Technological Update

November 2024

NVIDIA CONFIDENTIAL PRESENTATION

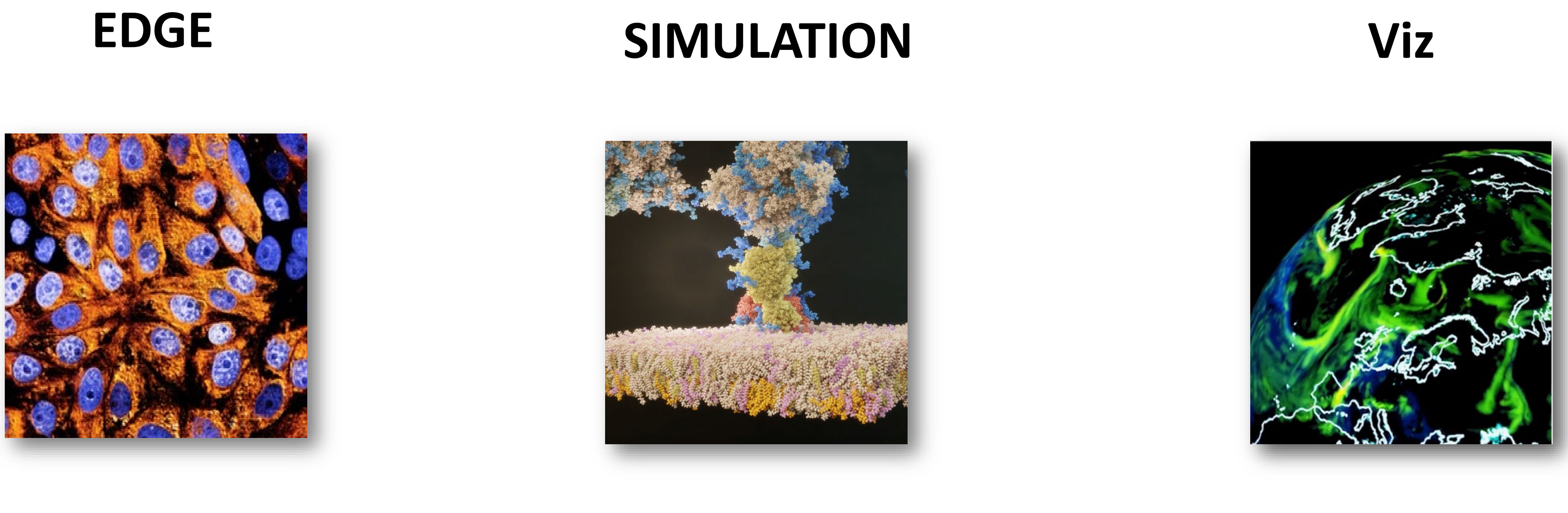
- All information in the following presentation is NVIDIA confidential, including codenames, future products, and performance projections
- No information in this presentation is allowed to be revealed or published without NVIDIA consent
- Sharing or distributing copies of this presentation to anyone is strictly prohibited
- Use of cameras to capture information is strictly prohibited

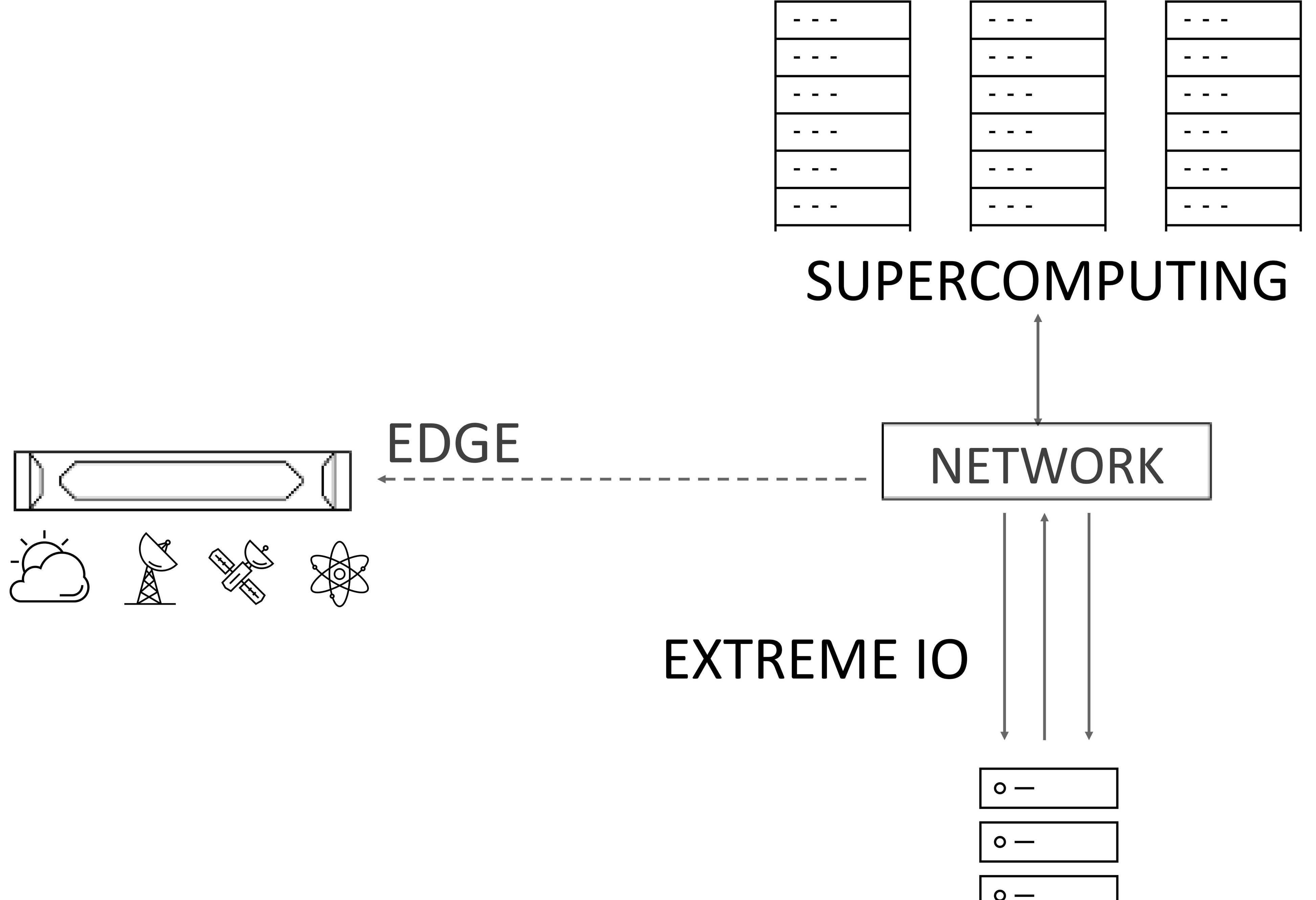
Customer Guidelines

Agenda

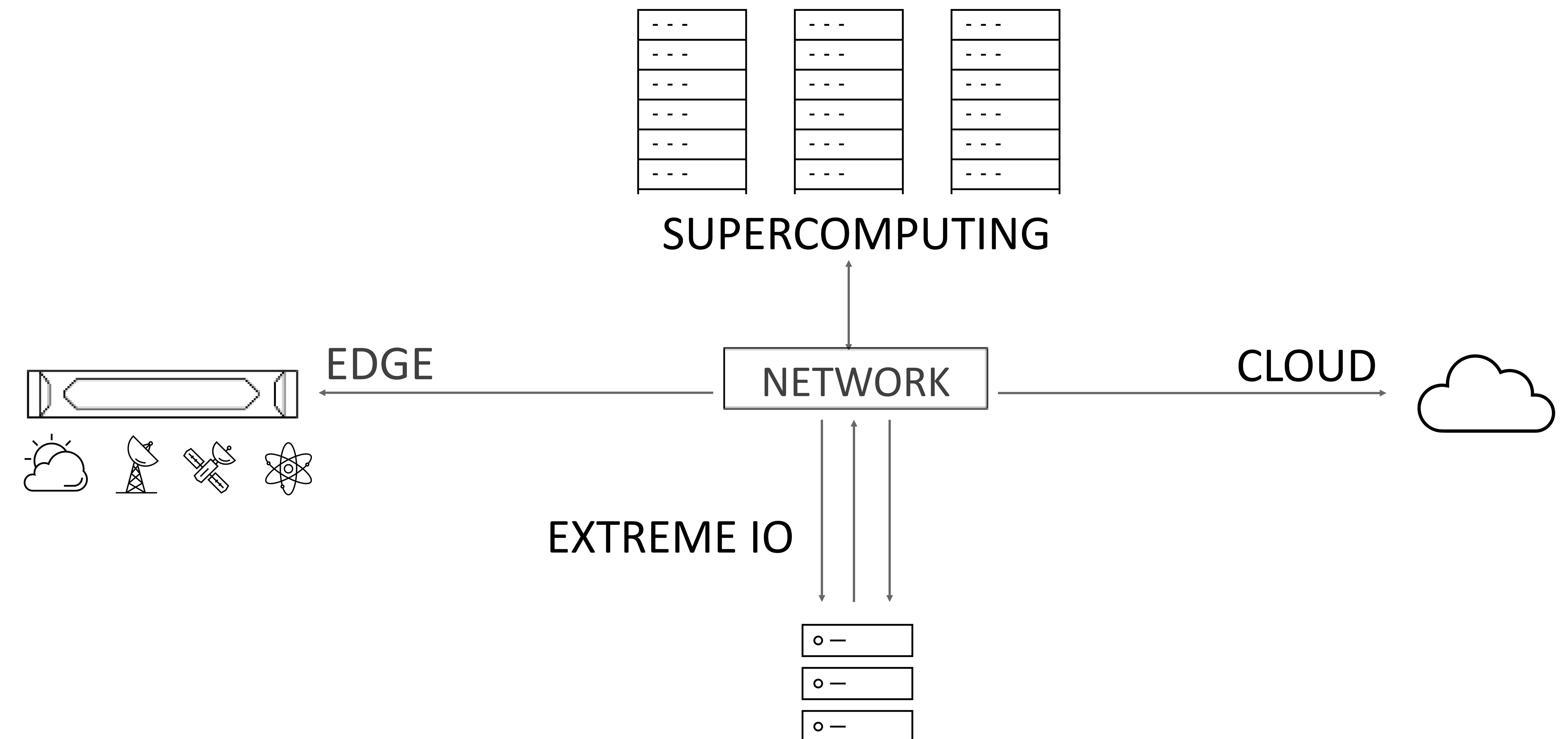
- NVIDIA hardware architecture overview;
- Main differences between the different NVIDIA GPU architectures;
- Overview of the main drivers / software libraries needed for the different hardware components;
- Programming languages available for GPUs;
- Tools for debugging and performance analysis (e.g. Visual/Compute Profiler);
- Main techniques for optimizing code performance;
- Main techniques for optimizing intra-node and inter-node multi-GPU communications with infiniband (e.g. GPUDirect P2P, RDMA); Roadmap of NVIDIA technological developments (for what concerns CPU/GPU);
- References to in-depth courses on the different topics.

