/s
  - ≤ 18x NVLink 4 → 900 GB/s
  - → Threads are threads

NVIDIA Grace Hopper Superchip

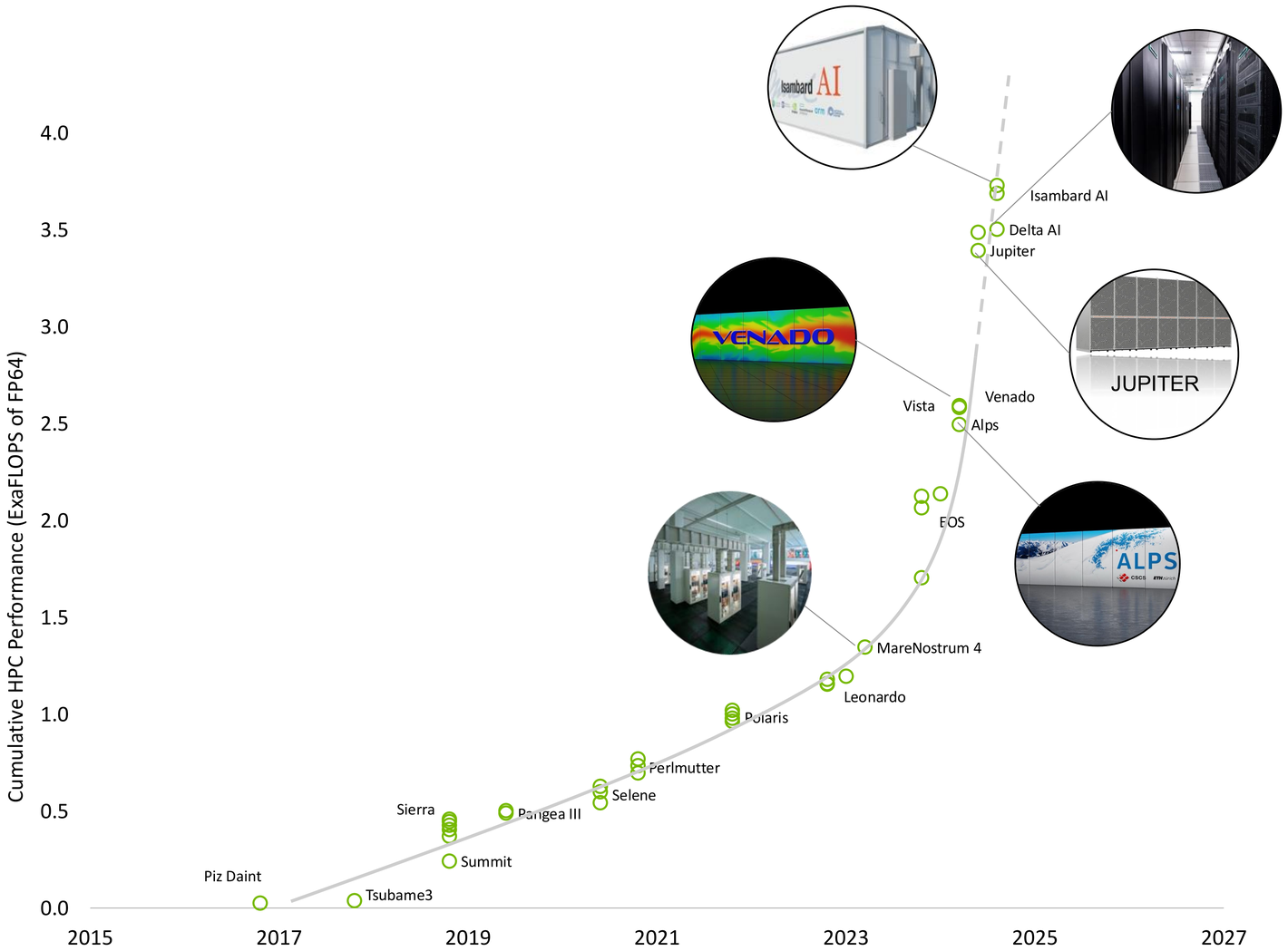
Soul is the new **NVLink-C2C** CPU ←→ GPU interconnect

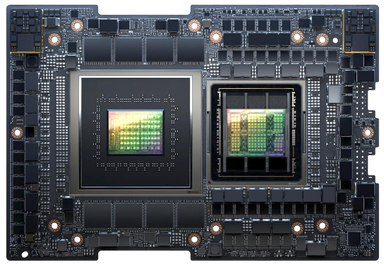
- **Memory consistency**: ease of use
  - →*All* threads – GPU and CPU – access system memory: C++ `new`, `malloc`, `mmap`'ed files, atomics, …
  - → Fast automatic page migrations
  - → Threads cache peer memory → Less migrations
- **High-bandwidth**: 900 GB/s (same as peer NVLink 4)
  - → GPU reads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

Next-Gen Supercomputing Datacenter

4 ExaFLOPs of HPC Performance Driving Scientific Innovation



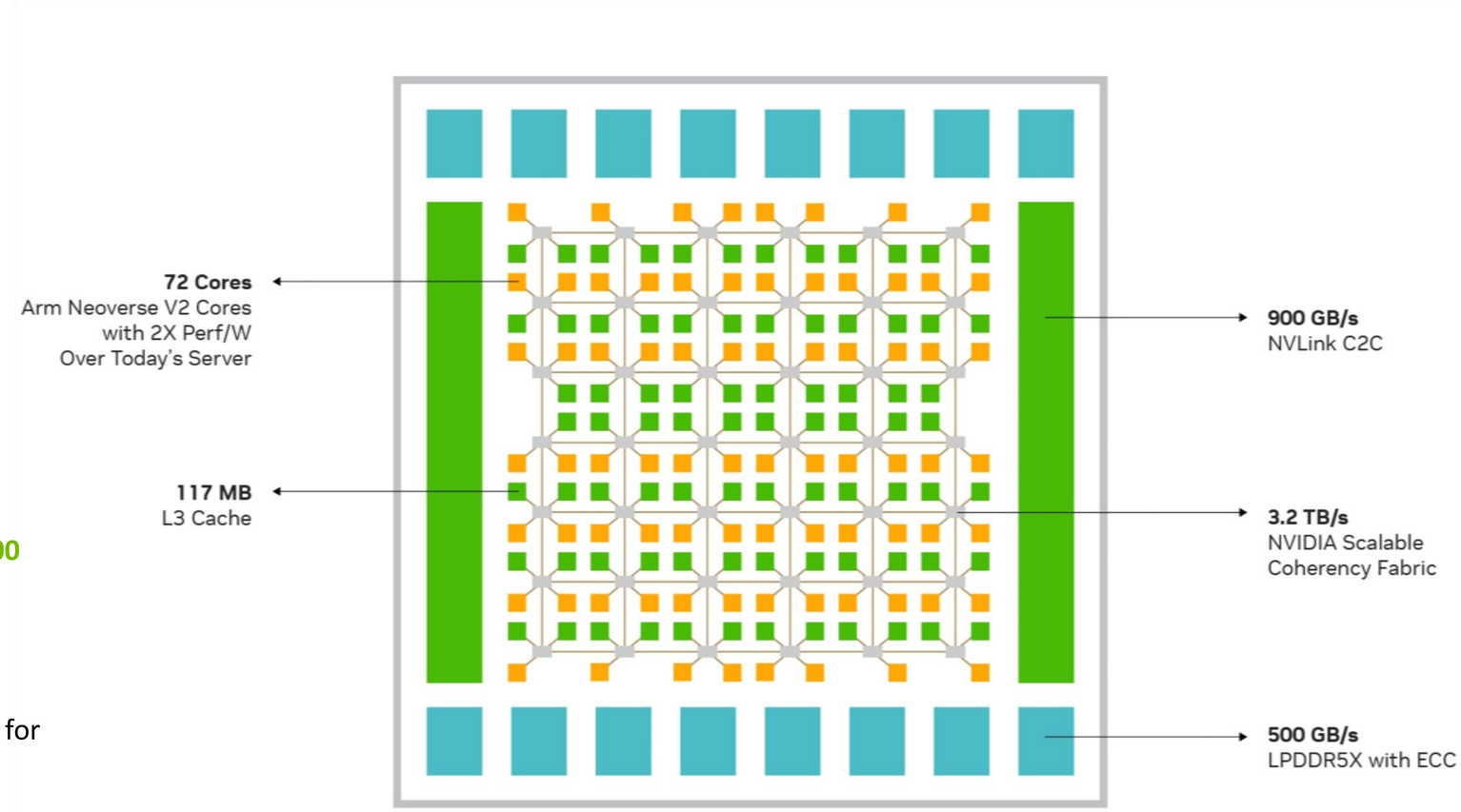


1.7 Exaflops Grace Hopper
Coming online 2024

Grace Architecture

The CPU building block of the Grace-Hopper superchip

- **High Performance Power Efficient Cores**
  - 72 flagship Arm Neoverse V2 Cores (Armv9-A)
  - 4x128b SVE2 SIMD units per core (SVE2 / NEON)
  - 3.16 GHz Base Clock / 2.7 GHz Vector Clock
  - **3.6 FP64 TFLOP/s**
- **Scalable Coherency Fabric**
  - **3.2 TB/s of bisection bandwidth** connects CPU cores, NVLink-C2C, memory, and system IO
- **High-Bandwidth Low-Power Memory**
  - Up to 480 GB of LPDDR5X memory that delivers **up to 500 GB/s of memory bandwidth**
- **Coherent Chip-to-Chip Connections**
  - NVLink-C2C with **900 GB/s raw bidirectional bandwidth** for coherent connection to CPU or GPU
  - ~7x BW that can be delivered by PCIe Gen 5 link
  - Supports up to 4 chip coherency over coherent NVLink

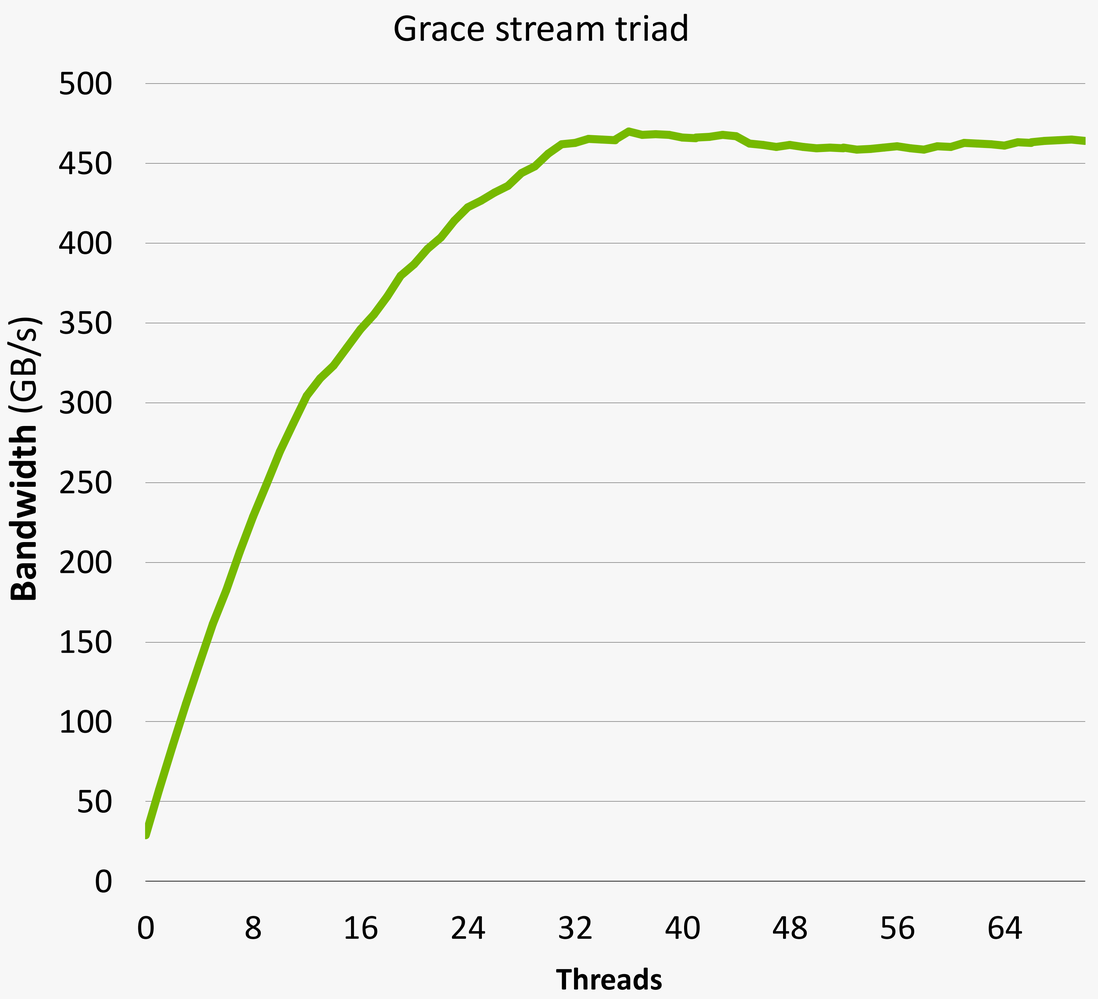


Example possible fabric topology for illustrative purposes

Grace Memory Subsystem

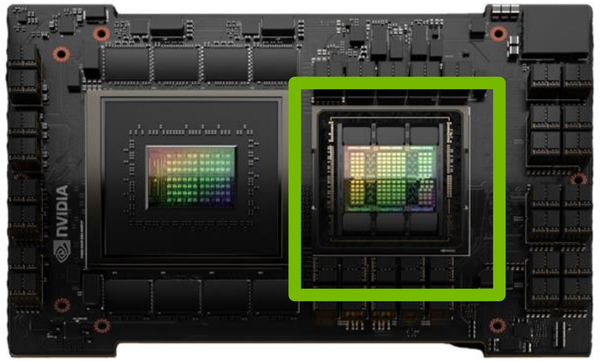
Per 72C Grace SoC

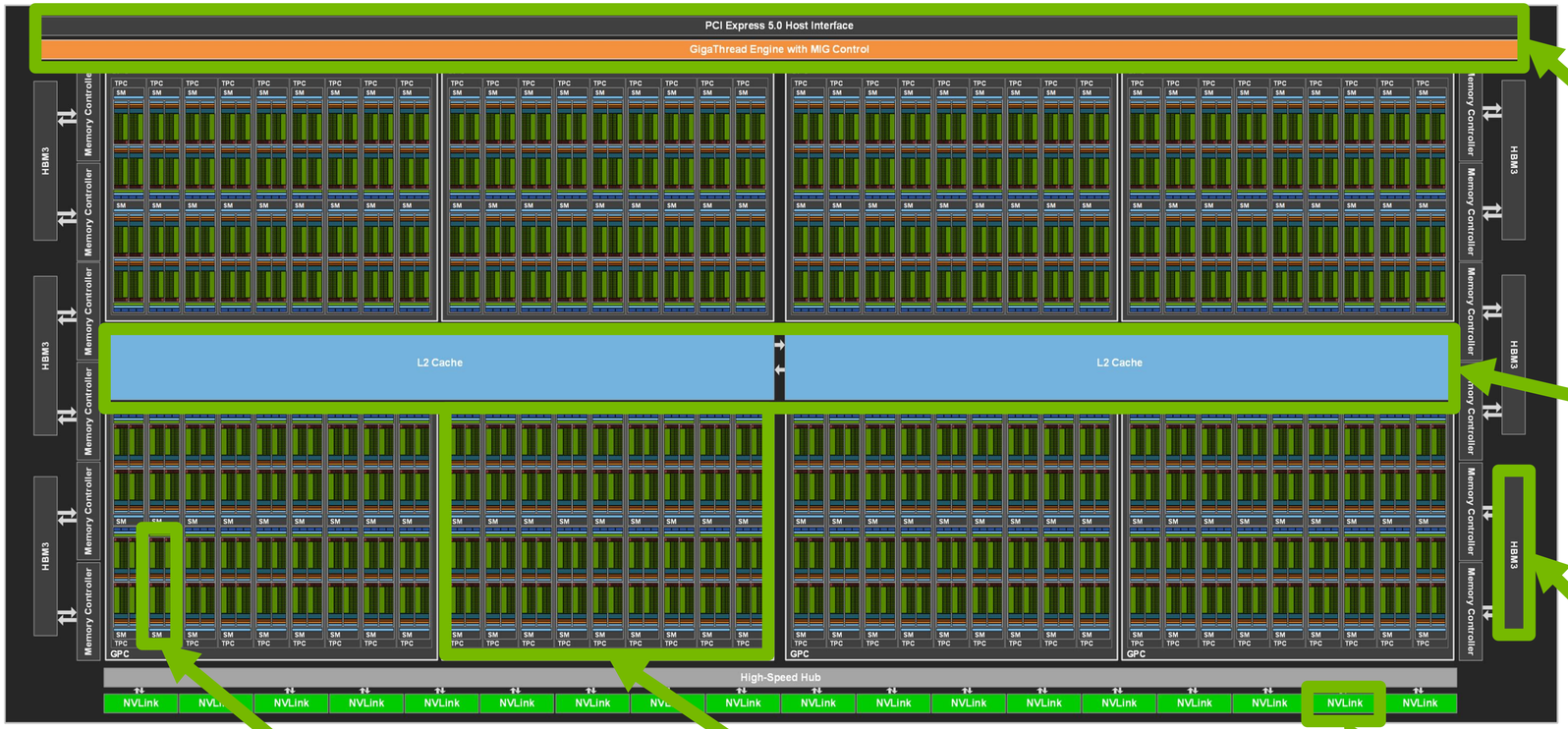
- **Single Die:** single NUMA domain
- **LPDDR5x** (3 variants based on memory capacity)
  - **120GB => BW: 500 GB/s (ALPS)**
  - 240GB => BW: 500 GB/s
  - 480GB => BW: 375 GB/s
- **Cache Hierarchy** (distributed L1 & L2, shared L3)
  - **L1 (per core) =>** 64 KB i-cache and 64 KB d-cache per core
  - **L2 (per core) =>** 1MB per core
  - **L3 (shared) =>** 117MB for the entire chip
- **Local caching of remote die memory !**