Pre-Exascale Supercomputing



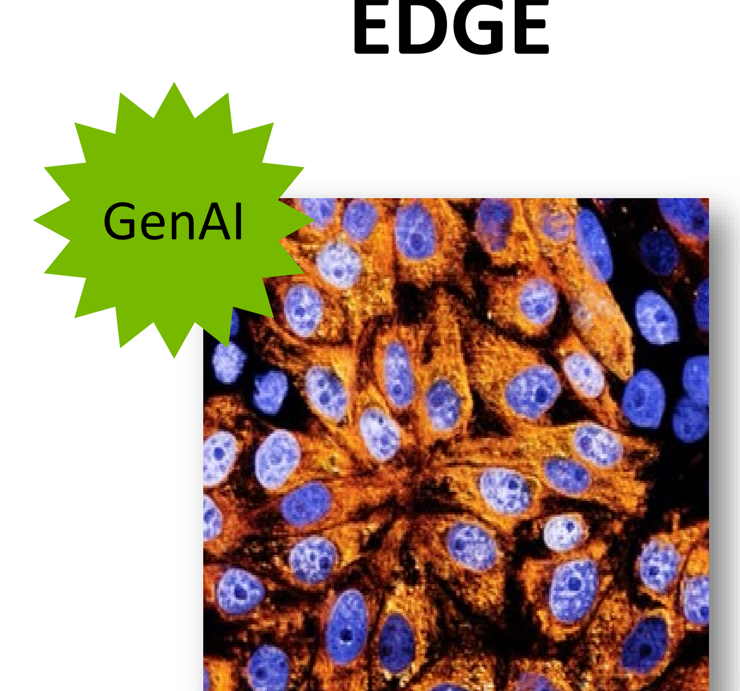


Exascale Supercomputing



AI Factory for Research

EDGE

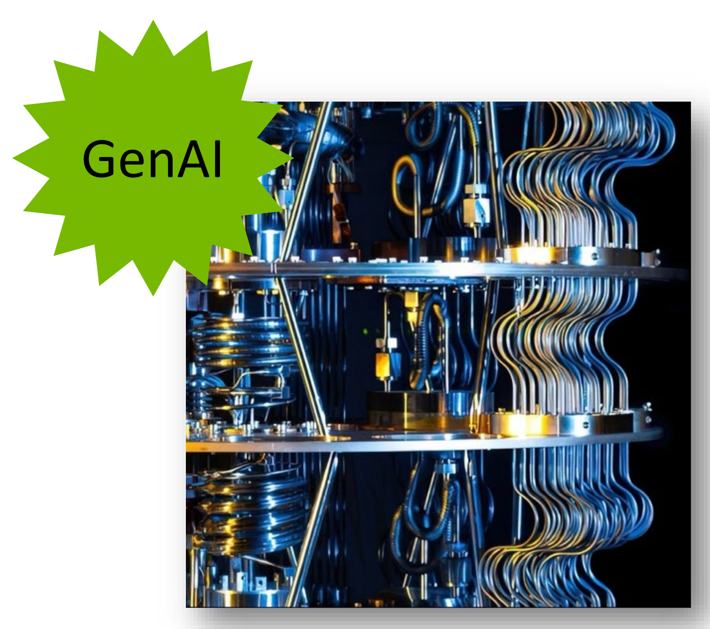


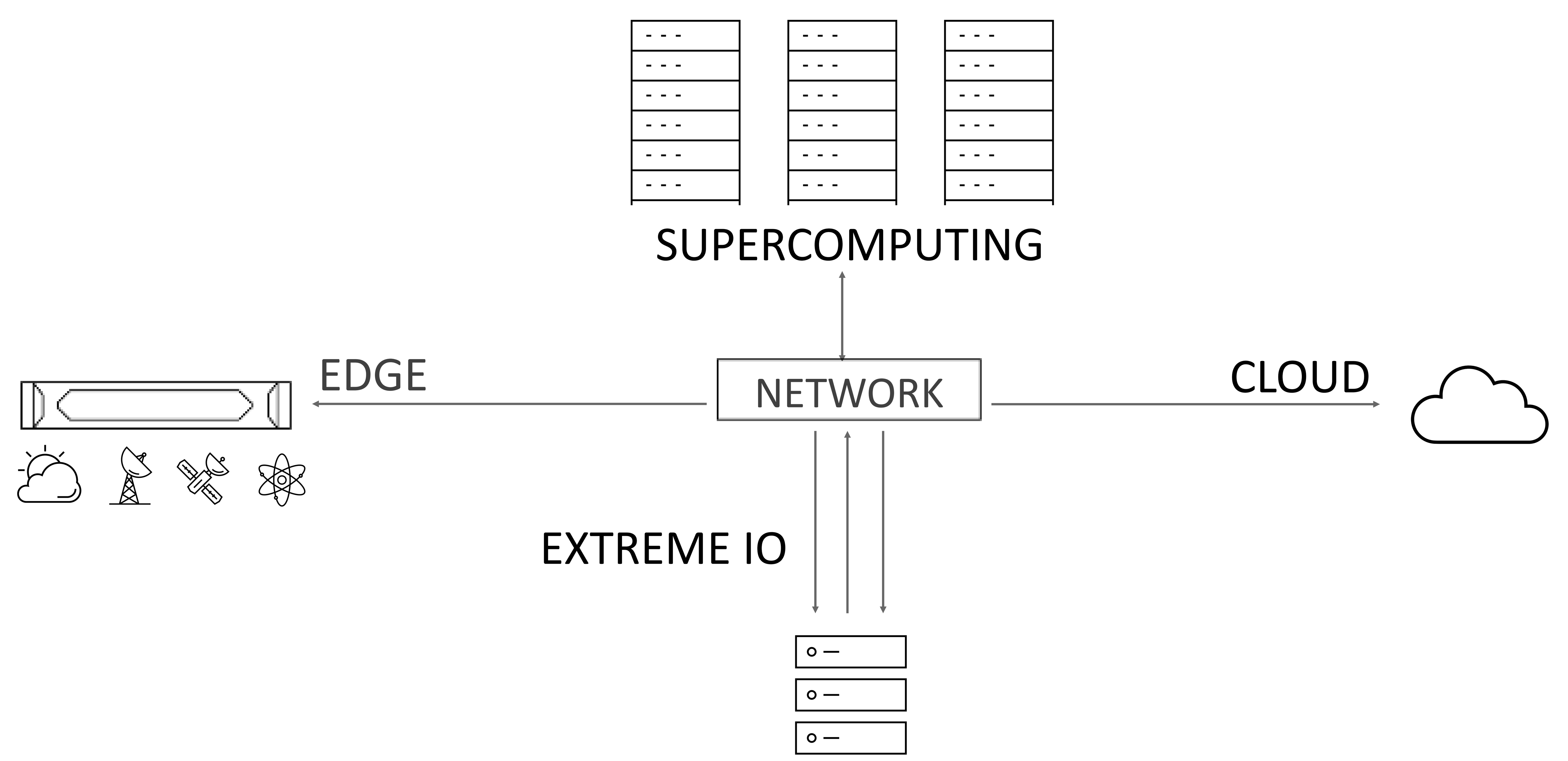
SIM + AI

SIMULATION

DIGITAL TWIN

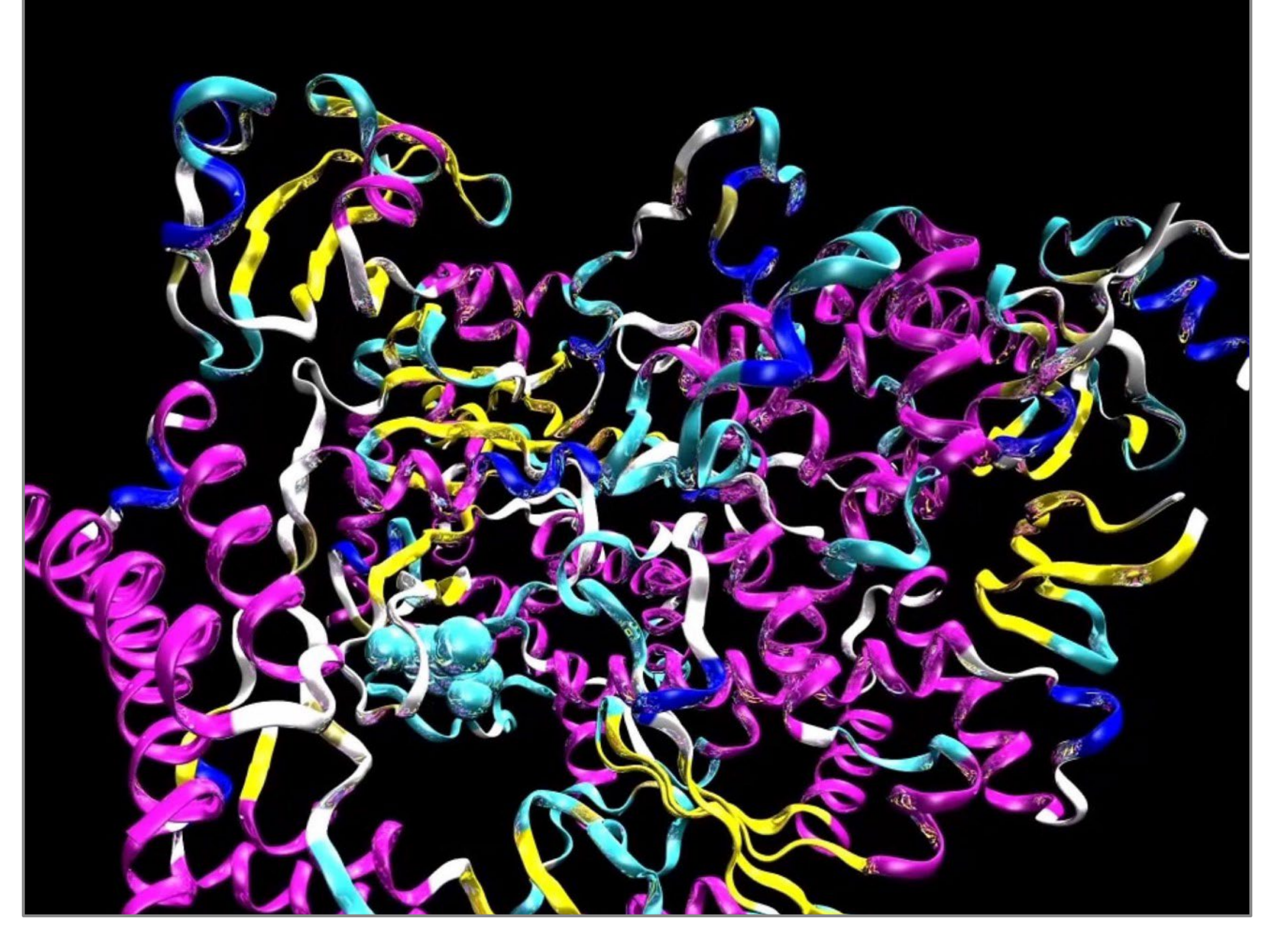
QUANTUM COMPUTING



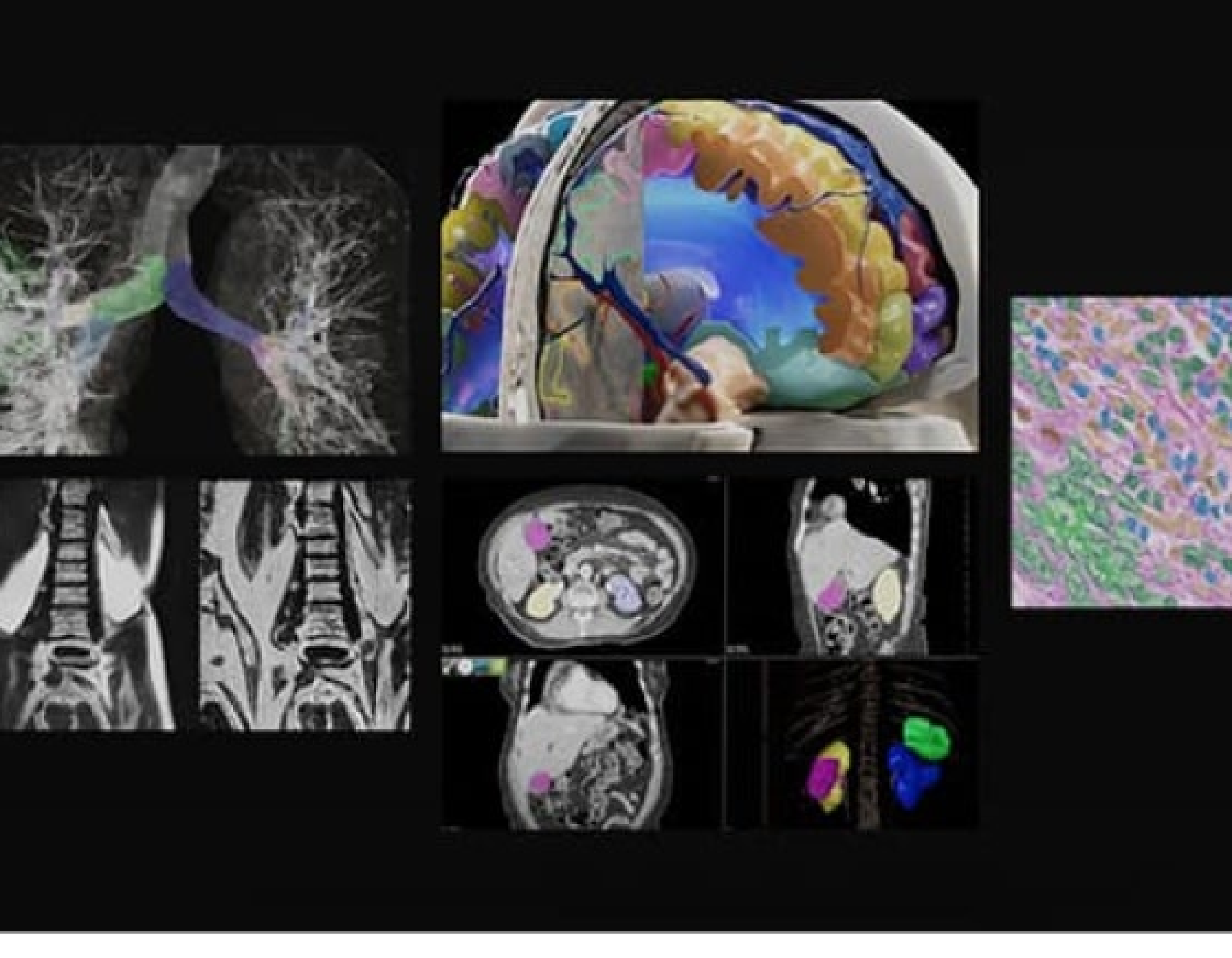


AI: The new tool for Science

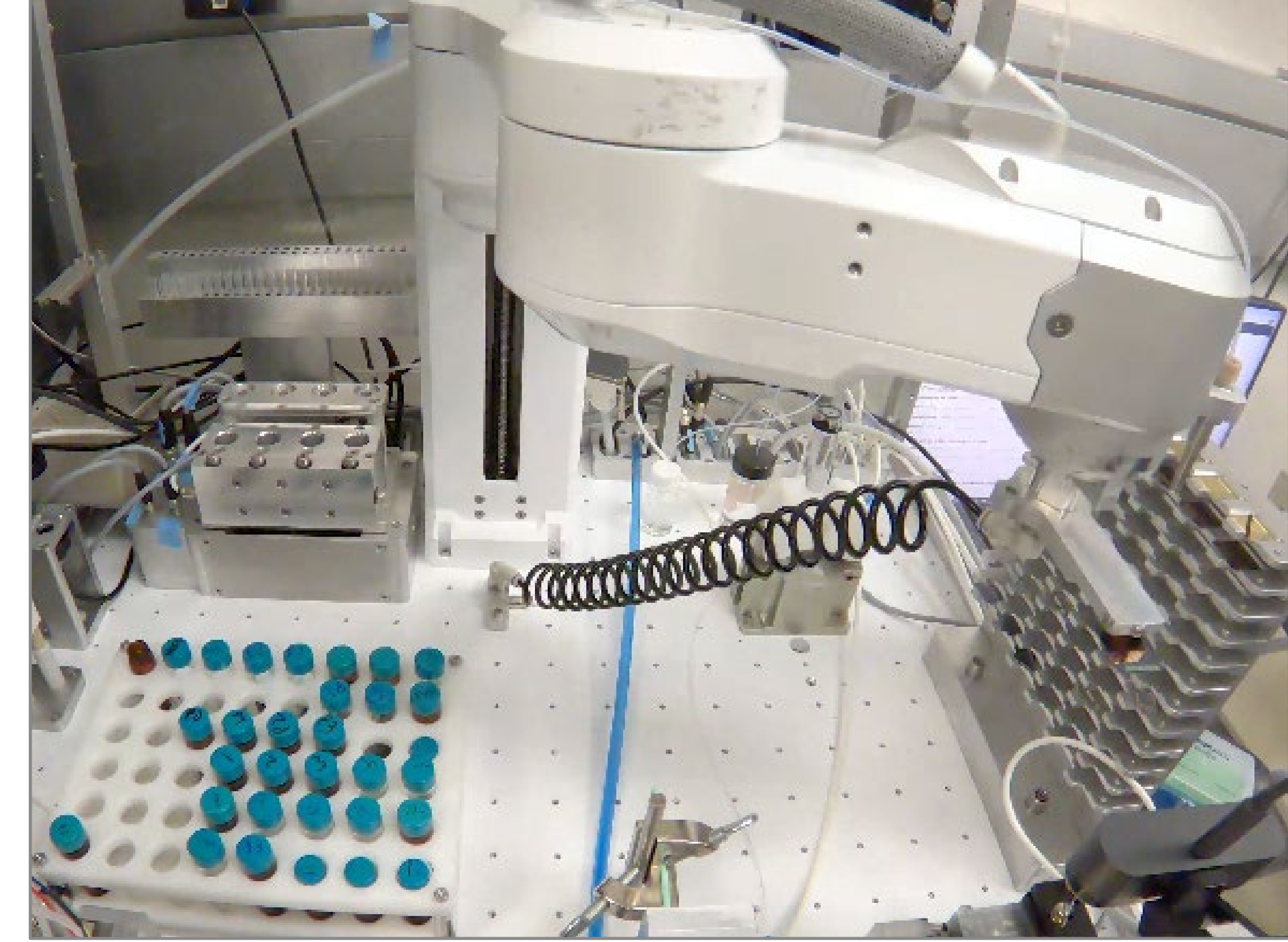