Hopper GPU Architecture

The GPU building block of the Grace-Hopper superchip





2nd Gen Multi-Instance GPU
Confidential Computing
PCIe Gen5

Larger 60 MB L2

96GB HBM3, 4 TB/s bandwidth

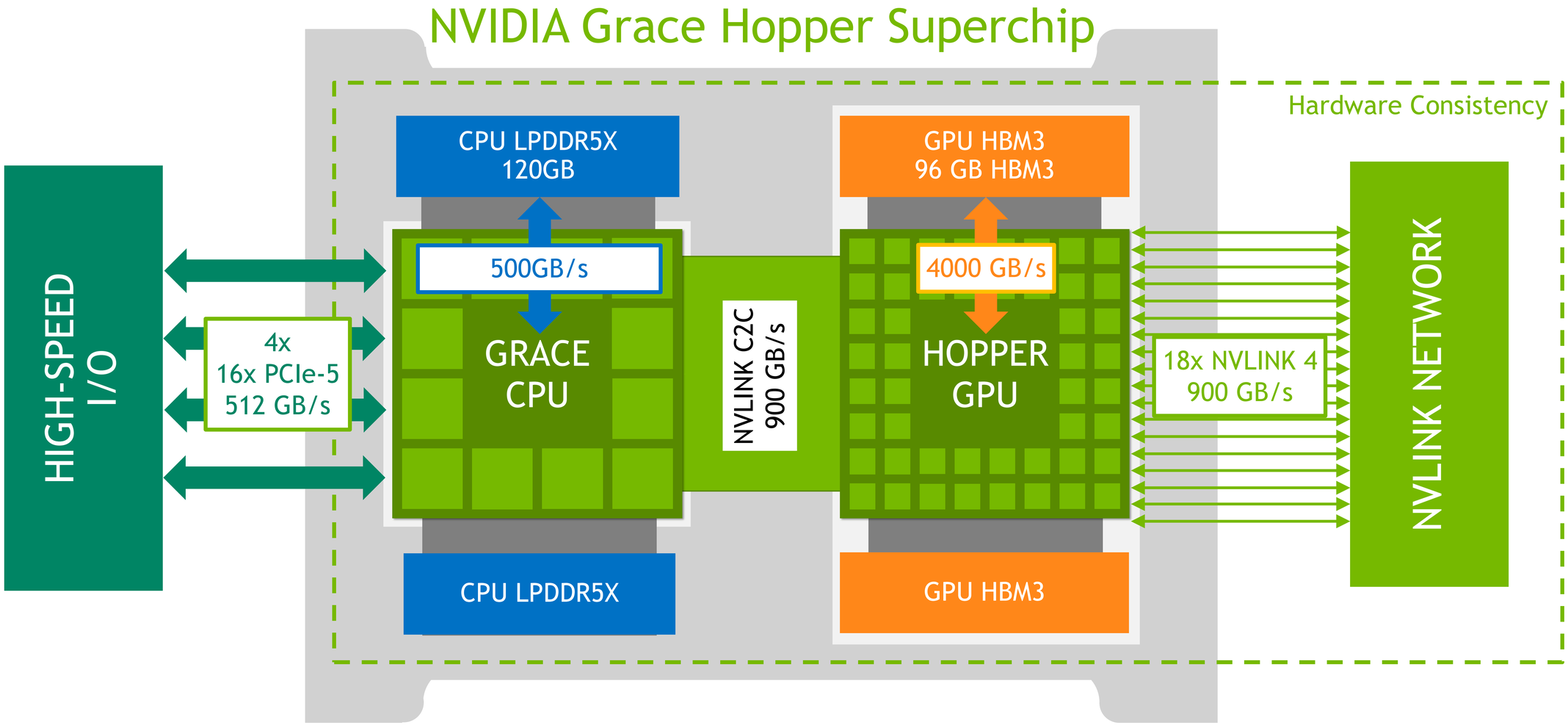
132 SMs
4th Gen Tensor Core

GPU Processing Clusters (GPC) "Thread Block Clusters"

4th Gen NVLink
900 GB/s total bandwidth

Grace Hopper Superchip

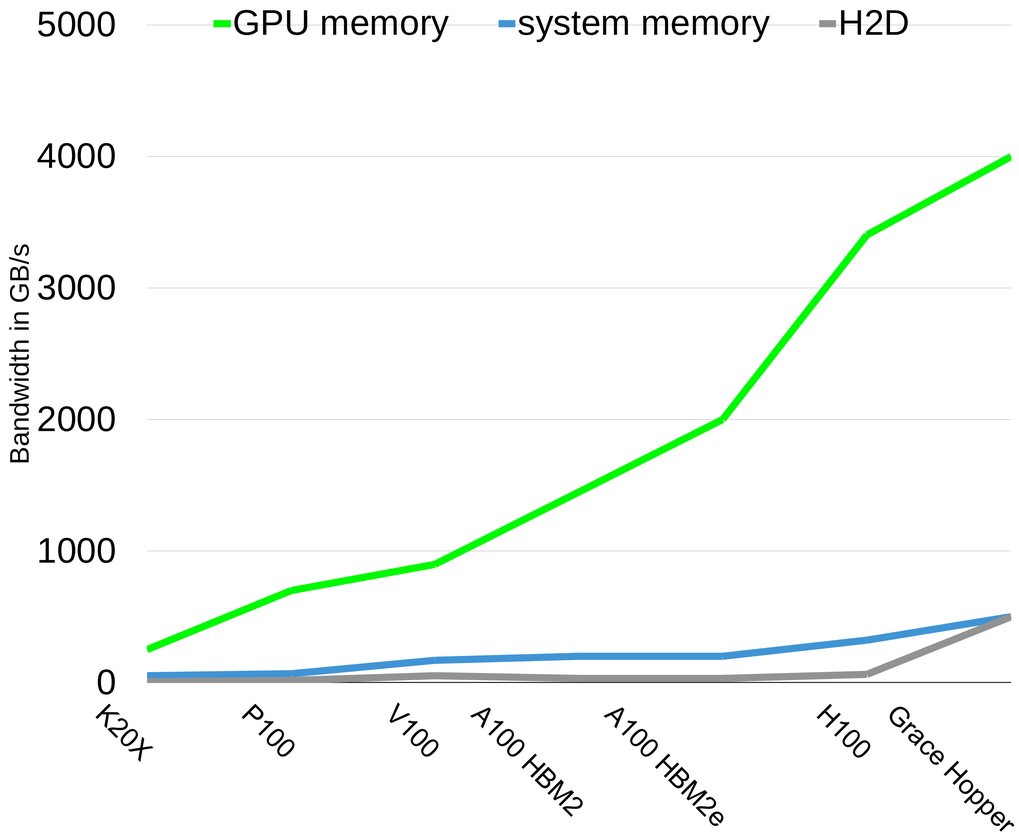
GPU can access CPU memory at CPU memory speeds



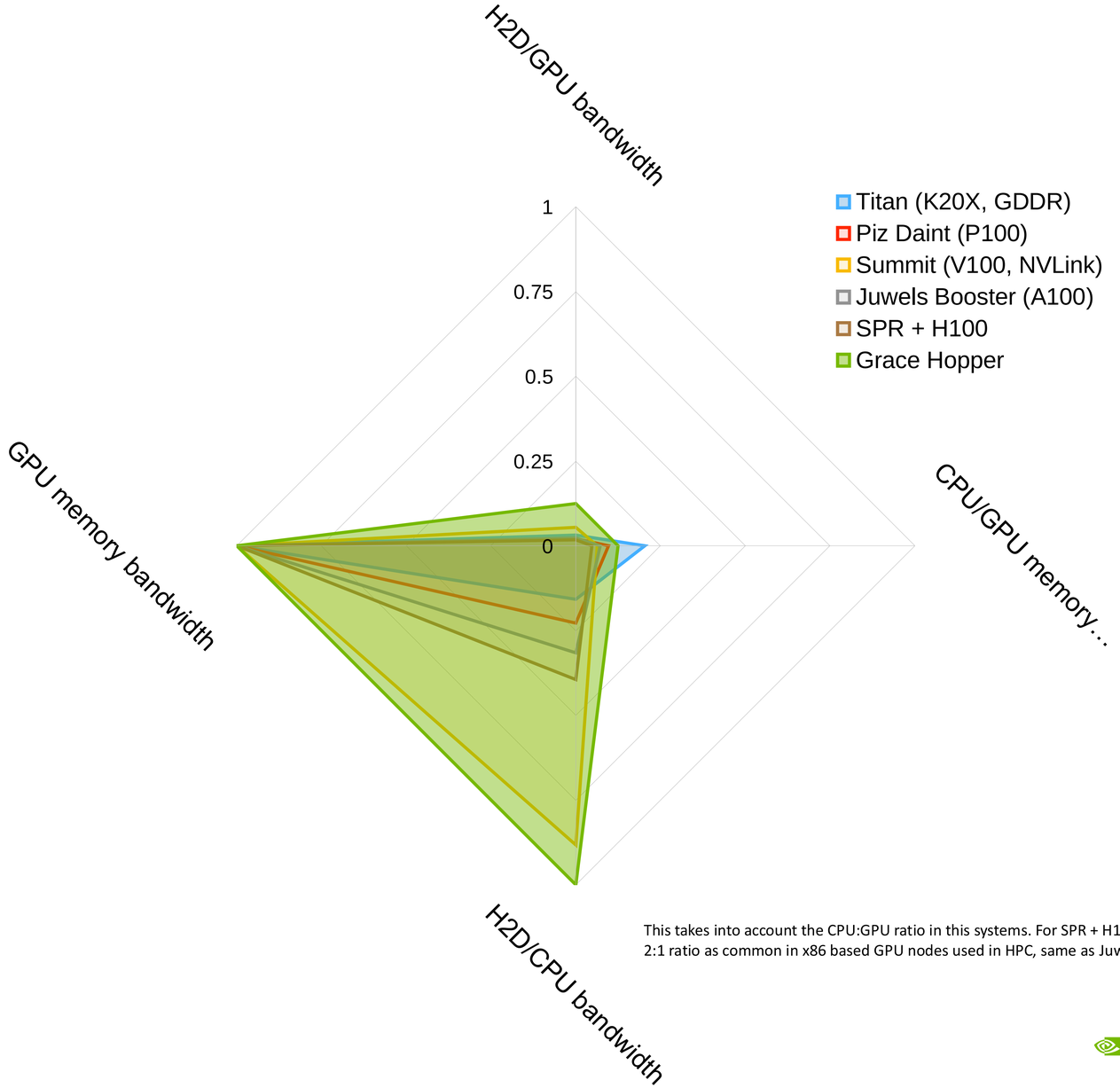
https://resources.nvidia.com/en-us-grace-cpu/nvidia-grace-hopper

Widening the bottlenecks

How much do transfer and system memory bandwidth limit your application?



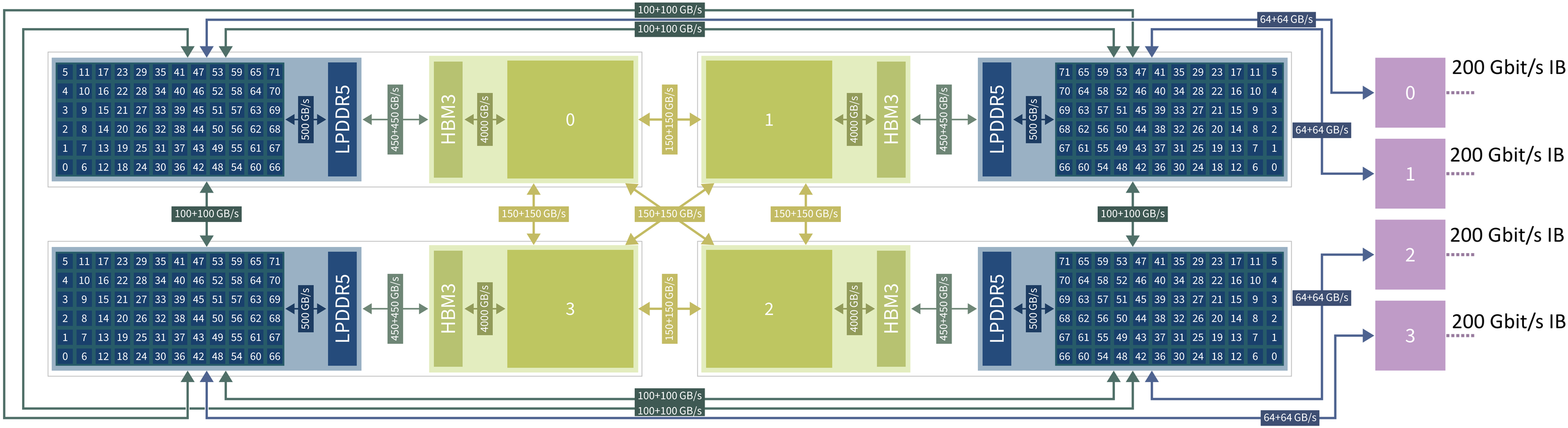
Assumes a typical CPU used in the timeframe.



This takes into account the CPU:GPU ratio in this systems. For SPR + H100 we assume a 2:1 ratio as common in x86 based GPU nodes used in HPC, same as Juwels Booster..

Node Architecture of Jupiter (Jedi) Supercomputer

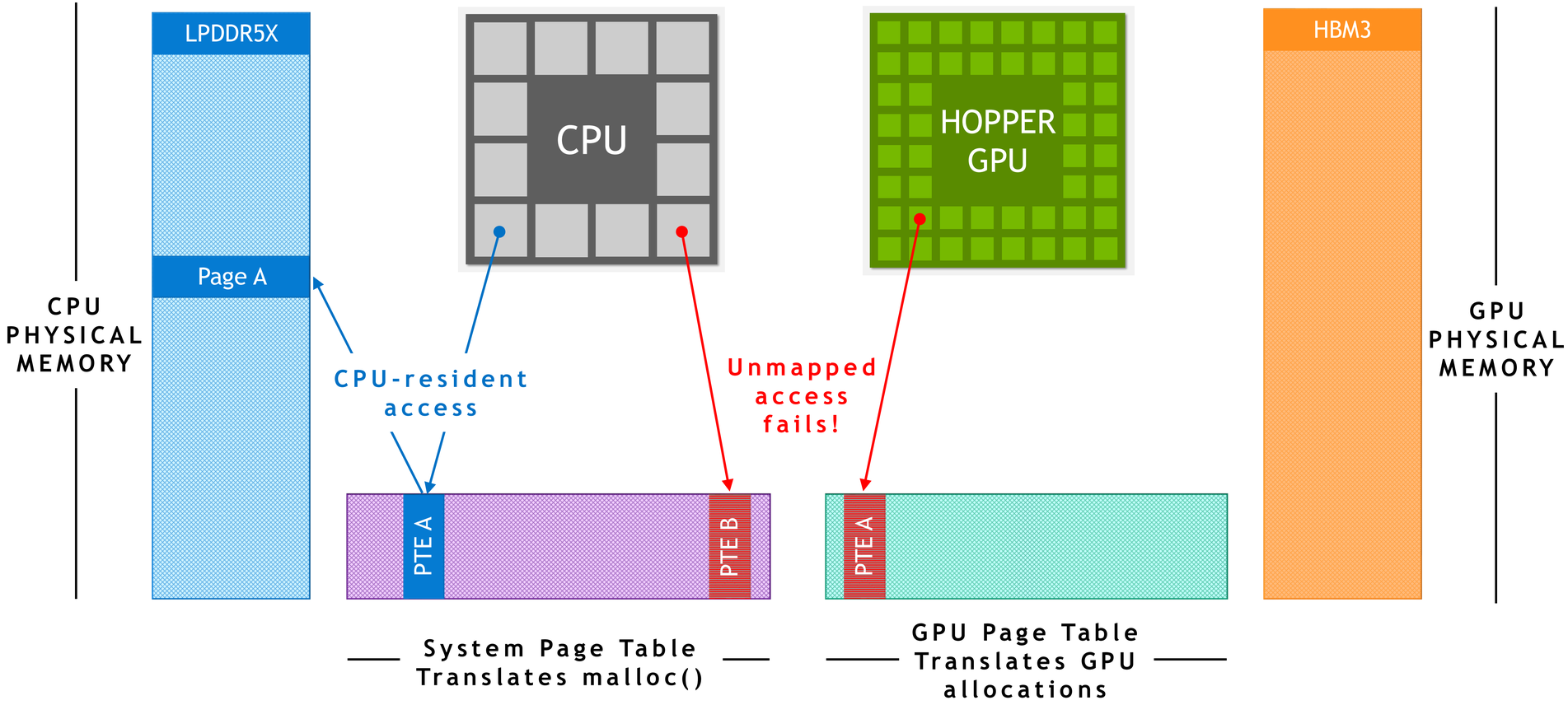
4 x Grace-Hopper Superchips



Memory Coherency on Grace Hopper

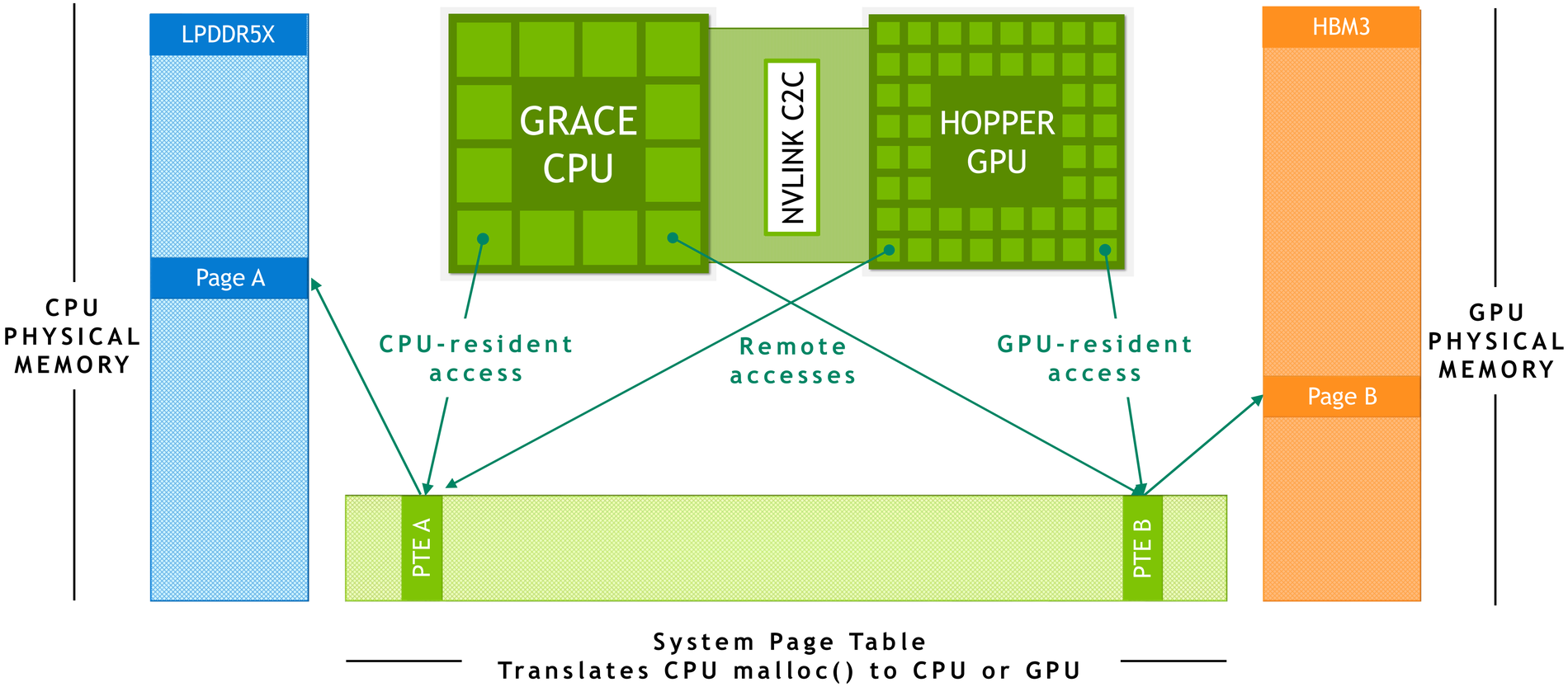
X86 + GPU

Separate page tables

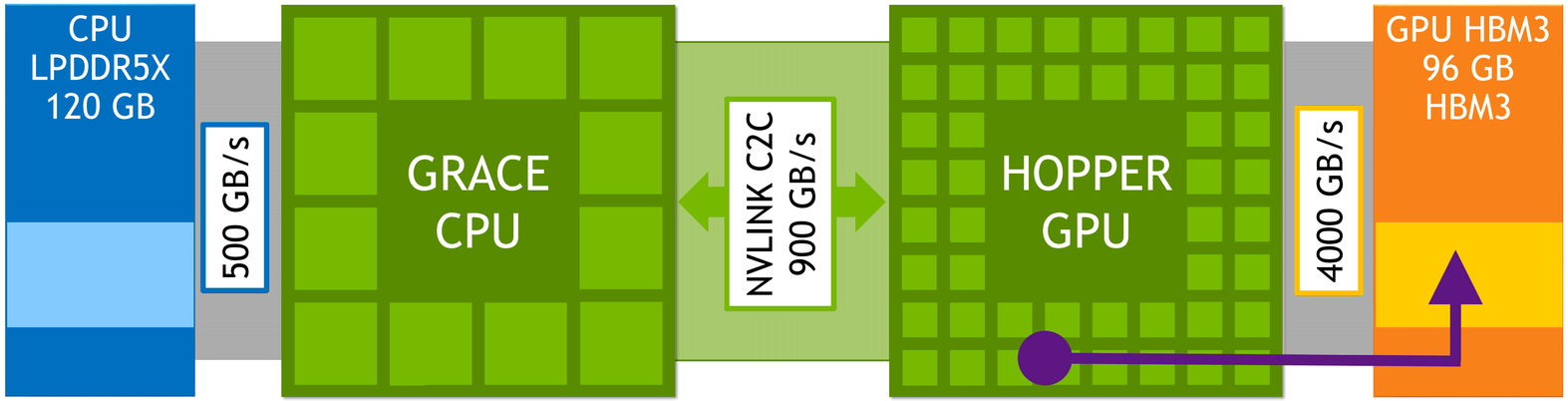


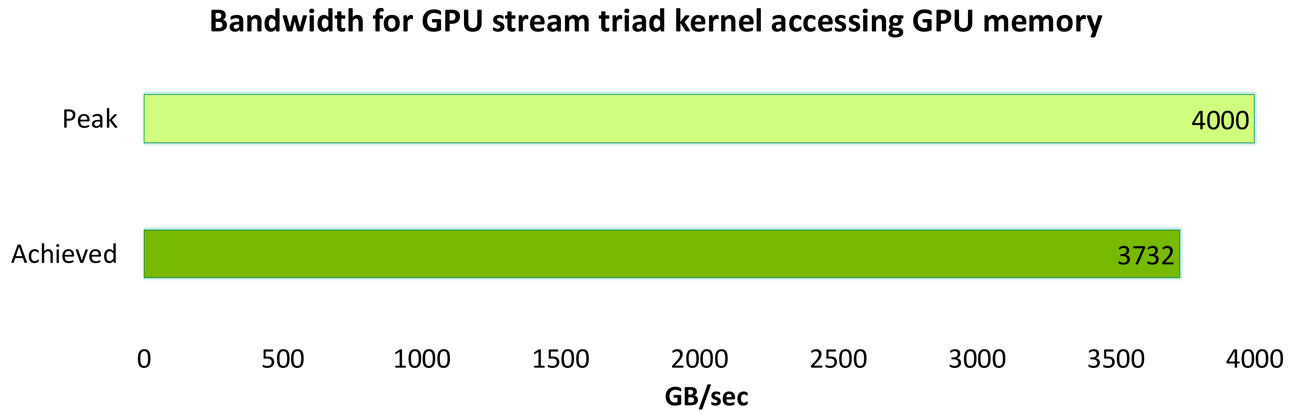
Grace Hopper

Address Translation Service (ATS)