DRUG DISCOVERY
EvolutionaryScale



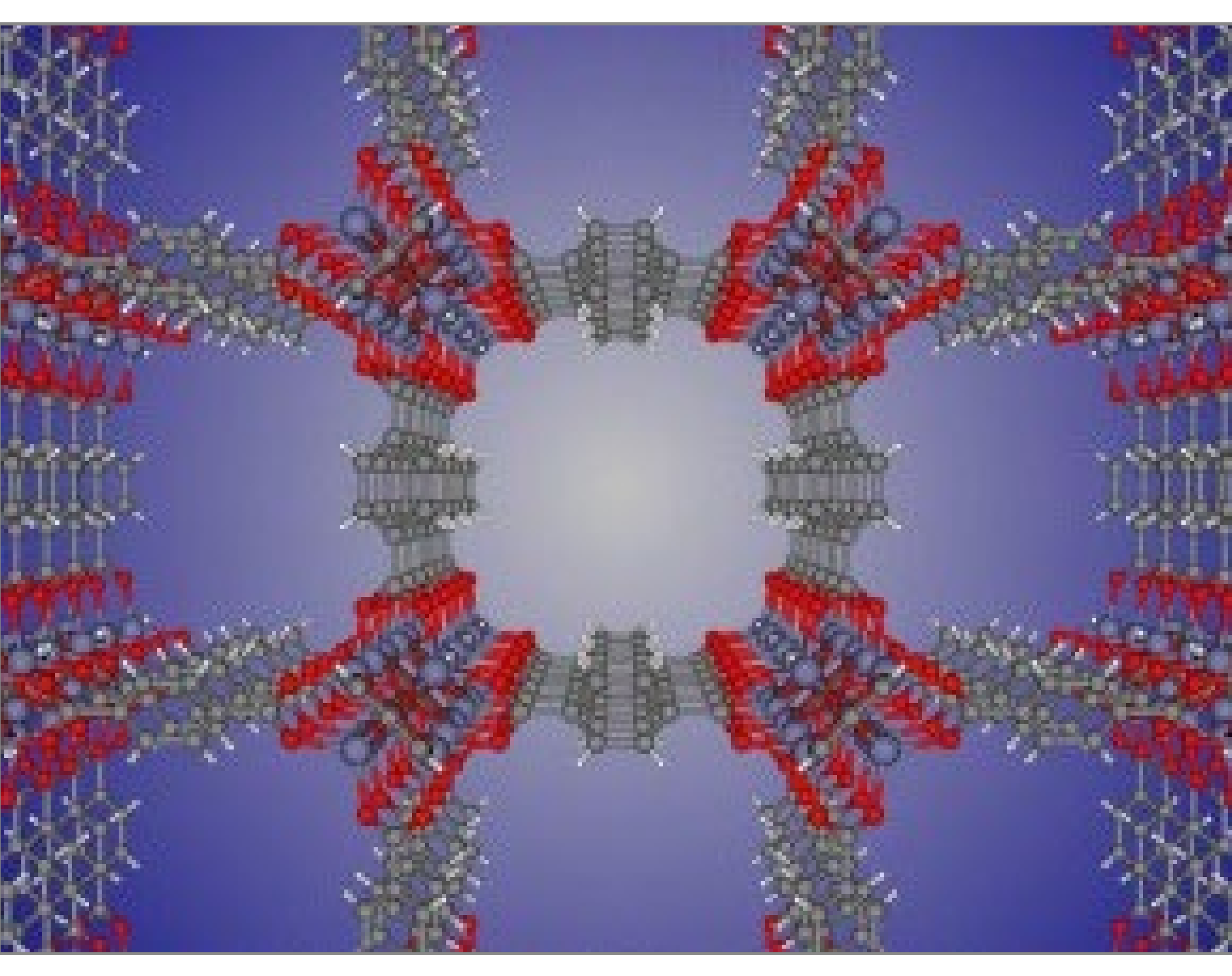
CANCER RESEARCH
Wellcome Sanger Institute



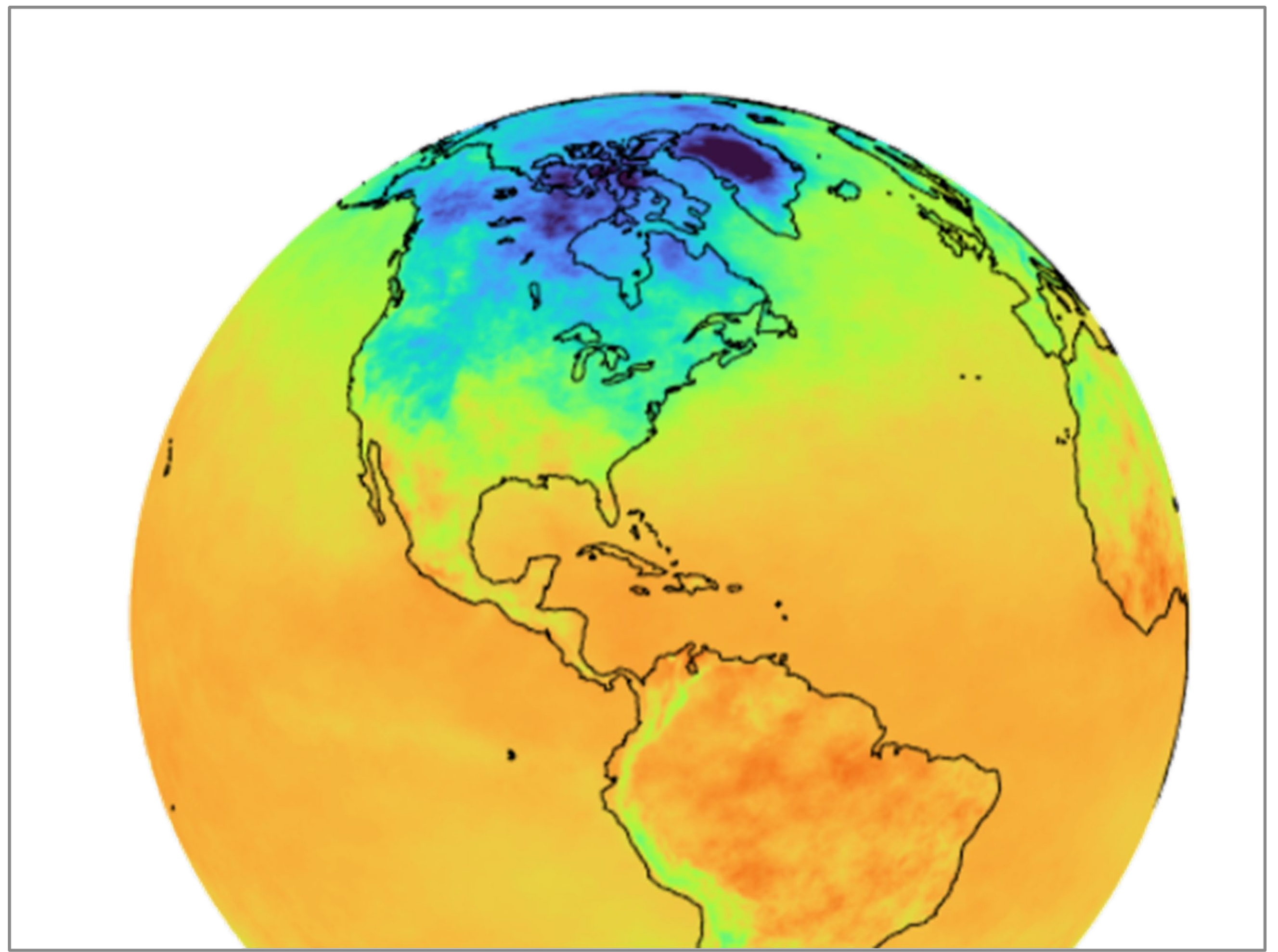
LABORATORY ROBOTICS
Argonne National Lab