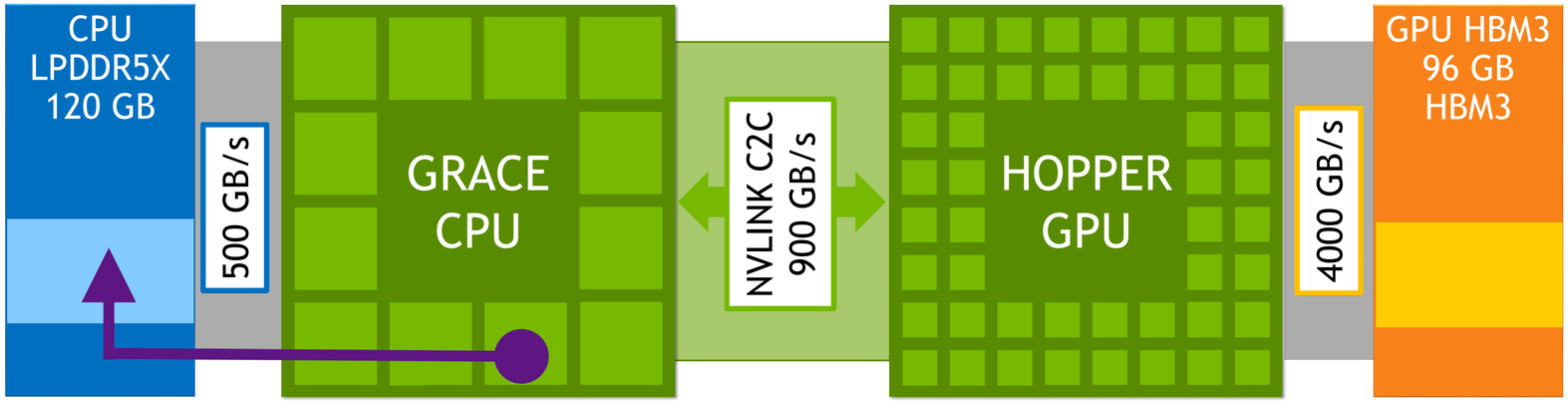


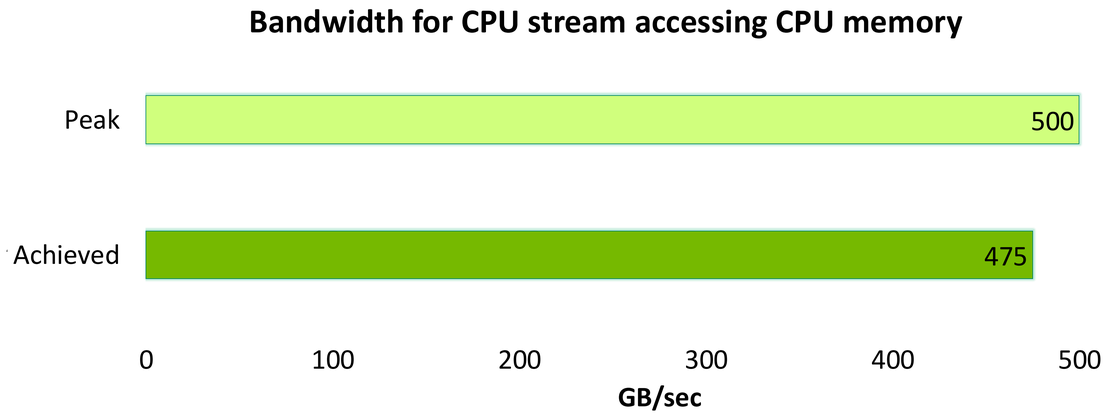
High Bandwidth Memory Access



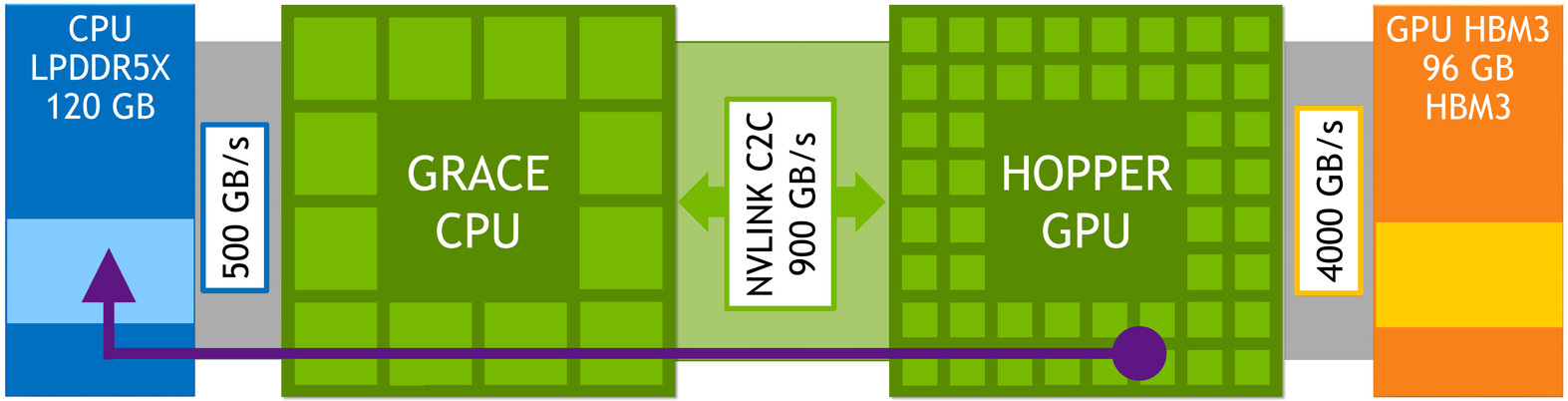


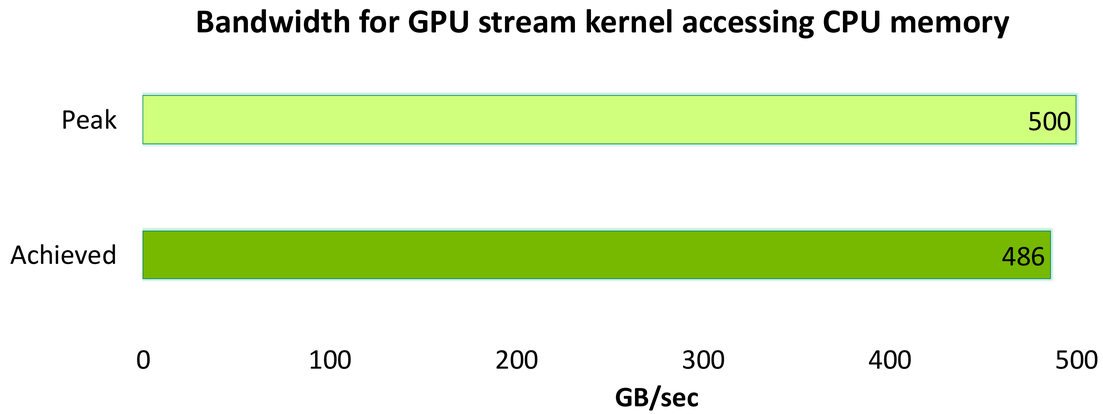
High Bandwidth Memory Access





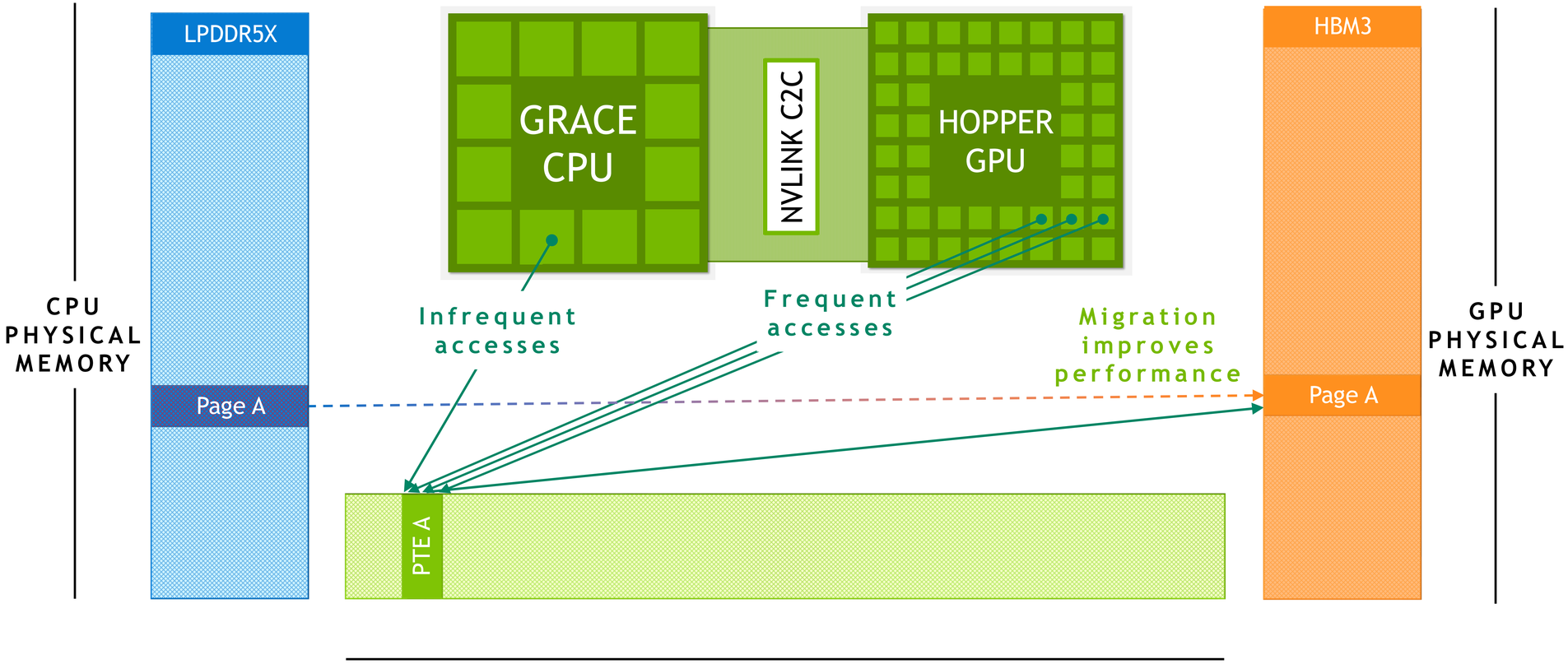
High Bandwidth Memory Access





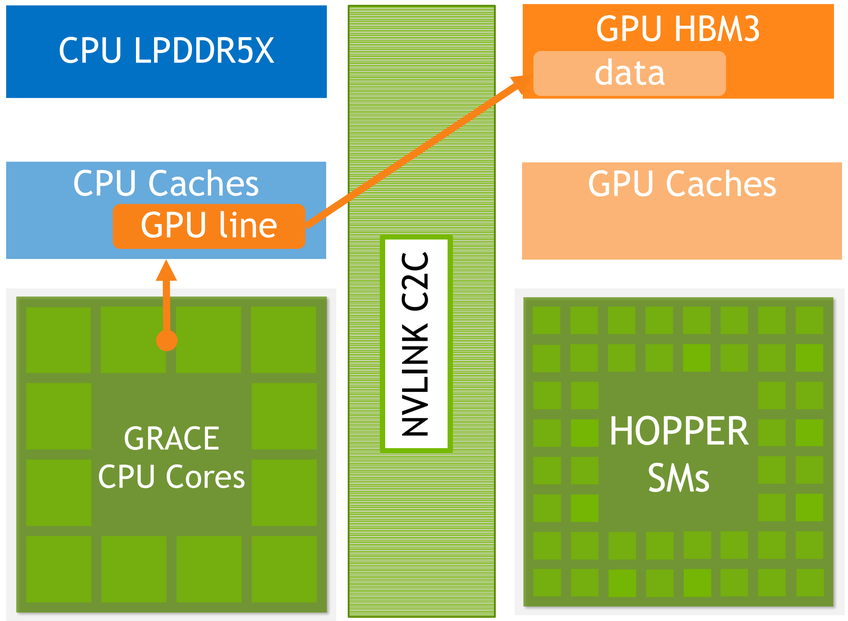
Grace Hopper

Automatic Page Migrations for Unified Memory (CPU -> GPU)



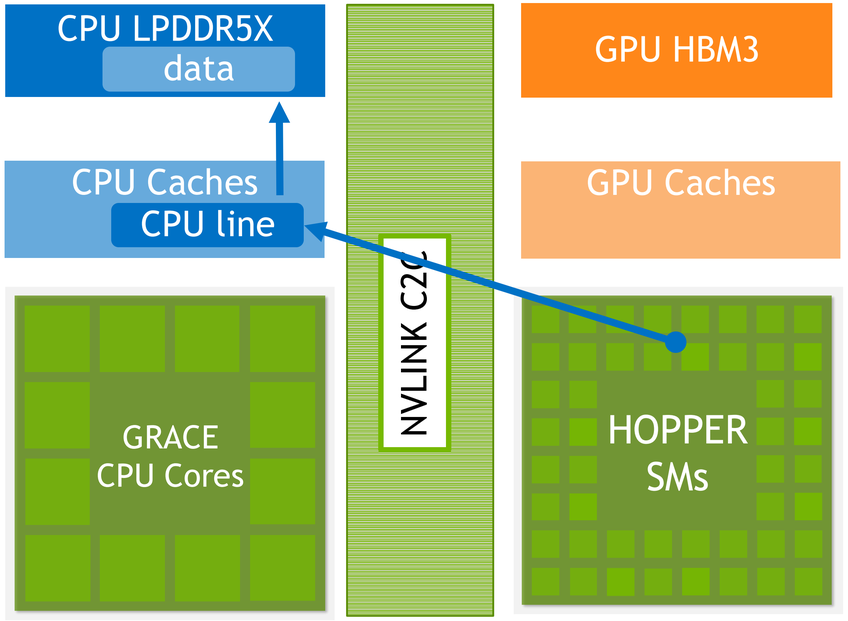
Global Access to All Data

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

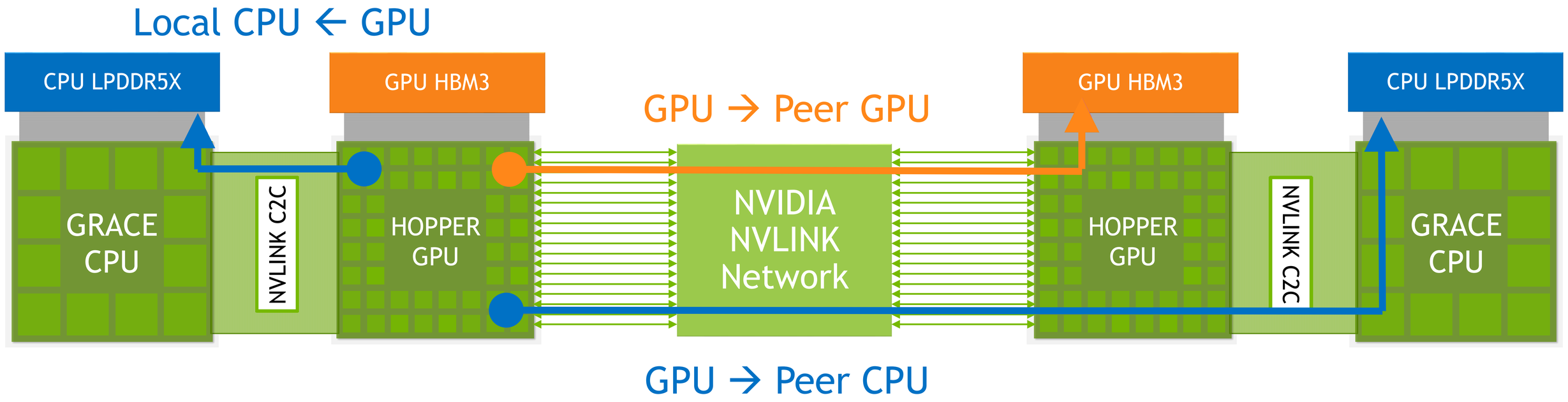


Hopper directly reading Grace's memory

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

Access Paths

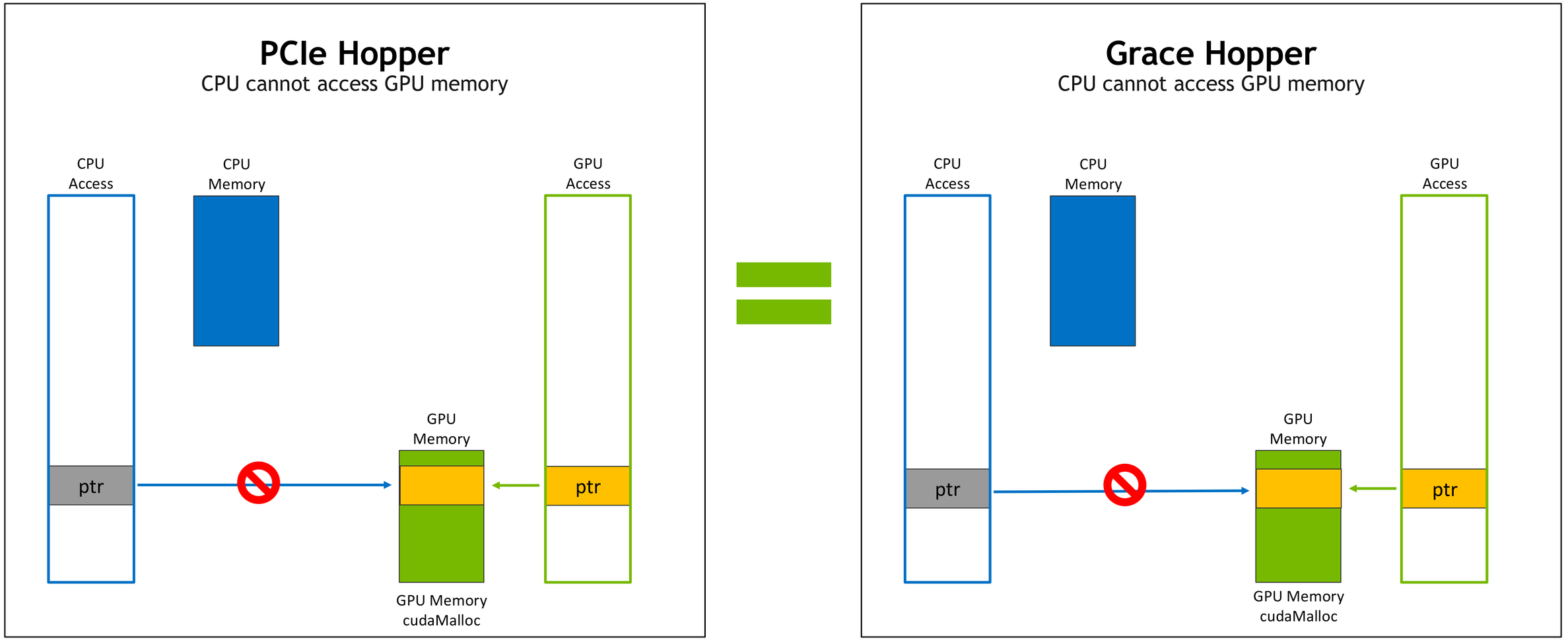
Connecting Grace Hopper Superchips with Memory Consistent NVLink