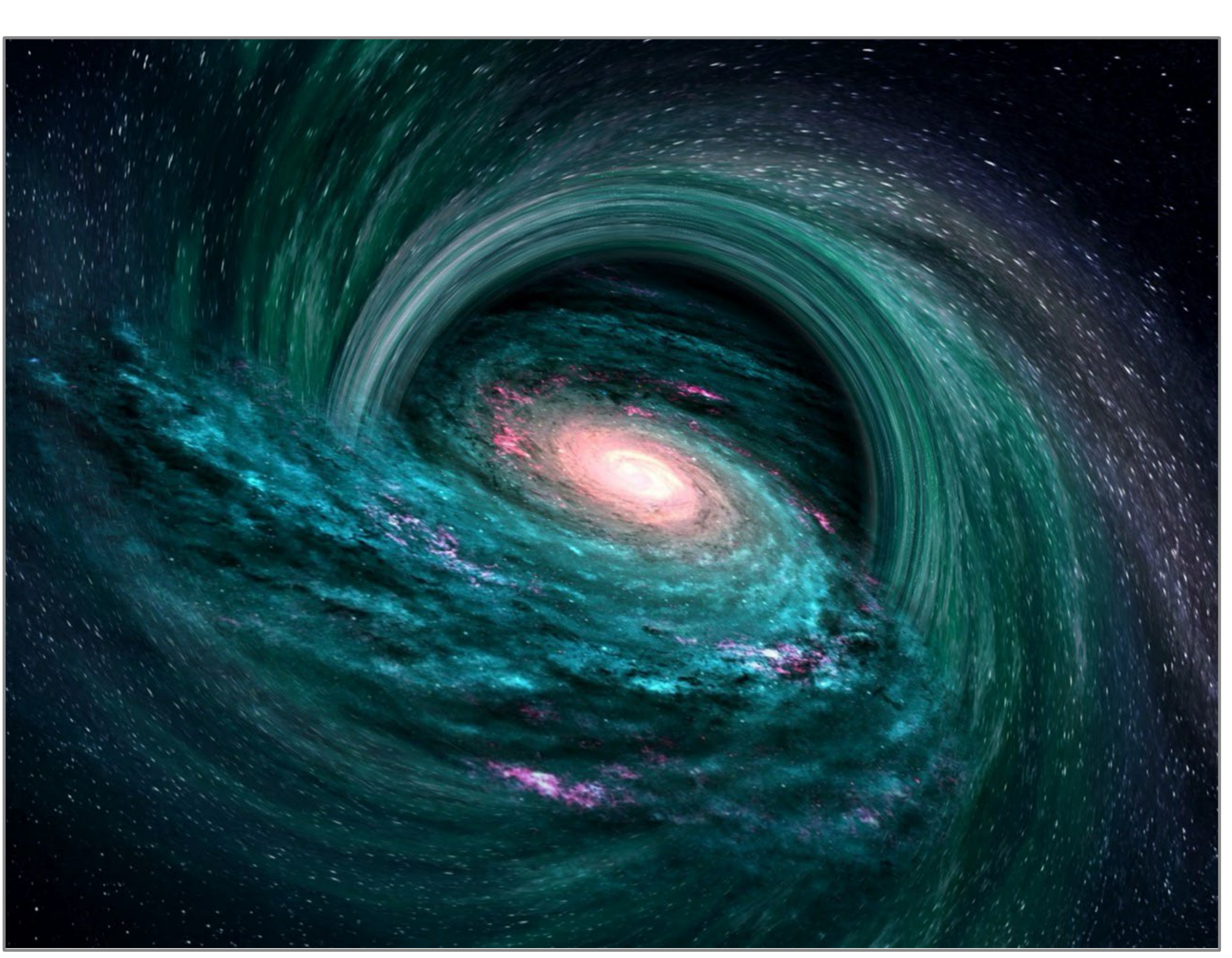
MATERIALS DISCOVERY
Microsoft Research



CLIMATE MODELING
KAUST

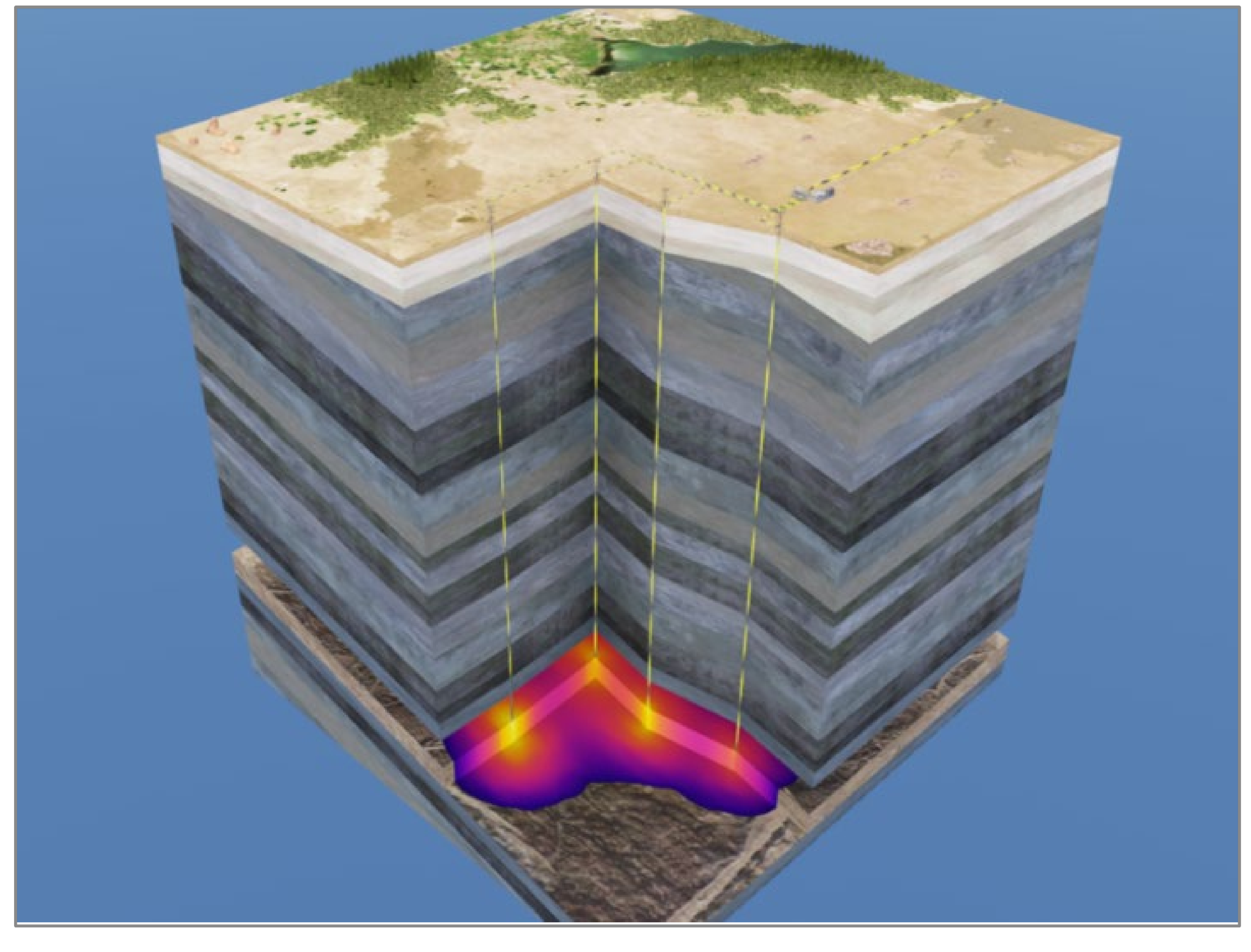


ASTRO FOUNDATION MODELS
The Flatiron Institute



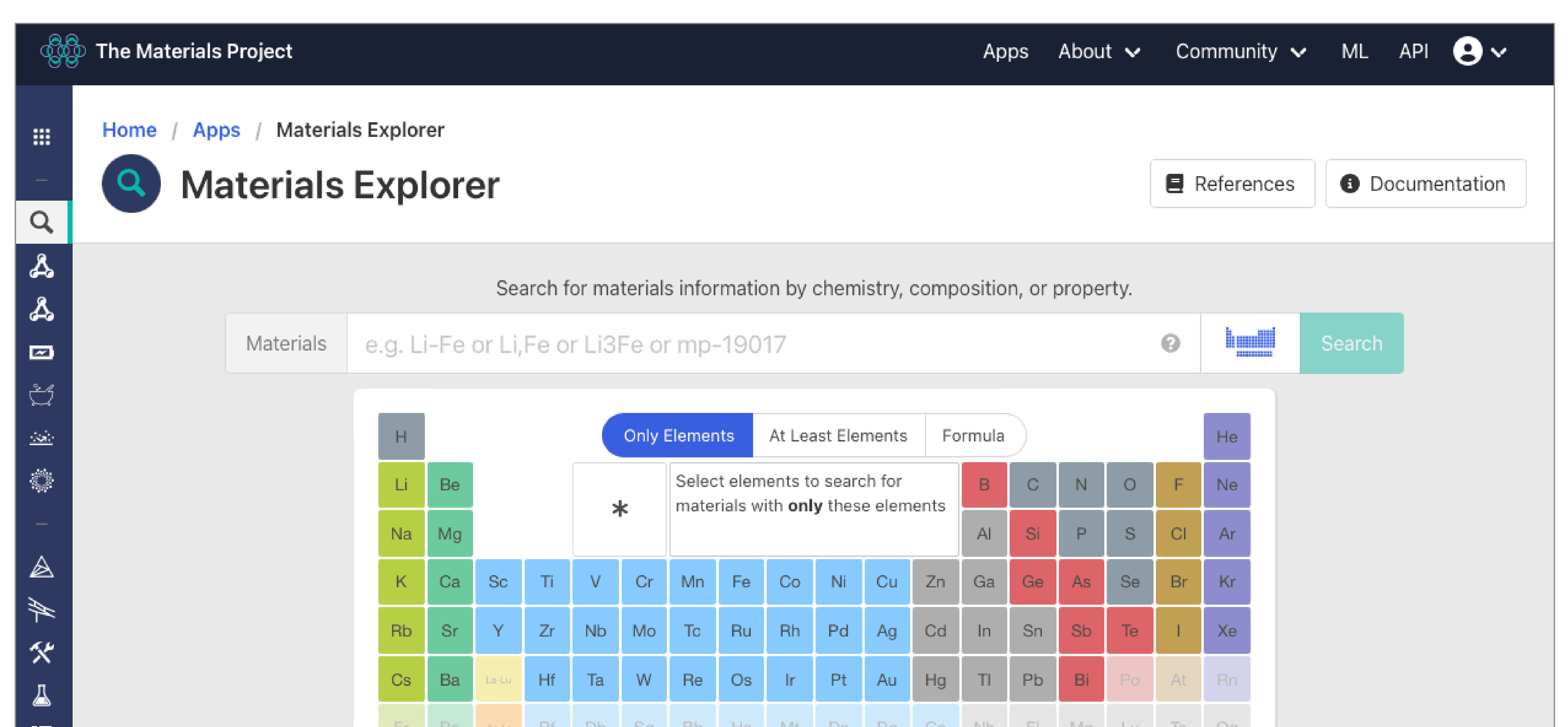
AGRICULTURAL HEALTH
Fermata

SEISMIC GEOSCIENCE
California Institute of Technology



AI is Transforming Material Science and Chemistry

Broad industry innovation and achievement



Materials Project
Multi-National Effort

Article
Scaling deep learning for materials discovery

https://doi.org/10.1038/s41586-023-06735-9
Received: 8 May 2023
Accepted: 10 October 2023
Published online: 29 November 2023
Open access
Check for updates

Amil Merchant[1,3], Simon Batzner[1,3], Samuel S. Schoenholz[1,3], Muratahan Aykol[1], Gowoon Cheon[2] & Ekin Dogus Cubuk[1,3]

Novel functional materials enable fundamental breakthroughs across technological applications from clean energy to information processing[1-11]. From microchips to batteries and photovoltaics, discovery of inorganic crystals has been bottlenecked by expensive trial-and-error approaches. Concurrently, deep-learning models for language, vision and biology have showcased emergent predictive capabilities with increasing data and computation[12-14]. Here we show that graph networks trained at scale can reach unprecedented levels of generalization, improving the efficiency of materials discovery by an order of magnitude. Building on 48,000 stable crystals identified in continuing studies[15-17], improved efficiency enables the discovery of 2.2 million structures below the current convex hull, many of which escaped previous

GNoME
Google DeepMind

MatterGen: a generative model for inorganic materials design

Claudio Zeni[1†], Robert Pinsler[1†], Daniel Zügner[1†], Andrew Fowler[1†], Matthew Horton[1†], Xiang Fu[1], Aliaksandra Shysheya[1], Jonathan Crabbé[1], Lixin Sun[1], Jake Smith[1], Bichlien Nguyen[1], Hannes Schulz[1], Sarah Lewis[1], Chin-Wei Huang[1], Ziheng Lu[1], Yichi Zhou[1], Han Yang[1], Hongxia Hao[1], Jielan Li[1], Ryota Tomioka[1*†], Tian Xie[1*†]

[1]Microsoft Research AI4Science.

*Corresponding author(s). E-mail(s): ryoto@microsoft.com; tianxie@microsoft.com;
†Equal contribution; non-corresponding authors are listed in random order.

rl-sci] 29 Jan 2024

MatterGen
Microsoft Research

MACE: Higher Order Equivariant Message Passing Neural Networks for Fast and Accurate Force Fields

Ilyes Batatia
Engineering Laboratory,
University of Cambridge
Cambridge, CB2 1PZ UK
Department of Chemistry,
ENS Paris-Saclay, Université Paris-Saclay
91190 Gif-sur-Yvette, France
ilyes.batatia@ens-paris-saclay.fr

Dávid Péter Kovács
Engineering Laboratory,