Memory Management on Grace Hopper

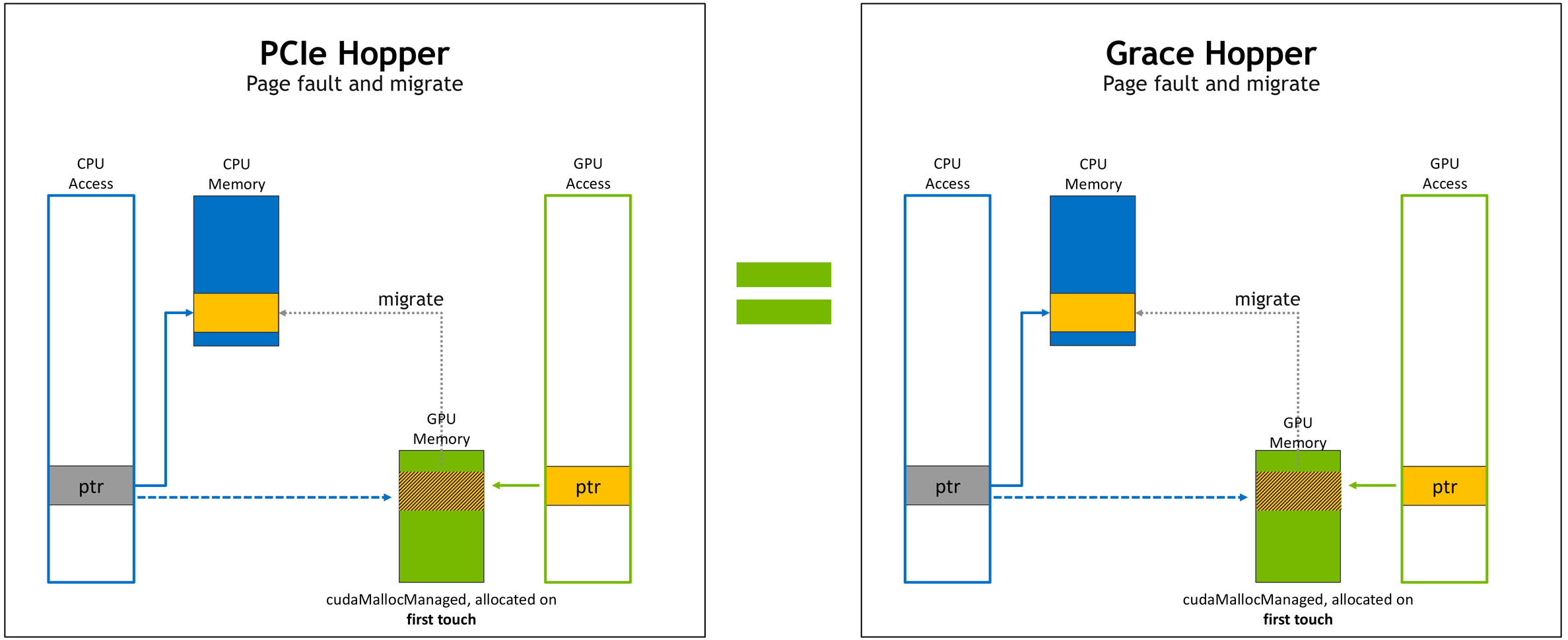
Memory Allocators

Device-only Memory Management (cudaMalloc)



Memory Allocators

CUDA Managed Memory (cudaMallocManaged)

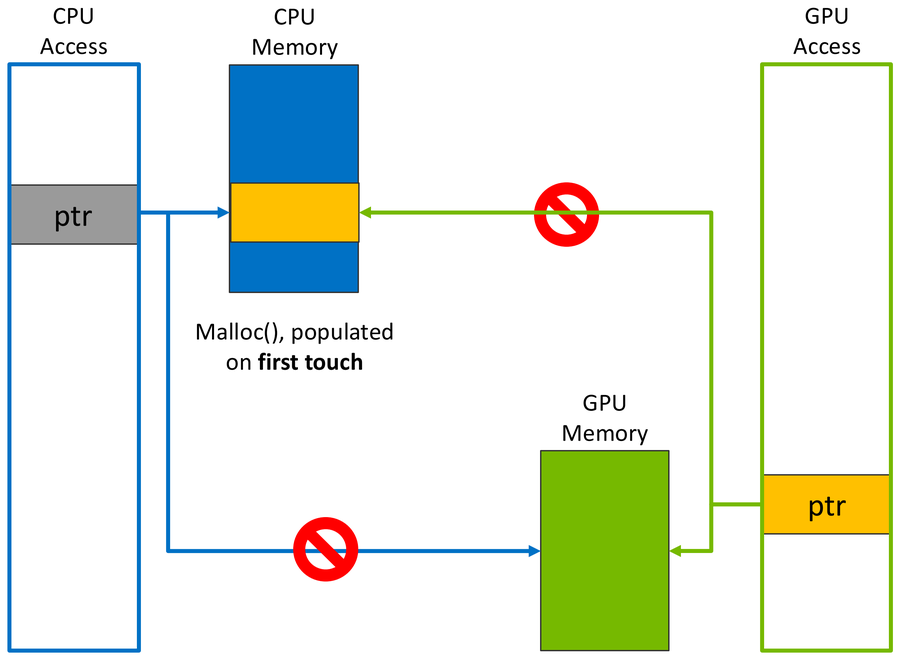


Memory Allocators

System Allocated Memory (malloc/mmap)

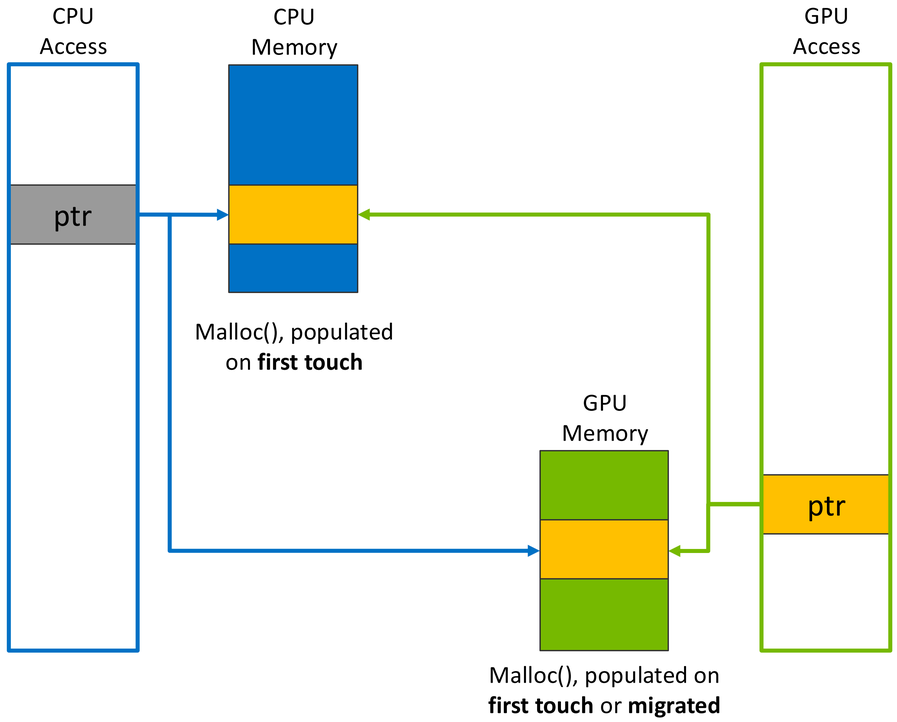
PCIe Hopper

GPU cannot access CPU memory and vice-versa*



Grace Hopper

Direct access over C2C for both CPU and GPU

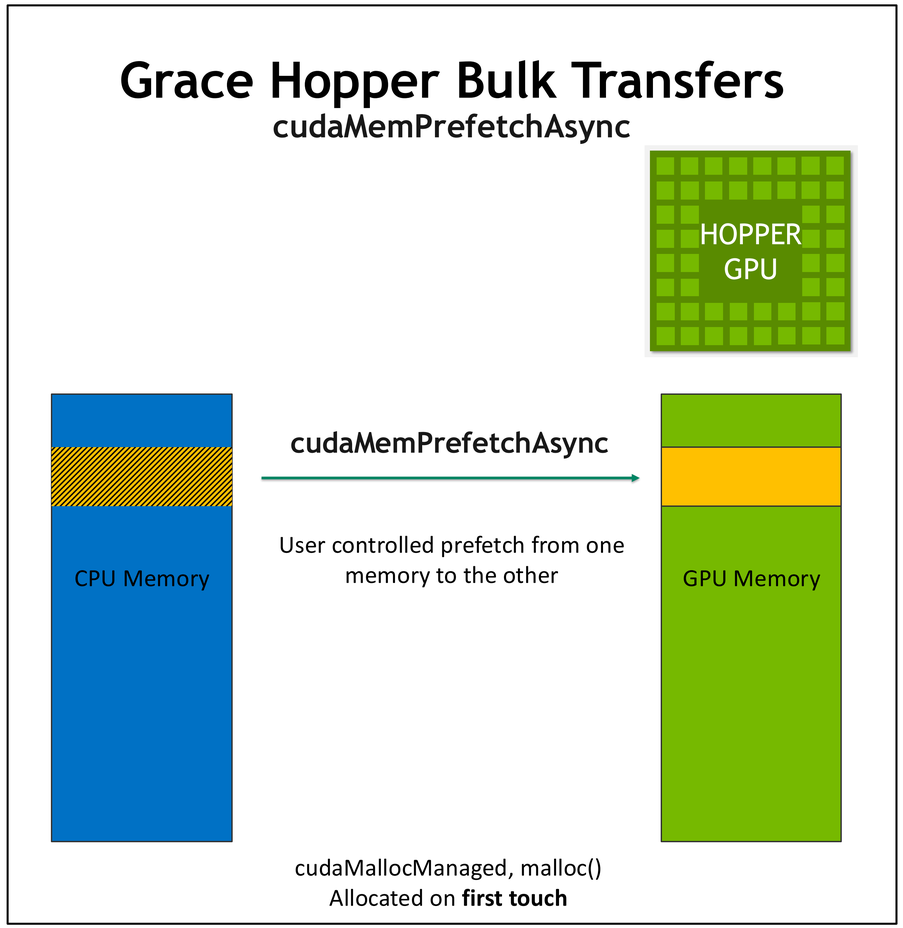


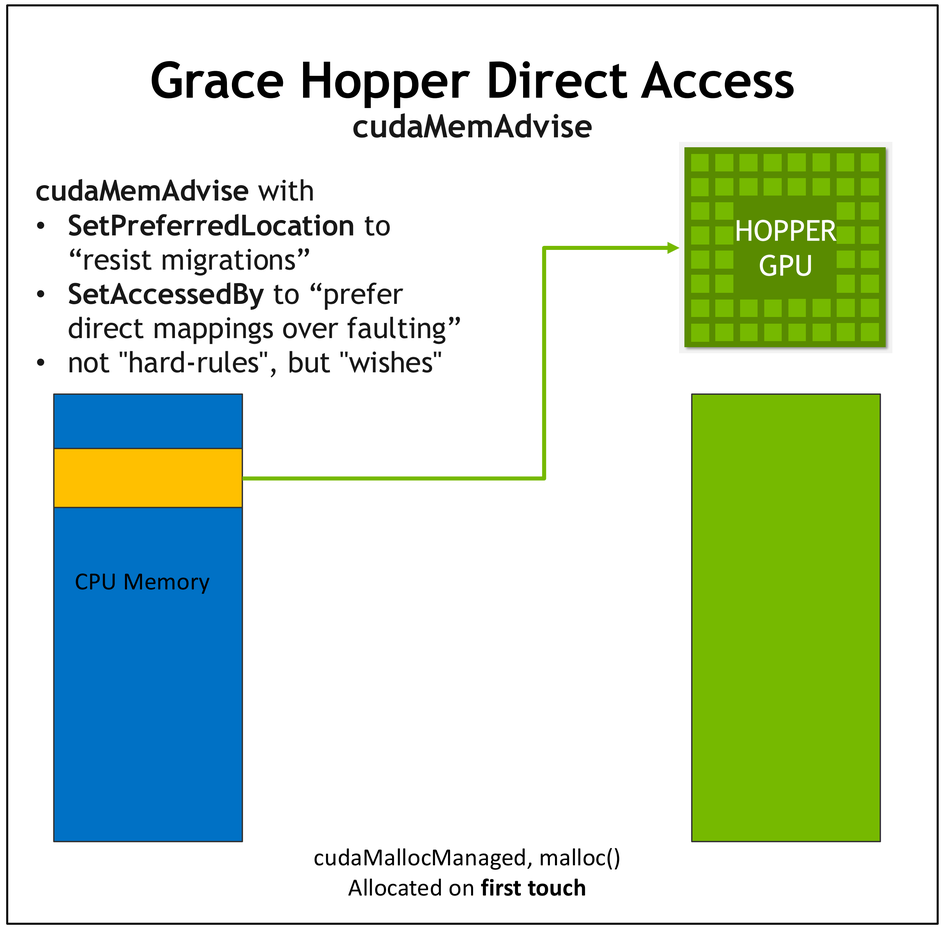
* Heterogeneous Memory Management (HMM) can emulate the behaviour using page faults and migration

Grace Hopper Optimizations

CUDA memory tuning APIs

- On Grace Hopper CUDA memory tuning APIs can be used for both cudaMallocManaged and System Allocated Memory.



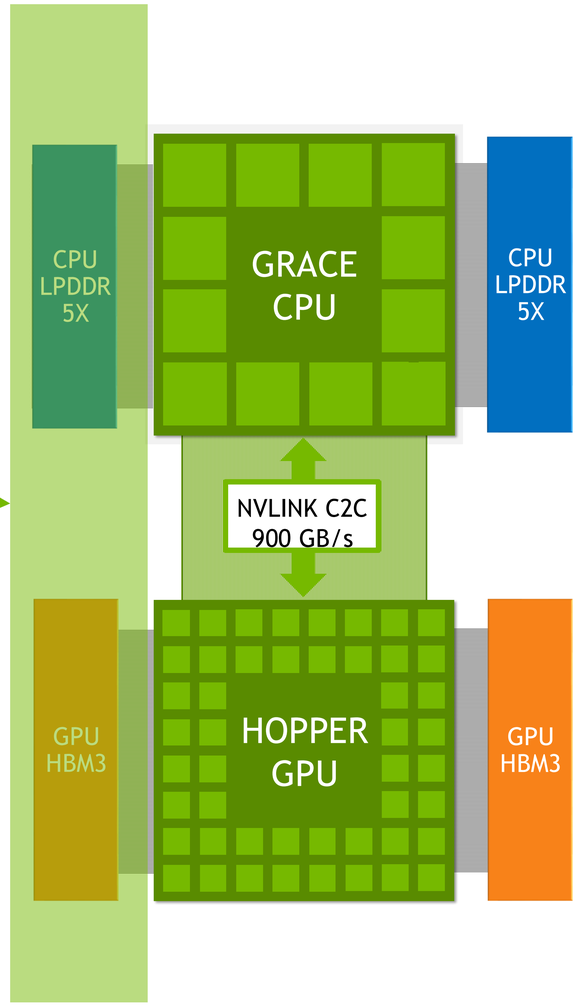


Memory Allocators

CUDA Managed Memory (cudaMallocManaged)

```
__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;
    }

    cudaMemPrefetchAsync(data, N * sizeof(int), 0);
    kernel<<<1, N>>>(data);
    cudaDeviceSynchronize();

    cudaFree(data);
}
```

Physical pages are not yet allocated

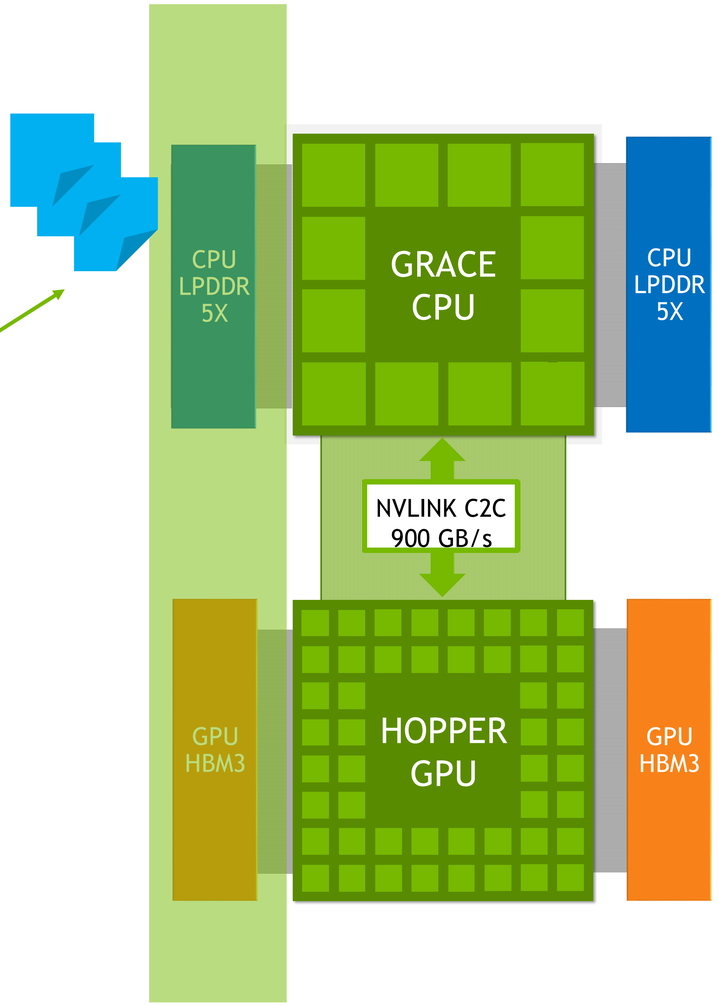


Memory Allocators

CUDA Managed Memory (cudaMallocManaged)

```
__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);
}
```

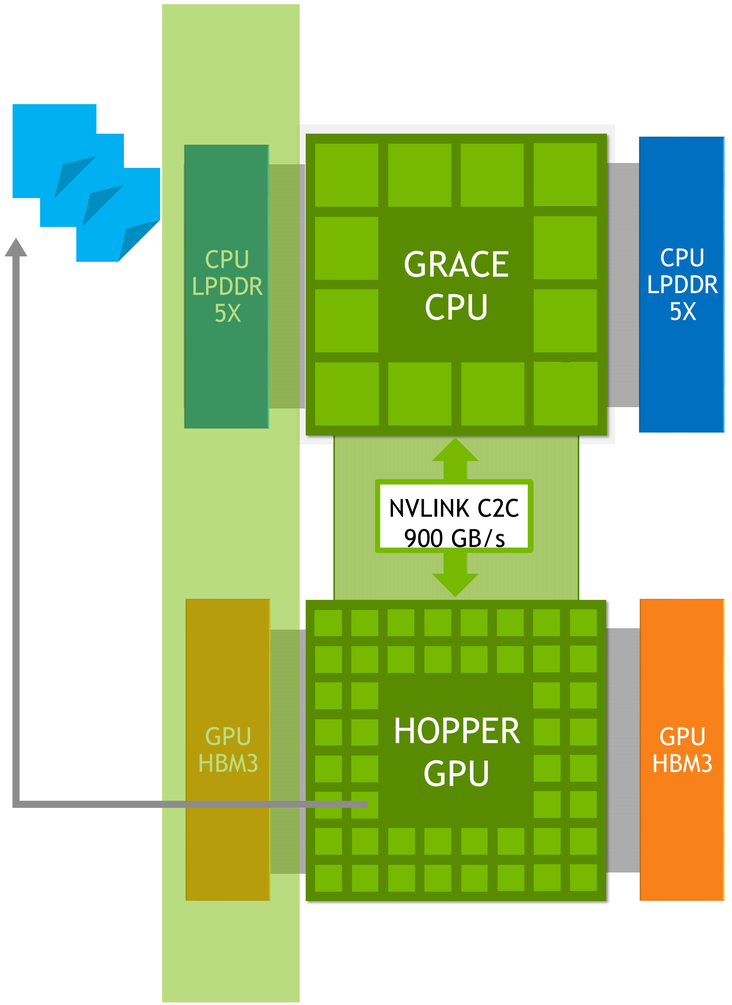
Physical pages are allocated on first touch



Memory Allocators

CUDA Managed Memory (cudaMallocManaged)

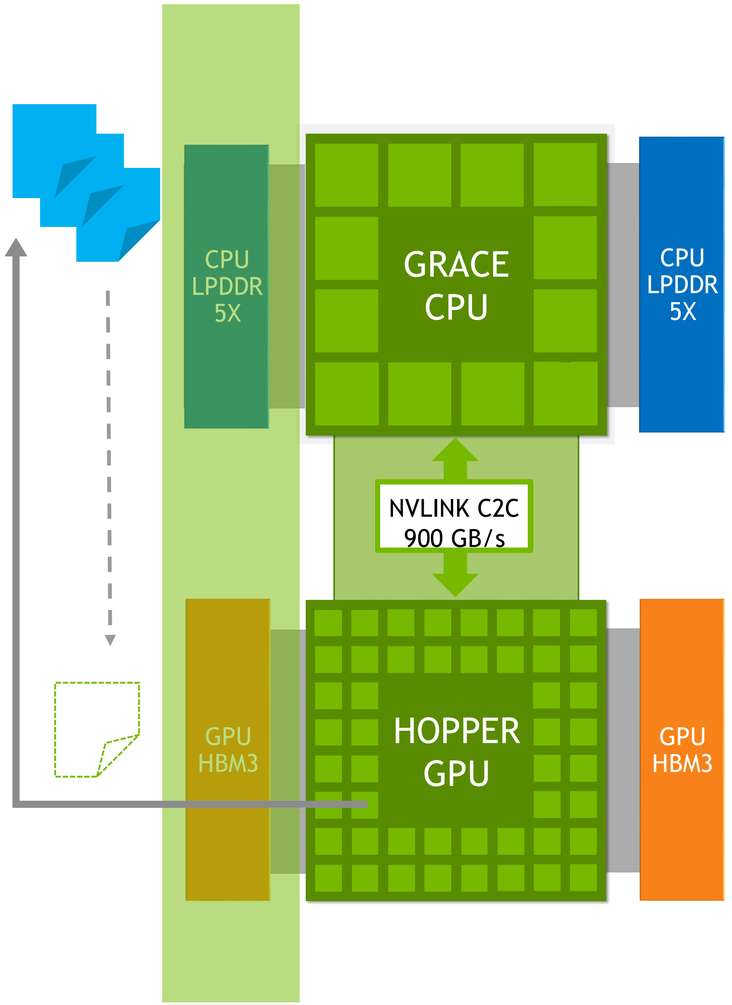
```
__global__ void kernel(int *data)
{
    data[threadIdx.x] += 2;
}
int main()
{
    int N = 128;

    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);
}
```



Memory Allocators

CUDA Managed Memory (cudaMallocManaged)

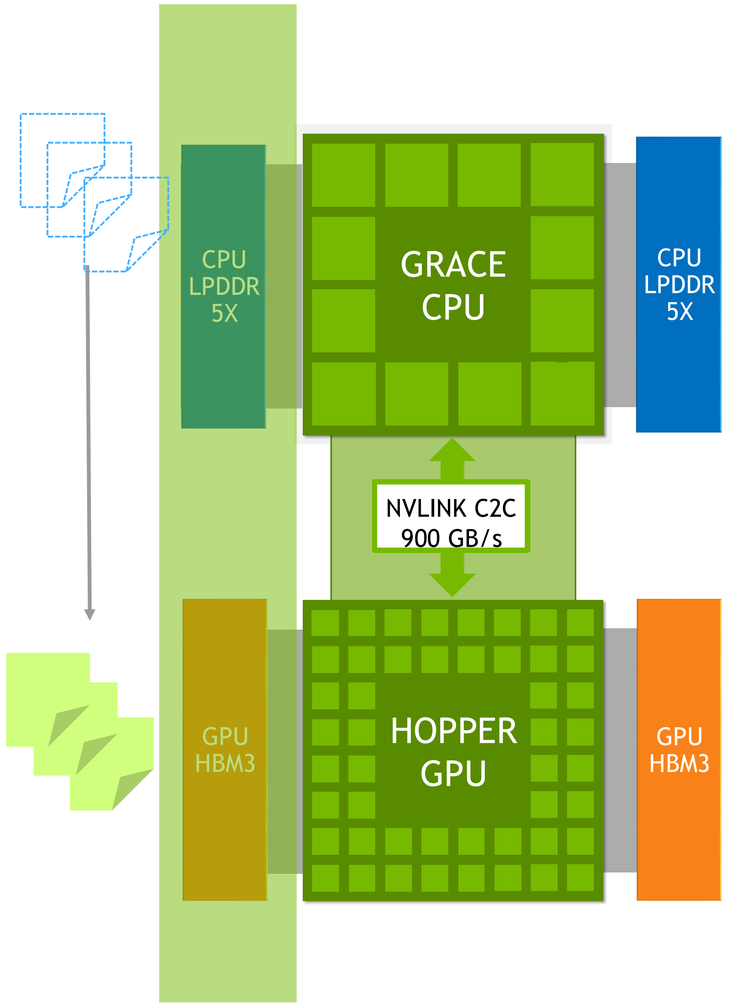
```
__global__ void kernel(int *data)
{
    data[threadIdx.x] += 2;
}
int main()
{
    int N = 128;

    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);
}
```



Memory Allocators

CUDA Managed Memory (cudaMallocManaged)

```
__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;
    }

    cudaMemPrefetchAsync(data, N * sizeof(int), 0);

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

    cudaFree(data);
}
```

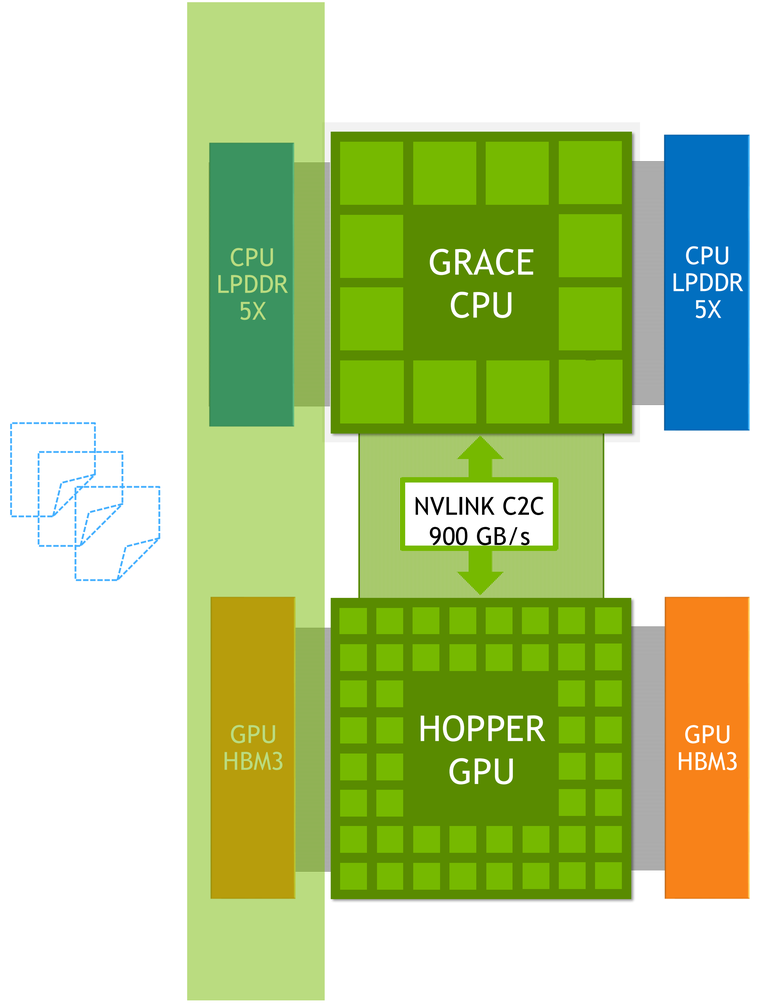


System allocator

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 not yet allocated

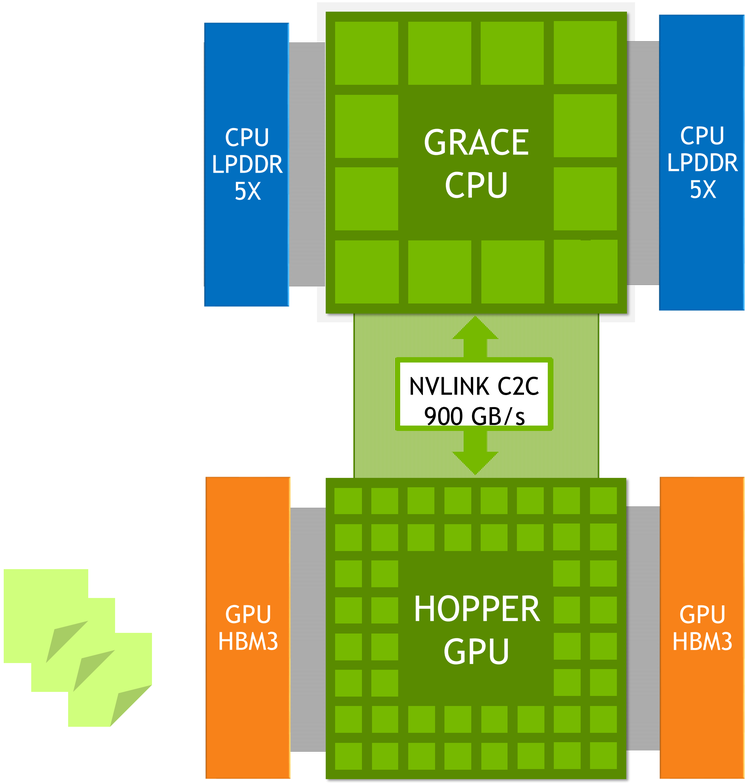


System allocator

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

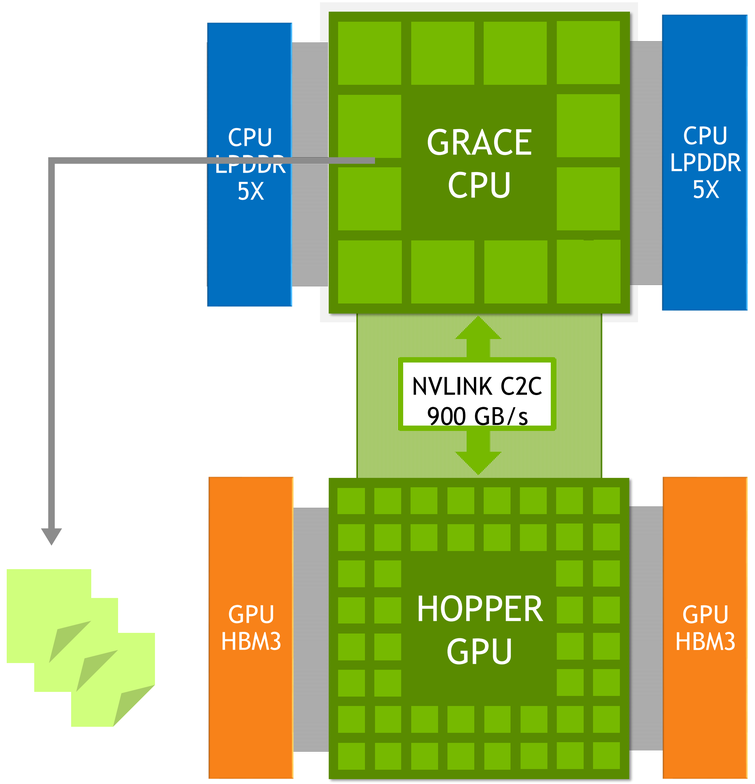


System allocator

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;
}
```

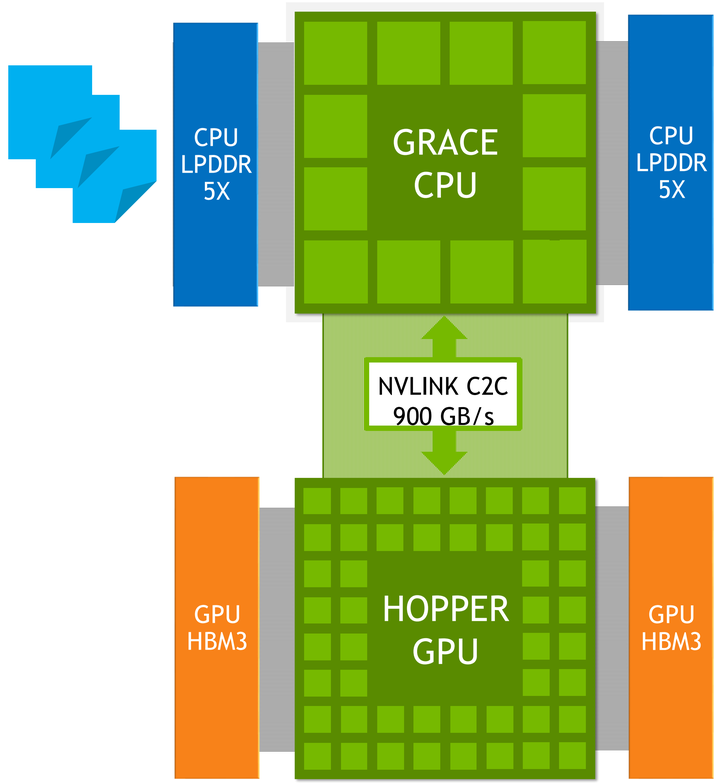
Direct access from CPU to GPU data



CUDA Memory Hints

Tuning memory movements

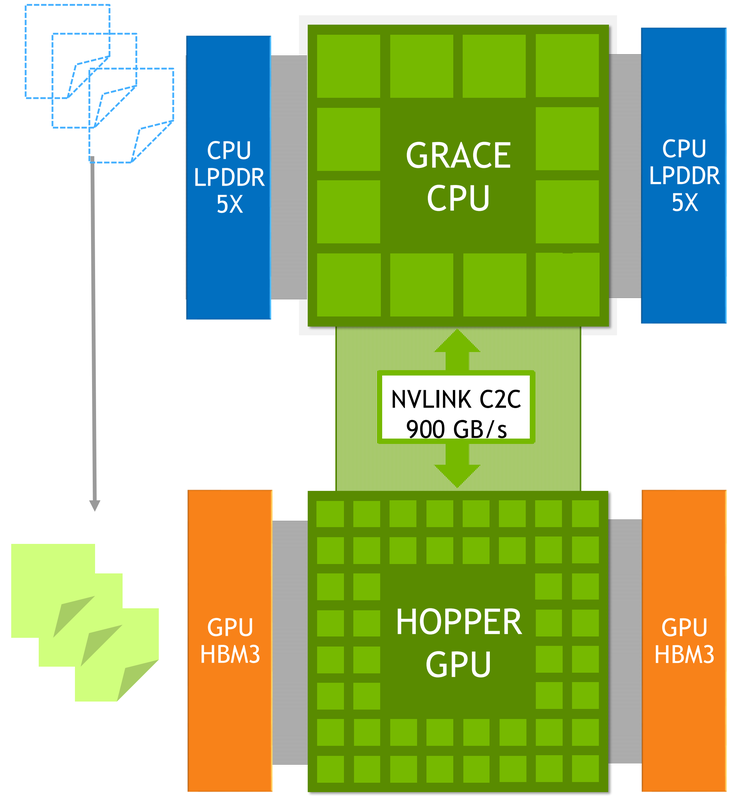
```
int main()
{
    int *data = (int *)malloc(128 * sizeof(int));
    memset(data, 0, 128 * sizeof(int));

    cudaMemPrefetchAsync(data, 128 * sizeof(int), cudaDeviceId, cudaStream);

    kernel<<<1, 128>>>(data);
    cudaDeviceSynchronize();
}
```



CUDA Memory Hints

CUDA Memory APIs or NUMA APIs can be used to tune memory placement and movement

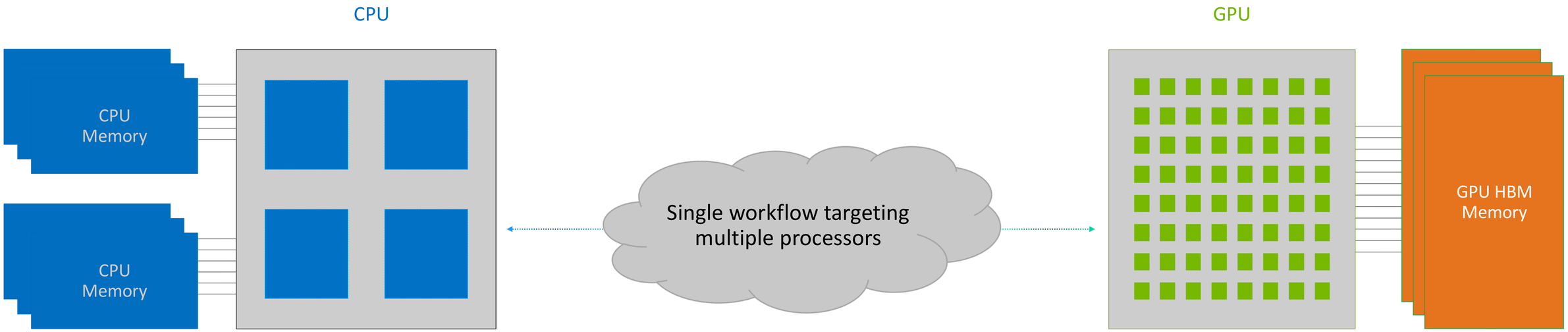
```
int main()
{
    int *data = (int *)malloc(128 * sizeof(int));
    memset(data, 0, 128 * sizeof(int));

    cudaMemPrefetchAsync(data, 128 * sizeof(int), cudaDeviceId, cudaStream);