University of Cambridge
Cambridge, CB2 1PZ UK

Gregor N. C. Simm
Engineering Laboratory,
University of Cambridge
Cambridge, CB2 1PZ UK

Christoph Ortner
Department of Mathematics
University of British Columbia
Vancouver, BC, Canada V6T 1Z2

ML] 26 Jan 2023

MACE
OpenSource

MatterSim: A Deep Learning Atomistic Model Across Elements, Temperatures and Pressures

Han Yang[1*†], Chenxi Hu[1†], Yichi Zhou[1†], Xixian Liu[1†], Yu Shi[1†], Jielan Li[1*†], Guanzhi Li[1†], Zekun Chen[1†], Shuizhou Chen[1†], Claudio Zeni[1], Matthew Horton[1], Robert Pinsler[1], Andrew Fowler[1], Daniel Zügner[1], Tian Xie[1], Jake Smith[1], Lixin Sun[1], Qian Wang[1], Lingyu Kong[1], Chang Liu[1], Hongxia Hao[1*], Ziheng Lu[1*]

[1*]Microsoft Research AI for Science.

*Corresponding author(s). E-mail(s): hanyang@microsoft.com; jielanli@microsoft.com; hongxiahao@microsoft.com; zihenglu@microsoft.com;
†These authors contributed equally to this work.

mtrl-sci] 10 May 2024

Abstract

MatterSIM
Microsoft Research

Open Materials 2024 (OMat24) Inorganic Materials Dataset and Models

Luis Barroso-Luque, Muhammed Shuaibi, Xiang Fu, Brandon M. Wood, Misko Dzamba, Meng Gao, Ammar Rizvi, C. Lawrence Zitnick, Zachary W. Ulissi

Fundamental AI Research (FAIR) at Meta

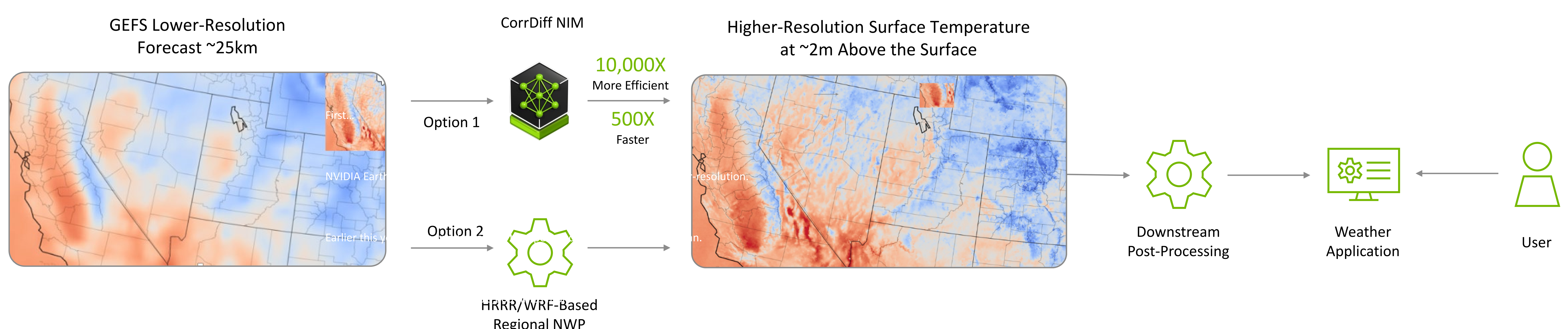
The ability to discover new materials with desirable properties is critical for numerous applications from helping mitigate climate change to advances in next generation computing hardware. AI has the potential to accelerate materials discovery and design by more effectively exploring the chemical space compared to other computational methods or by trial-and-error. While substantial progress has been made on AI for materials data, benchmarks, and models, a barrier that has emerged is the lack of publicly available training data and open pre-trained models. To address this, we present a Meta FAIR release of the Open Materials 2024 (OMat24) large-scale open dataset and an accompanying set of pre-trained models. OMat24 contains over 110 million density functional theory (DFT) calculations focused on structural and compositional diversity. Our EquiformerV2 models achieve state-of-the-art performance on the Matbench Discovery leaderboard and are capable of predicting ground-state stability and formation energies to an F1 score above 0.9 and an accuracy of 20 meV/atom, respectively. We explore the impact of model size, auxiliary denoising objectives, and fine-tuning on performance across a range of datasets including OMat24, MPtraj, and Alexandria. The open release of the OMat24 dataset and models enables the research community to build upon our efforts and drive