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

}
```



Applications on Grace Hopper

Heterogeneous Computing

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

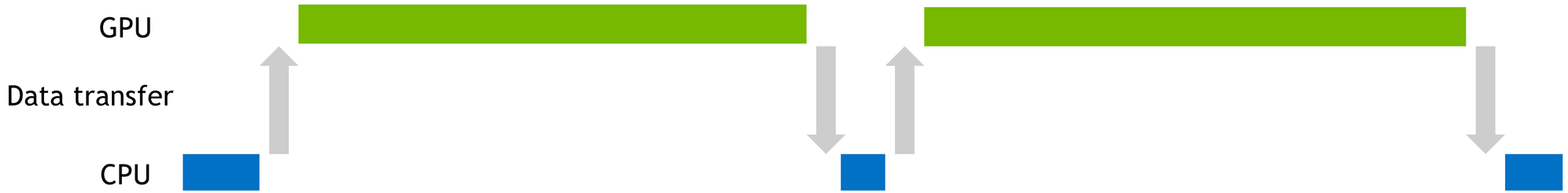


- Categorize applications into 3 categories
  - *Fully GPU Accelerated*
  - *Partially GPU Accelerated*
  - *Coherently GPU Accelerated*

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

Application on Accelerated Systems

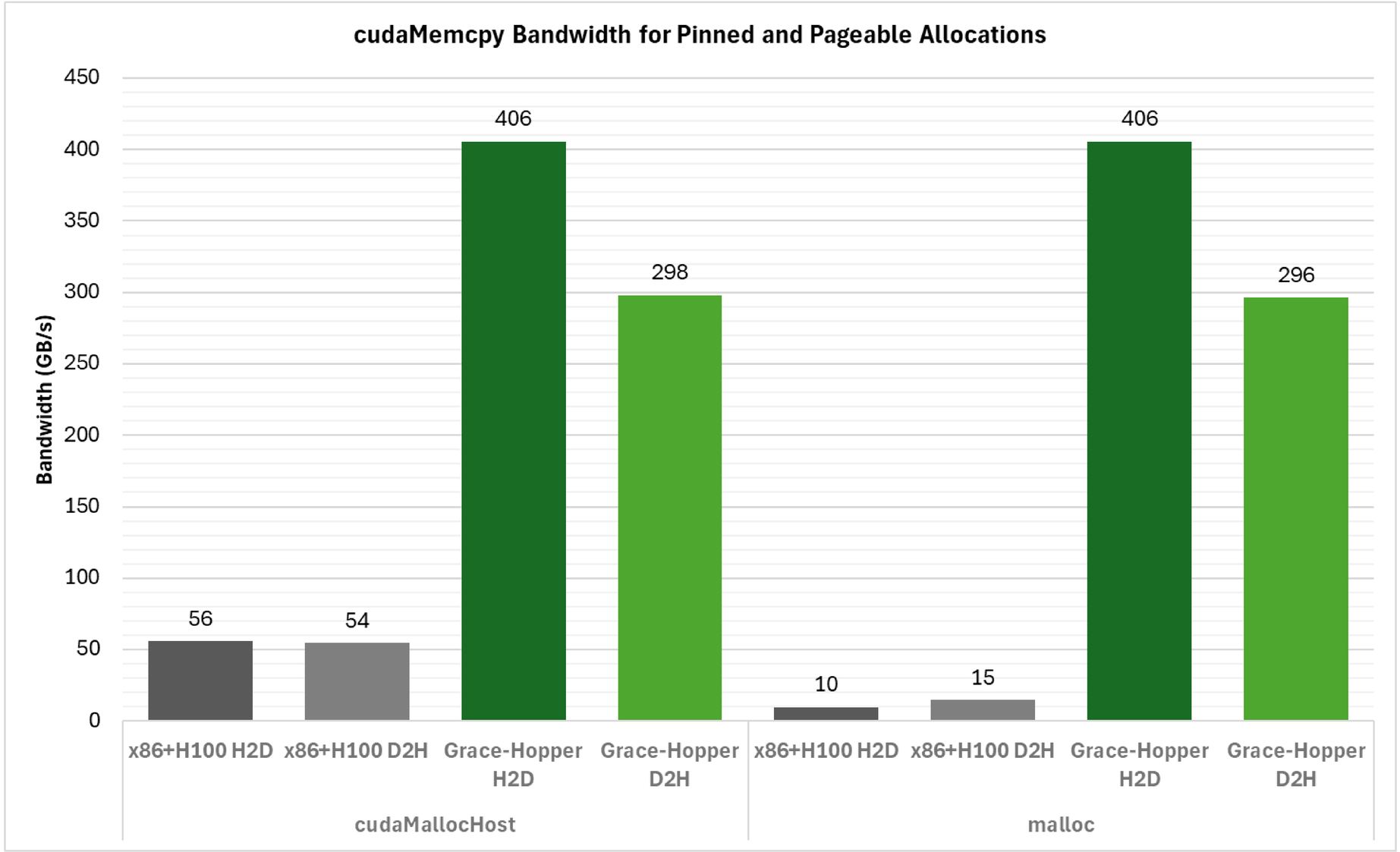
Partially GPU Accelerated

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

- mostly CPU limited

Performance of cudaMemcpy

x86+Hopper vs Grace-Hopper (480 GB)

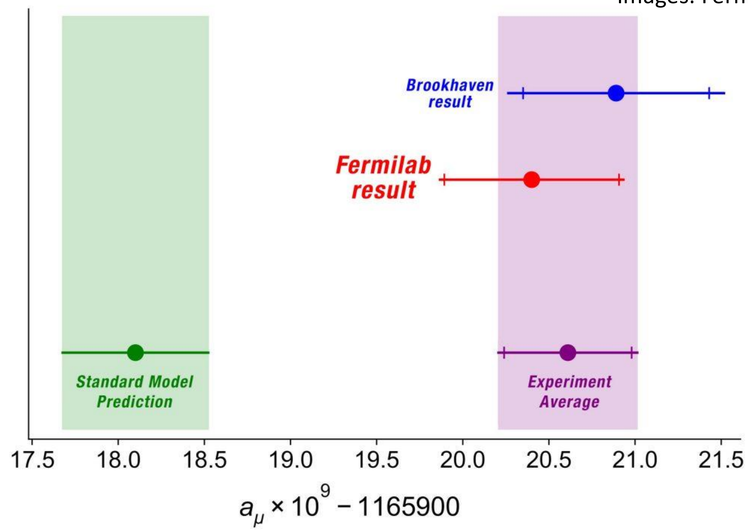


Muon anomalous magnetic moment

New physics?

Images: Fermilab, CERN

- New physics if deviations from Standard Model of Elementary Particle and Nuclear Physics
  - longstanding 3+ standard deviation difference between theory and experiment
  - new experiment was carried out at Fermilab with significantly lower error

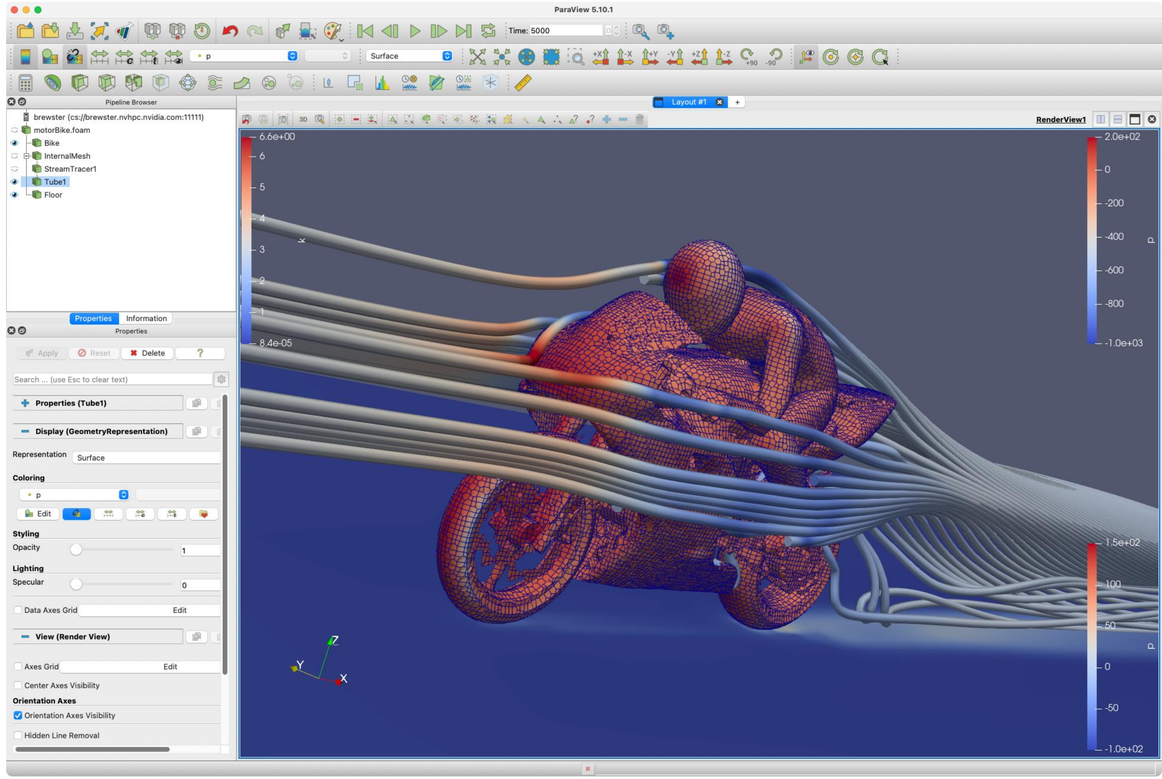


Franz Gross et al., Eur.Phys.J.C 83 (2023), 1125
T. Aoyama et al.Phys.Rept. 887 (2020), 1-166
Fermilab Lattice, HPQCD, and MILC Collaborations, Bazavov et al, Phys.Rev.D 107 (2023) 11, 114514

Computational Fluid Dynamics

Openfoam

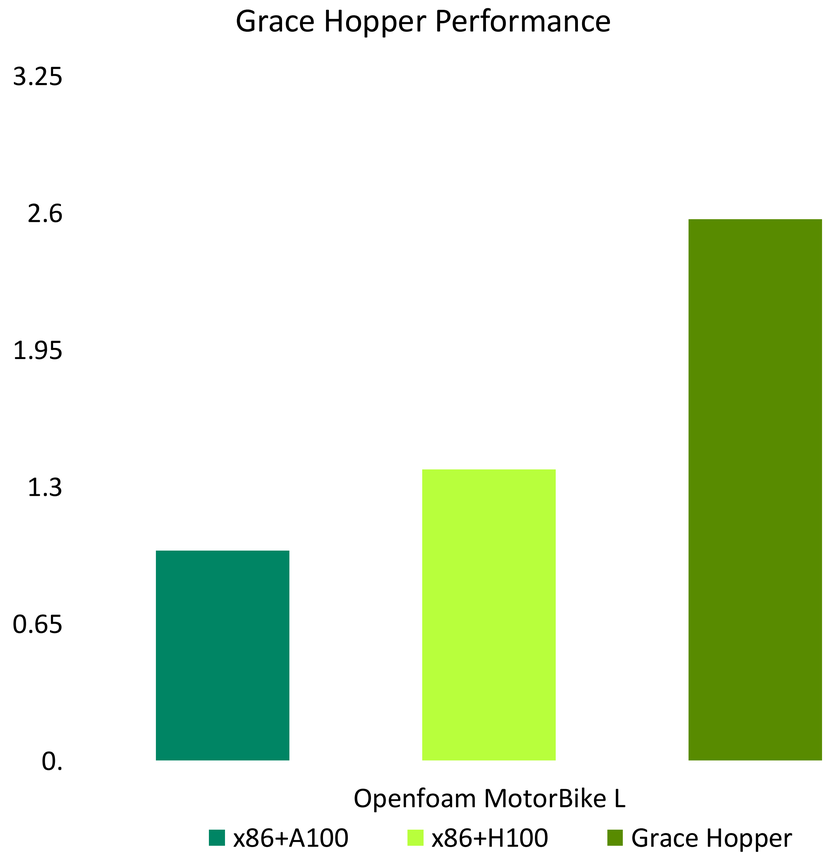
- Computational fluid dynamics (CFD) toolbox
  - Developed by ESI-OpenCFD
- Popular in automotive and other engineering sectors
- Highly configurable fluid flow solvers with turbulence / heat transfer / etc.
- Can leverage GPU-accelerated AmgX linear solvers via plugin interface (PETSc4FOAM)



OpenFoam

Partially GPU Accelerated – mostly CPU limited

- HPC motorbike problem (Large)
  - Solves with the simpleFoam application
  - Around 30-40% of CPU-only execution is spent in linear solves
  - Hybrid approach spends large proportion of time on the CPU
- Performance on Grace Hopper
  - High CPU and GPU memory bandwidth improve compute performance
  - C2C bandwidth minimizes the cost of migrating CPU matrix data



~35M cells benchmark designed by OpenFOAM HPC technical committee

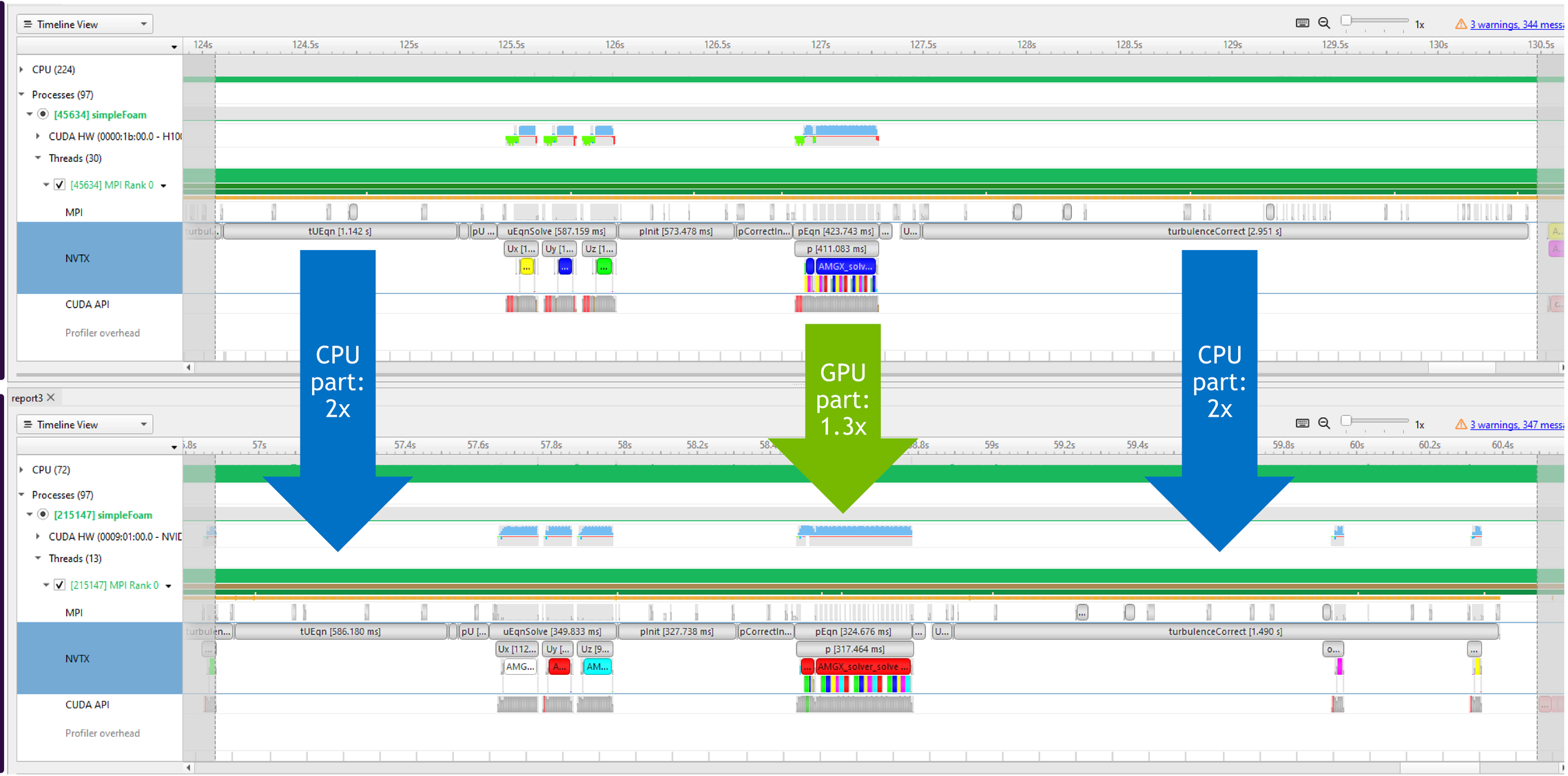
A100 runs were done using AMD EPYC (Rome) CPUs.
H100 runs were done using Intel Xeon (SPR) CPUs.

OpenFoam

Nsight Systems Profile

x86 + H100 (H100 80GB HBM3)

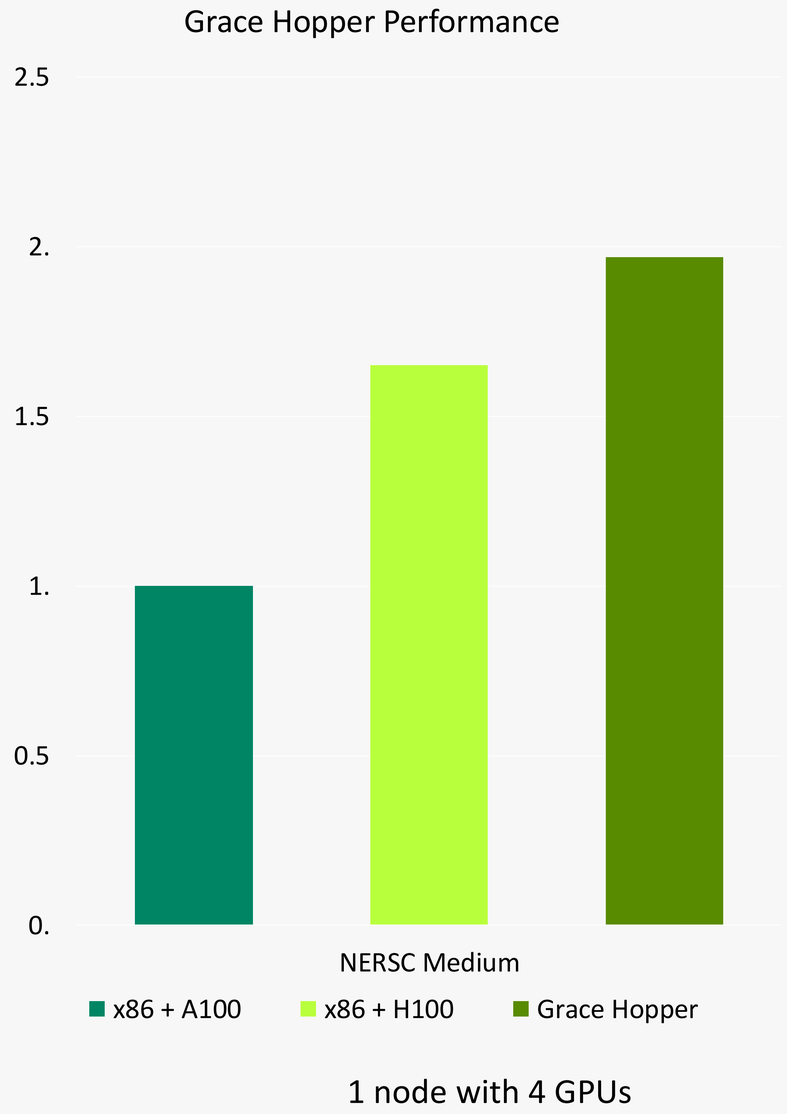
Grace Hopper (H100 96GB HBM3)



Lattice QCD with MILC

Fully accelerated using QUDA

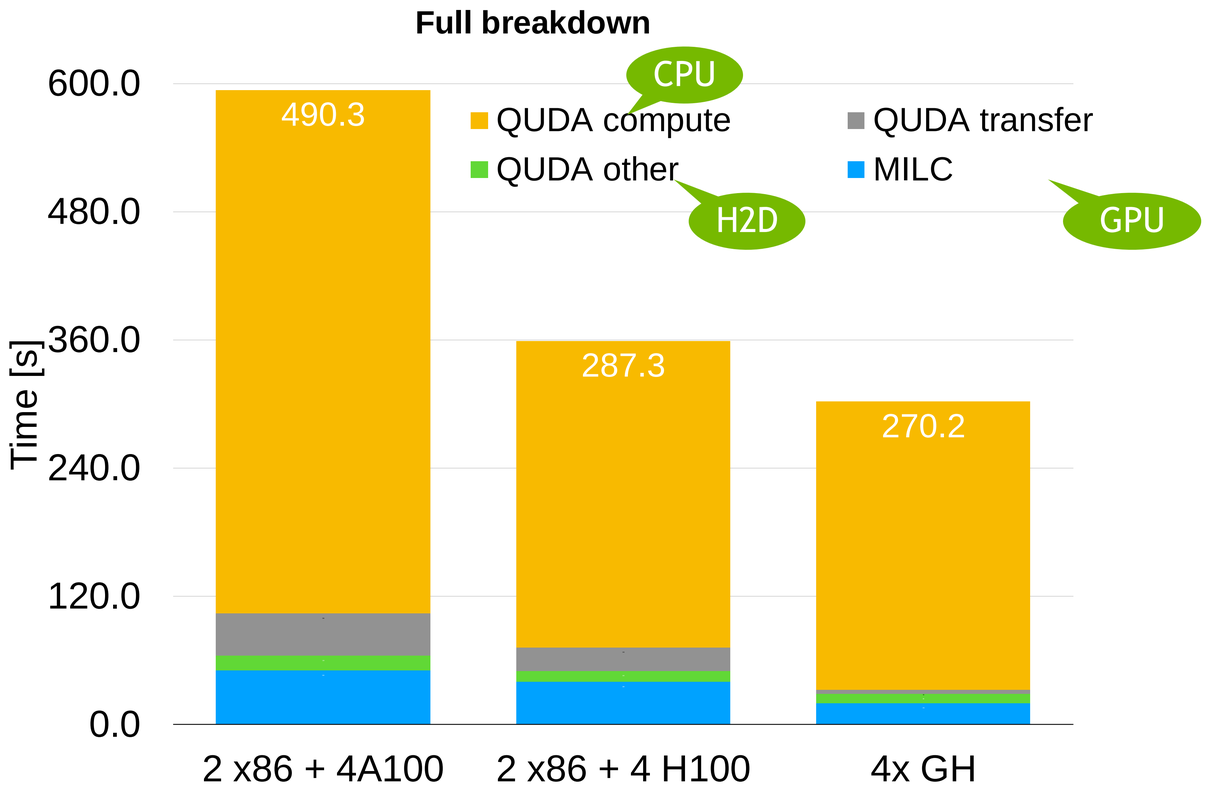
- GPU offload acceleration through QUDA with partial GPU data residency
- Multiple steps in workflow
  - **Generate so-called gauge configurations (requires strong-scaling as generated in a Markov chain)**
  - Calculate physical properties using these configurations as input (throughput problem)
  - Analysis
- **NERSC Medium benchmark (proxy, realistic scale O(100-1000) GPUs)**
- Performance on Grace-Hopper ensures 2x scaling over x86 +A100
  - C2C drastically reduces data-transfer time
  - Grace CPU memory bandwidth accelerates remaining CPU parts
  - Both combined restore scaling between generations

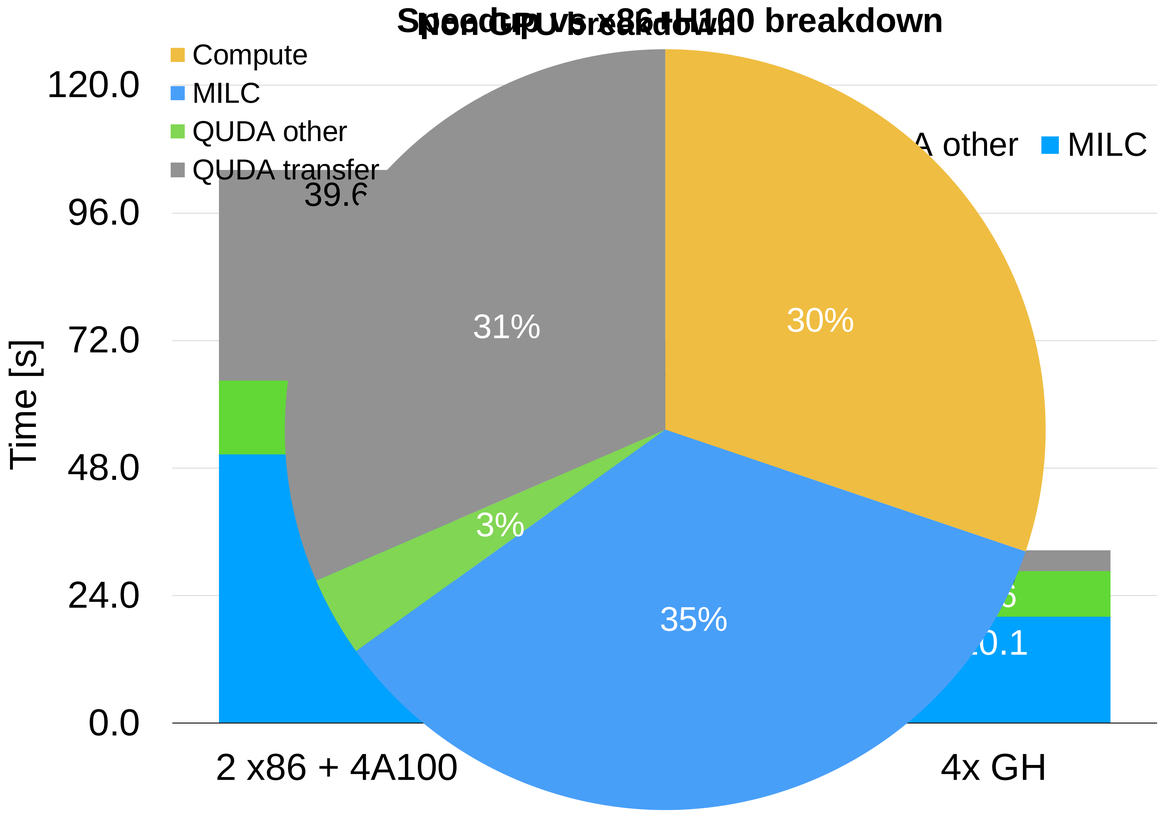


A100 runs were done using AMD EPYC (Rome) CPUs.
H100 runs were done using Intel Xeon (SPR) CPUs.

MILC

Breakdown



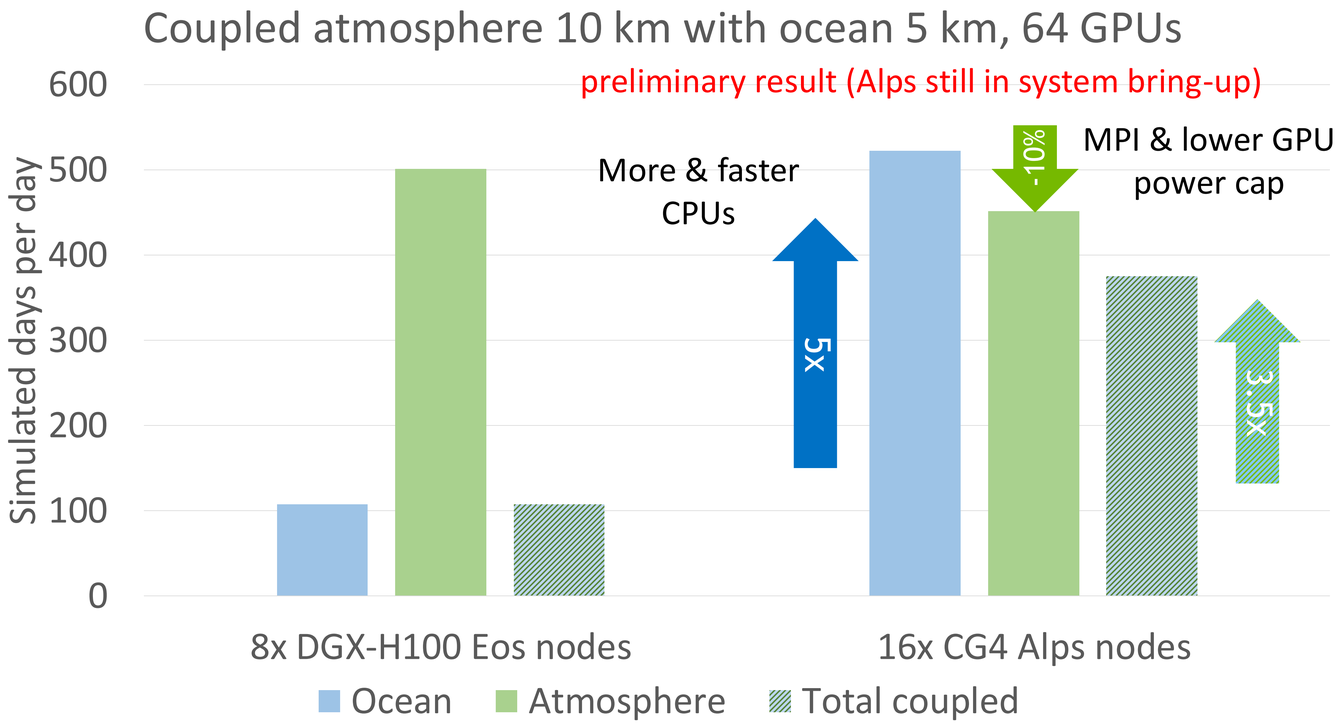


A100 runs were done using AMD EPYC (Rome) CPUs.
H100 runs were done using Intel Xeon (SPR) CPUs.

ICON Coupled Ocean

Grace-Hopper: 3.5x speedup

- On EOS:
  - Performance limited by Ocean running on the CPU
- On Alps:
  - Unleash full performance of Hopper GPUs
  - Grace is powerful enough to run the ocean in the background
  - Alps network is still in bring-up phase, which introduces some atmosphere-only and coupling overhead
  - 3.5x end-to-end performance



See also: Thomas Schulthess, CSCS – S62157

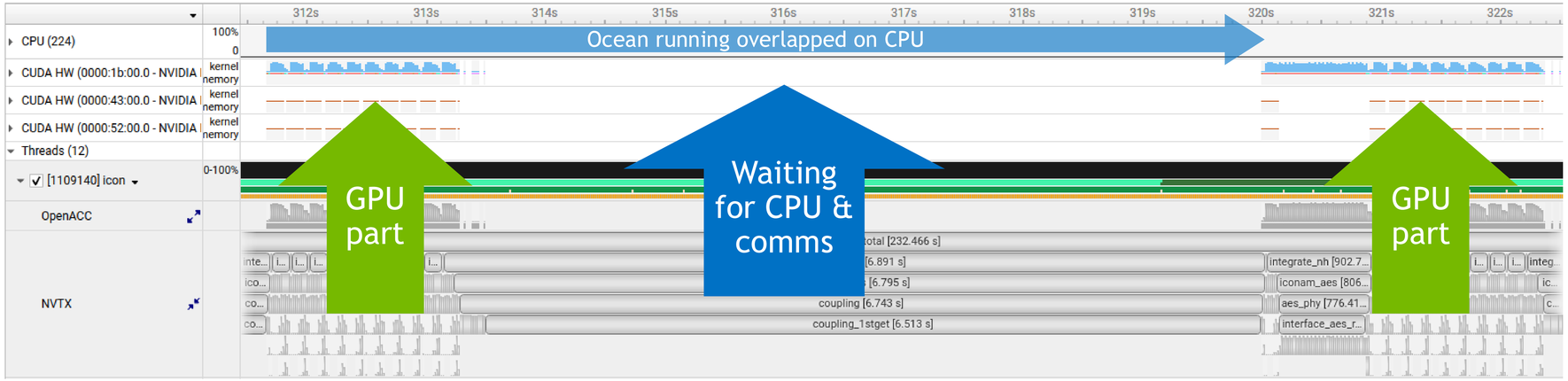
ICON Coupled Ocean

Profile

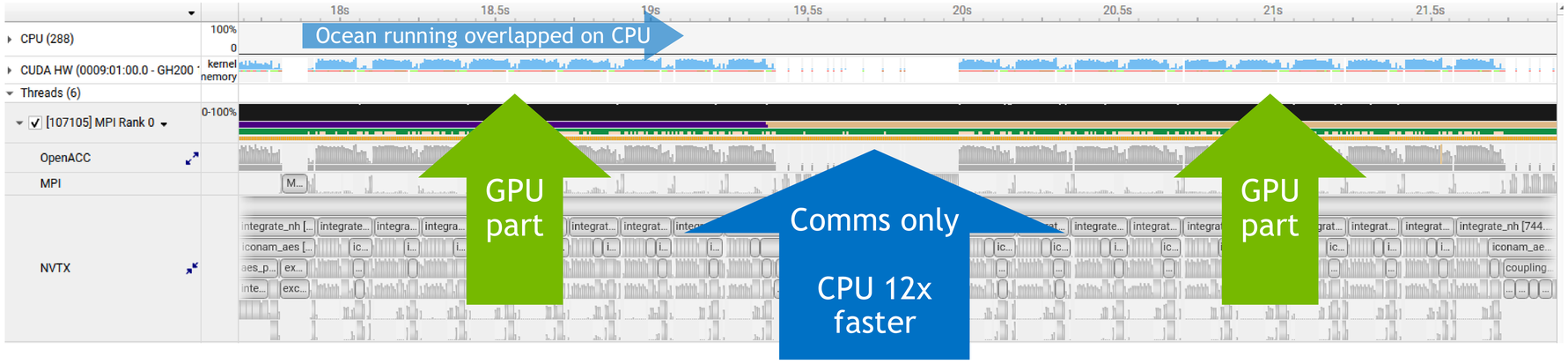
- Full globe coupled simulation at 10 km atmosphere resolution and 5 km ocean resolution. 90 vertical atmosphere layers, 72 vertical ocean layers. Atmosphere time-step is 90s, ocean time-step is 5 min and coupling time-step is 15 min. Atmosphere and ocean run in different ranks within the same MPI job. 64 GPUs and 512 (Eos) or 3008 (Alps) CPU ranks

Profiling atmosphere GPU process

Eos



Alps



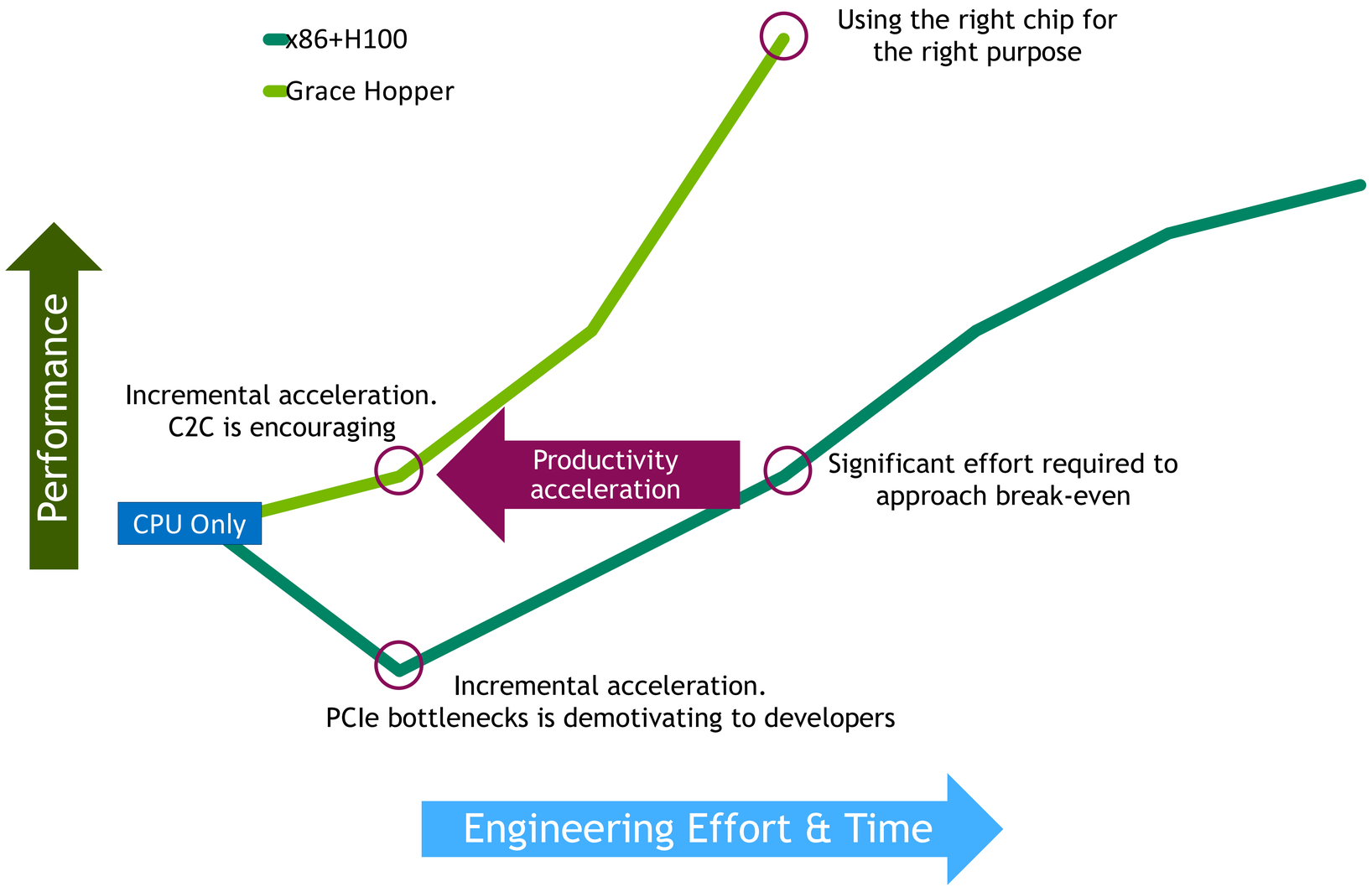
Application on Accelerated Systems

Coherently GPU Accelerated

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

Developer Velocity with Grace Hopper

Accelerating the path to accelerated computing



NEMO Ocean Model

A partially accelerated case utilizing unified memory on Grace-Hopper

The "**N**ucleus for **E**uropean **M**odelling of the **O**cean" (**NEMO**) is a state-of-the-art modelling framework, used for research activities and forecasting services in ocean and climate sciences.

- **Setup (NEMO v4.2.0)**
  - **GYRE_PISCES** benchmark
    - Scaling factor for grid resolution: **nn_GYRE = 25**
      - ~ORCA ½ grid
      - ~80 GB RAM, fits on single GPU
  - **MPI-only**, single core to every MPI process for CPU runs

- **Incremental porting** on Grace-Hopper **(480GB)** using unified memory and access-counter based migrations
  - Memory management left to runtime – **system-allocated memory with automatic migrations**
    - compile with `-gpu=unified,nomanaged`
  - Simply offloading loops to GPU using **OpenACC**, in 3 steps:
    - **Horizontal (lateral) diffusion,**
    - **Advection,**
    - **Vertical diffusion and time-filtering,**

    for both “active” (**TRA**) and “passive” (**TRC**) tracer transport

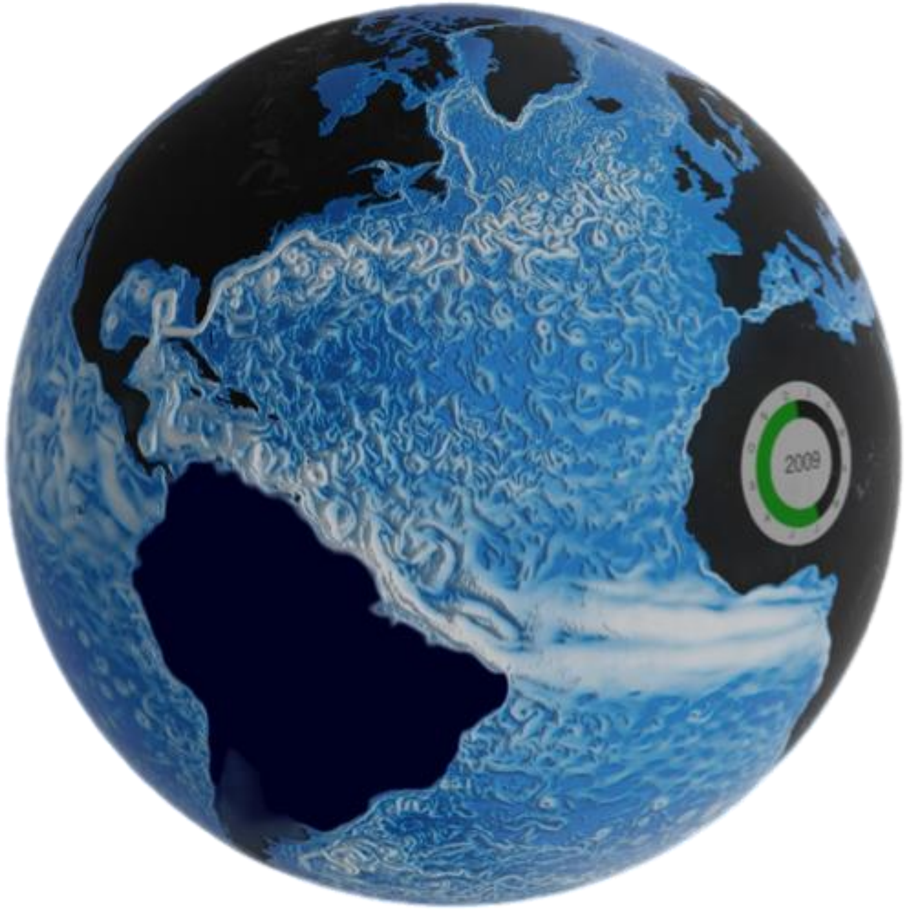


Image source:
NEMO User Guide — NEMO release-4.2.2 documentation (nemo-ocean.io)

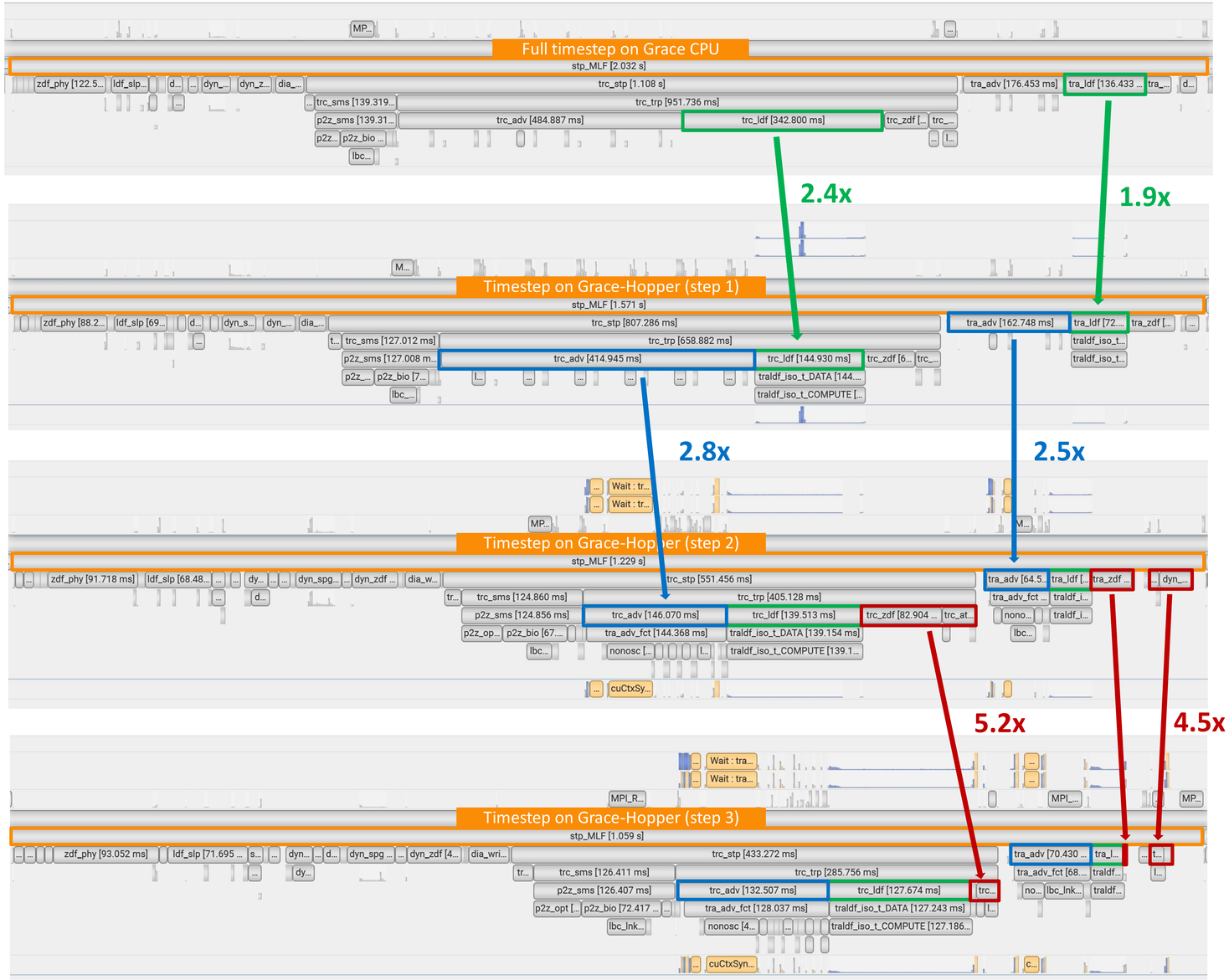
Porting NEMO to Grace-Hopper using Unified Memory

Incremental porting, zooming in to a single timestep …

Ported to GPU:

- **Horizontal diffusion**
- **Advection**
- **Vertical diffusion and time-filtering**

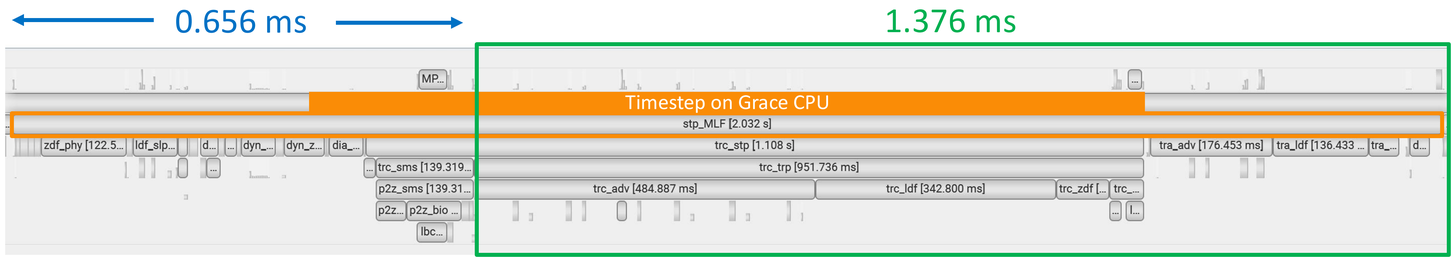
We run multiple (i.e. 40) MPI processes on **CPU and GPU** using **MPS**, and use **“migratable”** system allocated memory



Porting NEMO to Grace-Hopper using Unified Memory

A deeper look into the effect of access-counter based migrations on the partially accelerated port

CPU only run

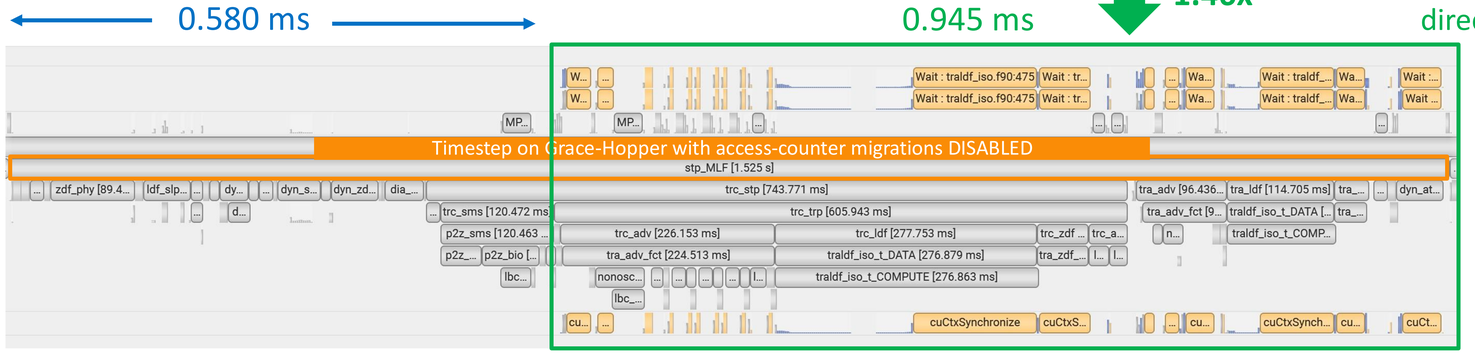


1.46x

1.33x

GPU kernels pull data first-touched by CPU directly from **CPU memory**

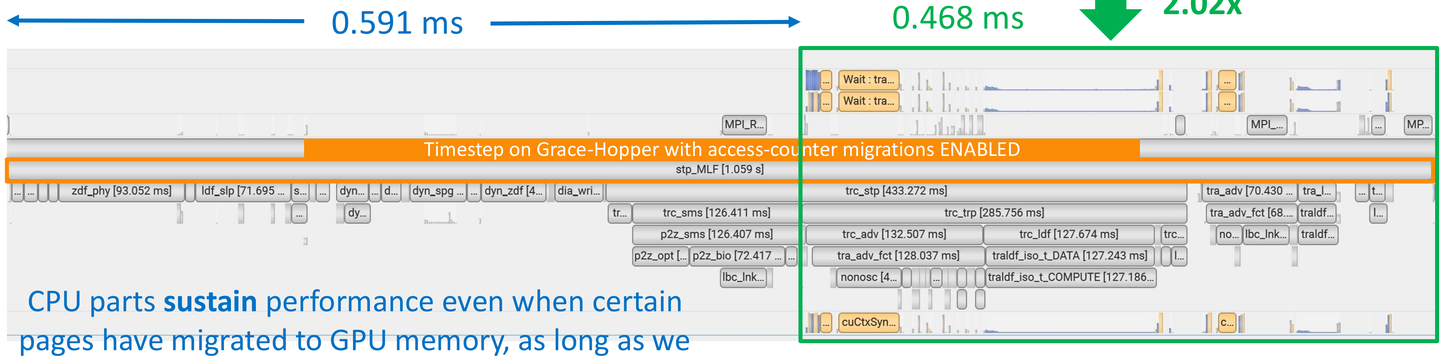
Tracer transport on GPU with migrations disabled
(buffers with first-touch on CPU will never migrate to GPU memory)



2.02x

1.44x

Enabling automatic page migrations from CPU to GPU
(“hot” pages migrate to GPU)



CPU parts **sustain** performance even when certain pages have migrated to GPU memory, as long as we use enough processes to saturate C2C BW

GPU kernels become faster as more and more pages migrate to GPU

Getting ready for Grace-Hopper

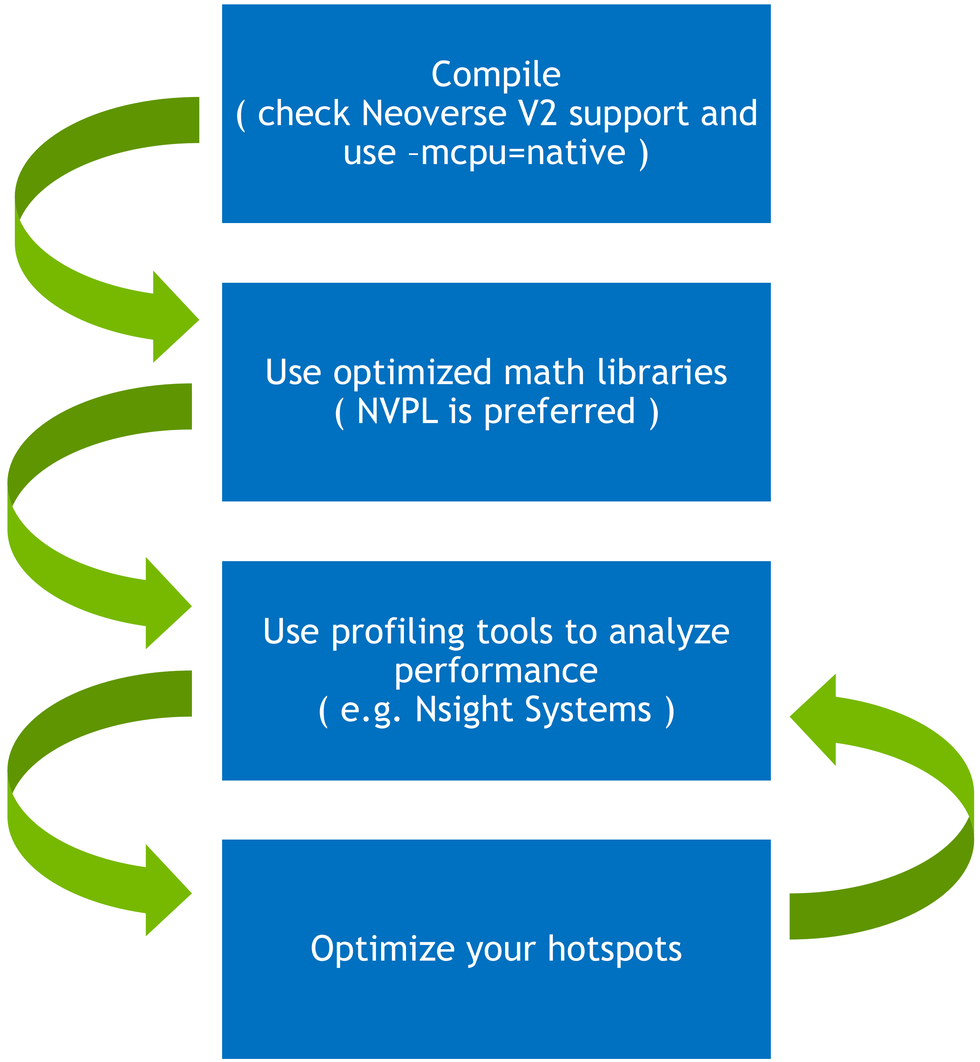
Recompile and Run

- Currently existing applications do not need to be changed
  - Recompile the application for ARM Neoverse-V2 (Grace) and sm_90 (Hopper)*
  - Benefit from more bandwidth everywhere

- Accelerate existing applications
  - Easier to port than ever
  - Large selection of programming models and language available
  - Large selection of tools (NVIDIA tools and 3rd party) available
  - Balanced architecture results in fewer Amdahl’s limiters
  - Hardware coherency
  - Obtain overall speedup even for partially ported applications with the Grace CPU and C2C

Porting to Grace CPU

Most applications will recompile easily and work “out of the box”



Compiler	Version ≧
GCC	12.2
LLVM (Clang)	16
NVIDIA HPC	23.3
Arm Compiler	23.04

NVPL