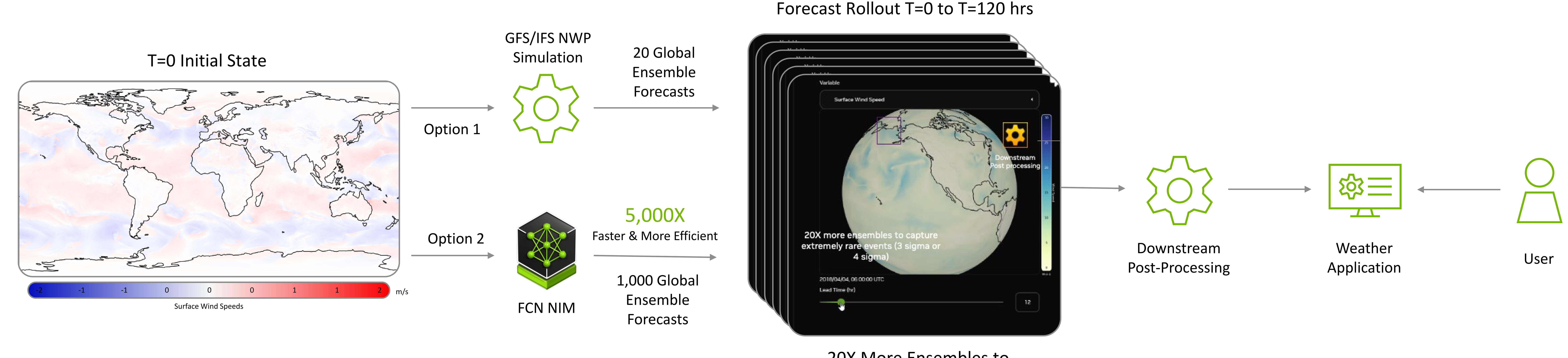
rl-sci] 16 Oct 2024

Open Materials 2024
Meta FAIR

Announcing Earth-2 NIMs for CorrDiff & FourCastNet

CorrDiff NIM for Generative AI Powered Downscaling | FourCastNet NIM for Global Weather Forecasting



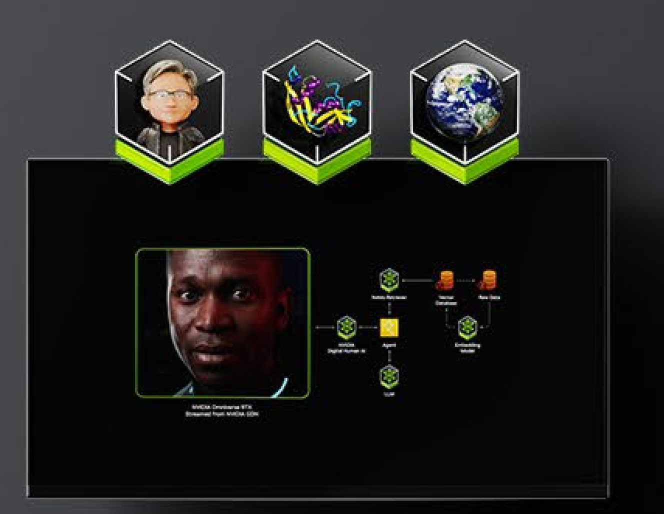


NVIDIA

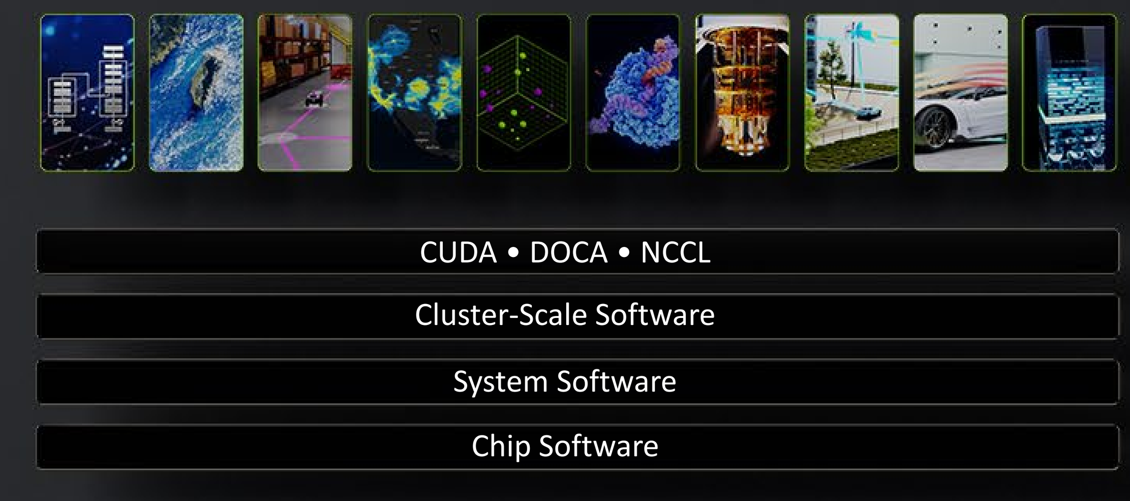
NVIDIA BUILDS AI SUPERCOMPUTING INFRASTRUCTURE

One Year Rhythm | Supercluster Scale | Full-Stack | CUDA Everywhere

NIM
CUDA-Accelerated
Agentic AI Libraries



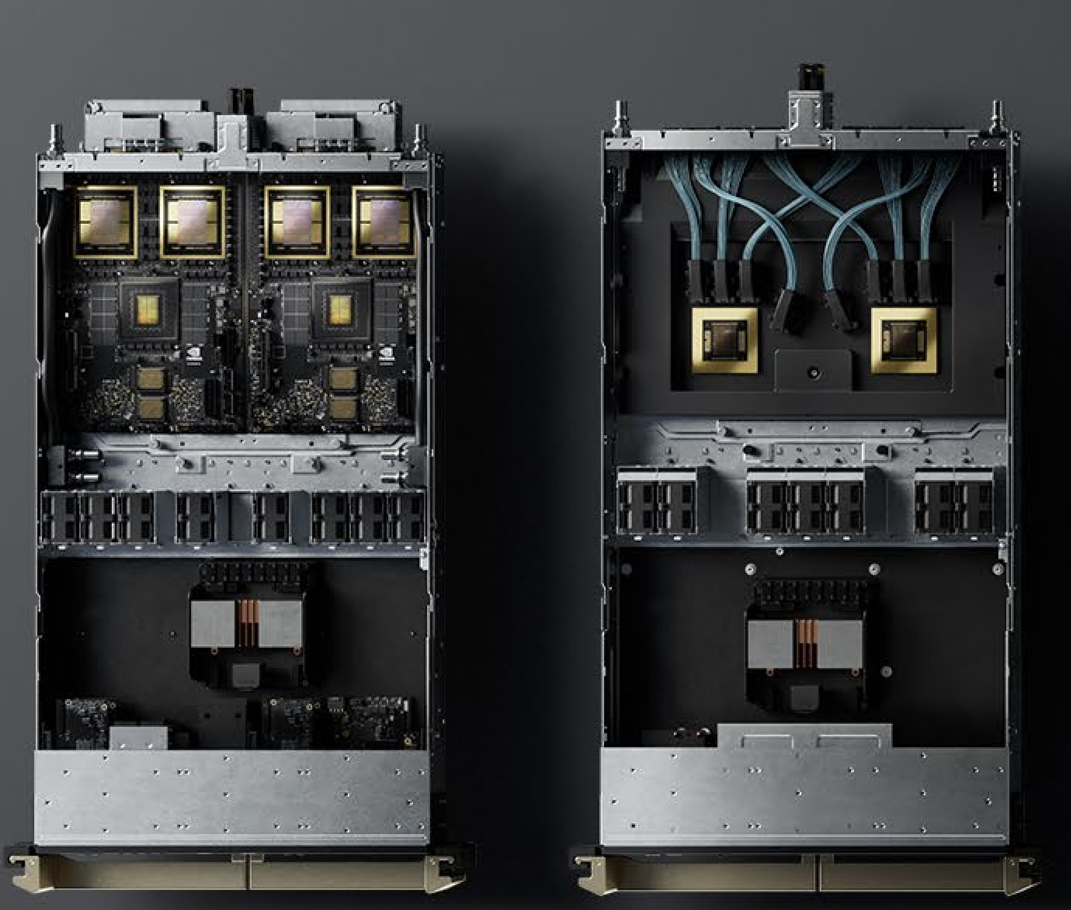
Omniverse
CUDA-Accelerated
Physical AI Libraries