```
gcc -DUSE_CBLAS -ffast-math -mcpu=native -O3 \
    -I/PATH/TO/nvpl/include \
    -L/PATH/TO/nvpl/lib \
    -o mt-dgemm.nvpl mt-dgemm.c \
    -lnvpl_blas_lp64_gomp
```

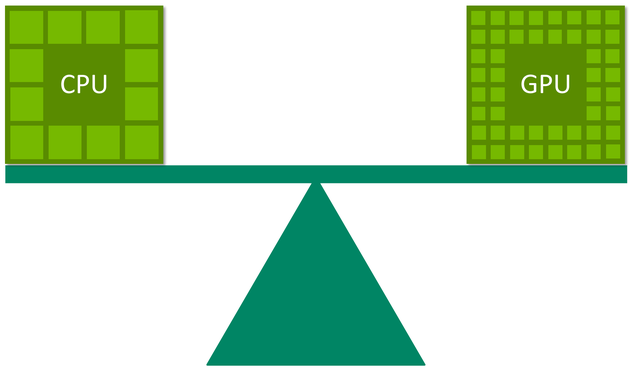
ArmPL

```
gcc -DUSE_CBLAS -ffast-math -mcpu=native -O3 \
    -I/opt/arm/armpl-23.10.0_Ubuntu-22.04_gcc/include \
    -L/opt/arm/armpl-23.10.0_Ubuntu-22.04_gcc/lib \
    -o mt-dgemm.armpl mt-dgemm.c \
    -larmpl_lp64
```

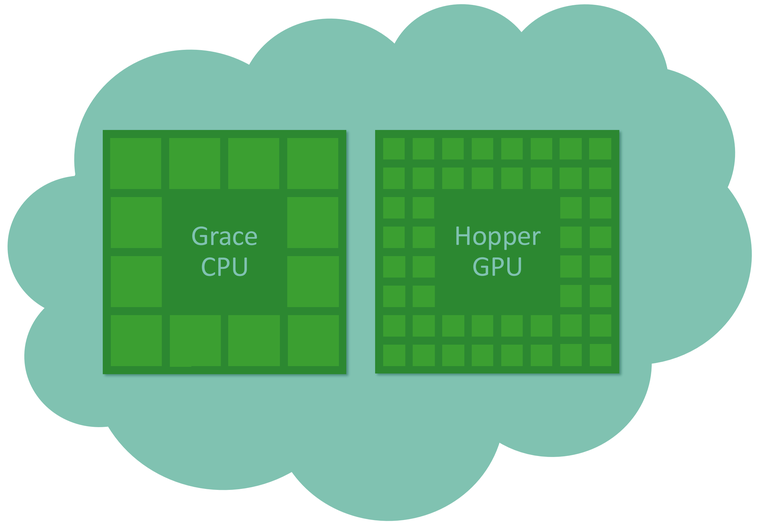
BLAS	LAPACK	PBLAS	SCALAPACK
TENSOR	SPARSE	RAND	FFTW

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