CUDA-X Libraries

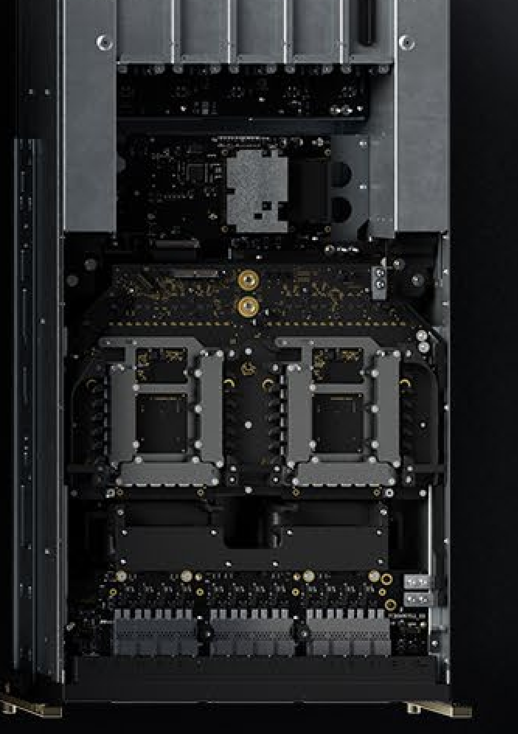
Accelerated
Software Stack

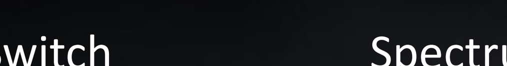
GB200 NVL72 SuperPOD



Grace Blackwell
MGX Node

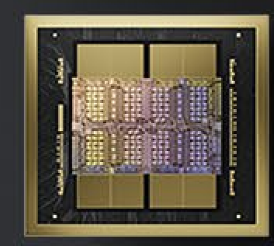
NVLink Switch

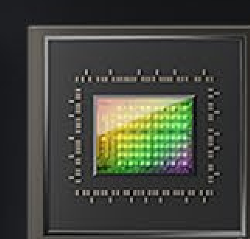


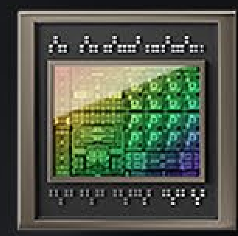


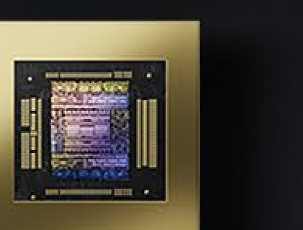
Quantum Switch

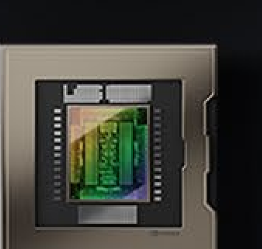
Spectrum-X Switch









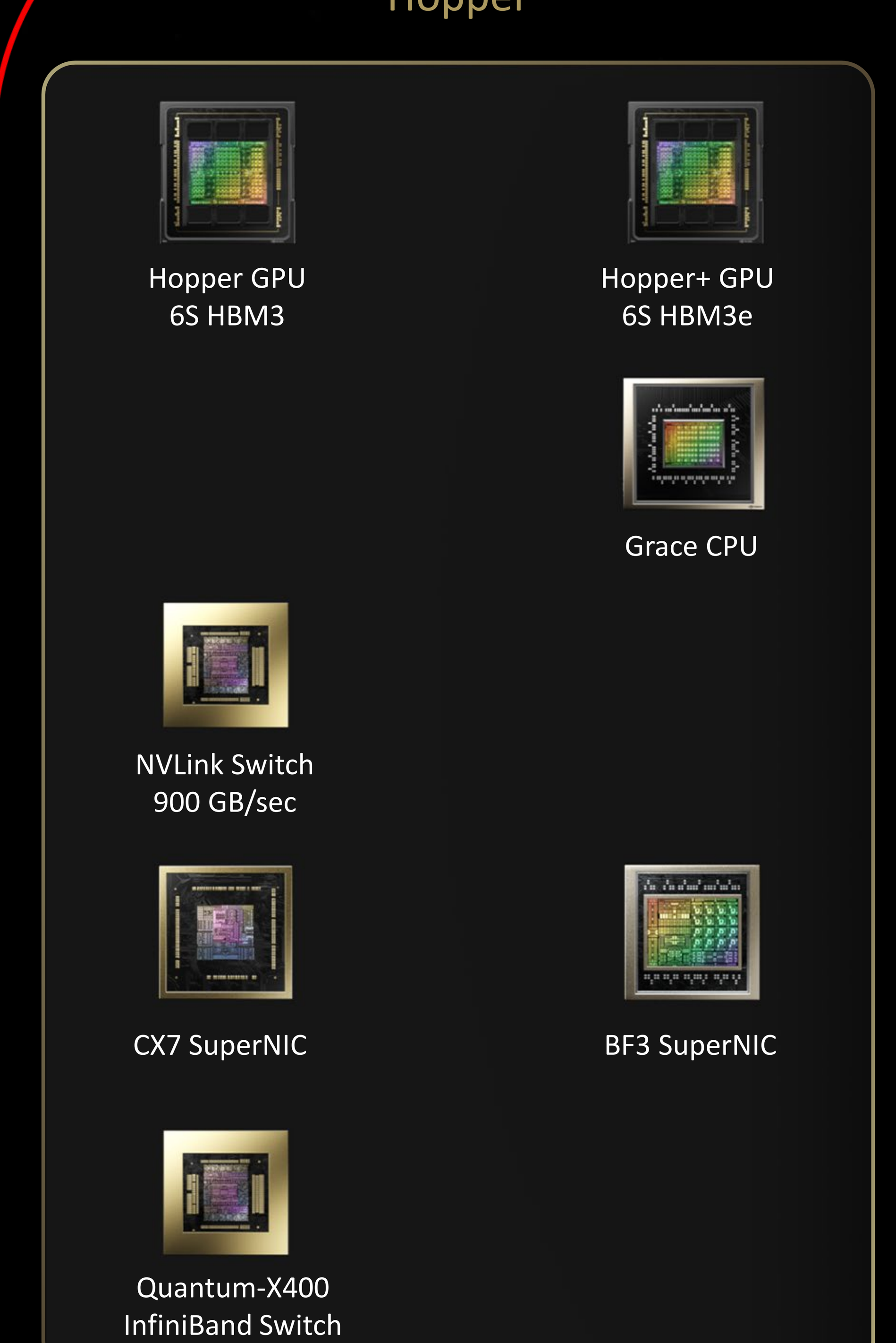


Chips Purpose-Built for AI Supercomputing

GPU | CPU | DPU | NIC | NVLink Switch | IB Switch | ENET Switch

ONE YEAR RHYTHM | SUPERCLUSTER SCALE | FULL-STACK | CUDA EVERYWHERE

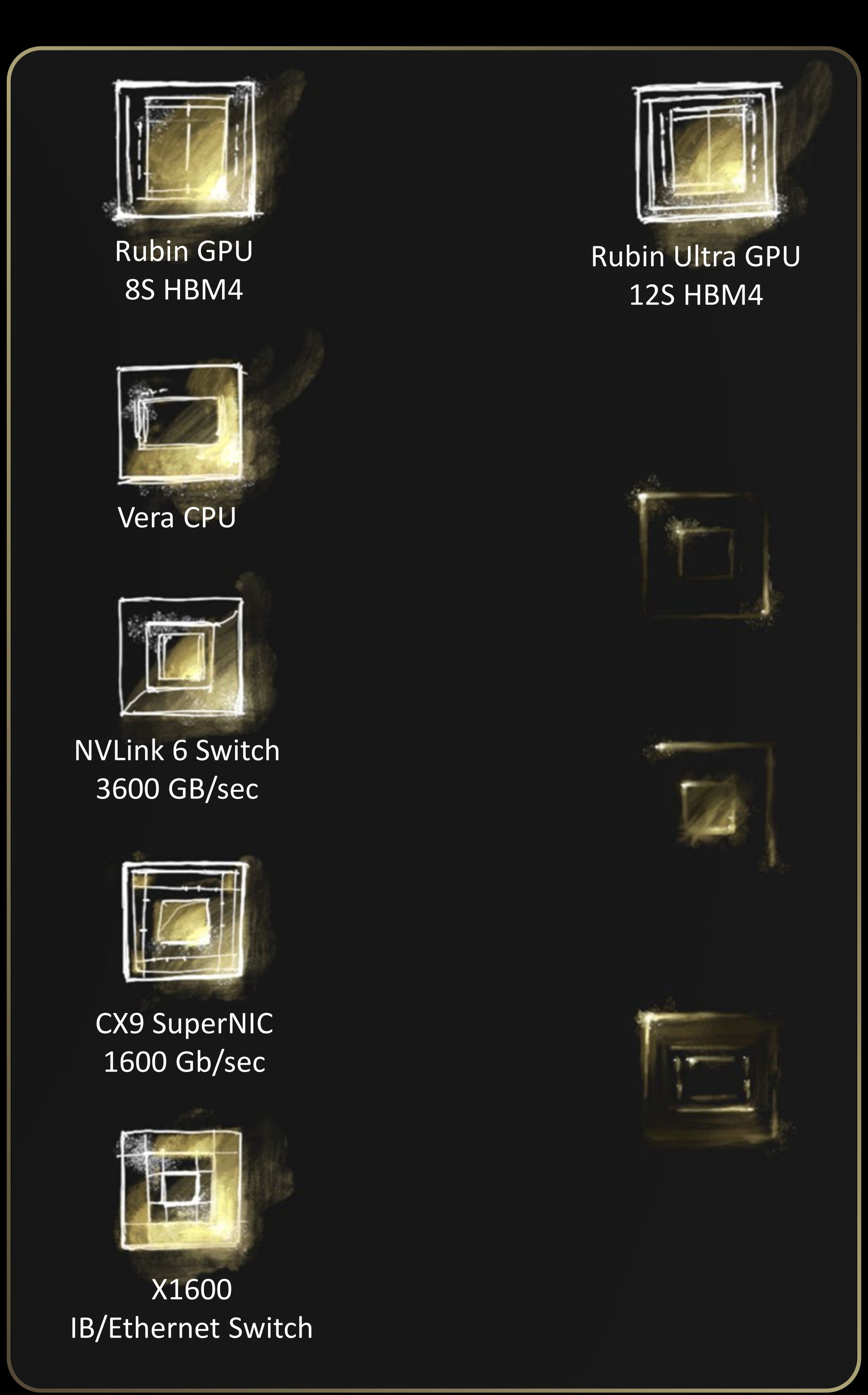
Hopper



Blackwell

Blackwell-Ultra

Rubin



2022 2023 2024 2025 2026 2027

Ecosystem Powering the Next Wave of AI Supercomputing Systems

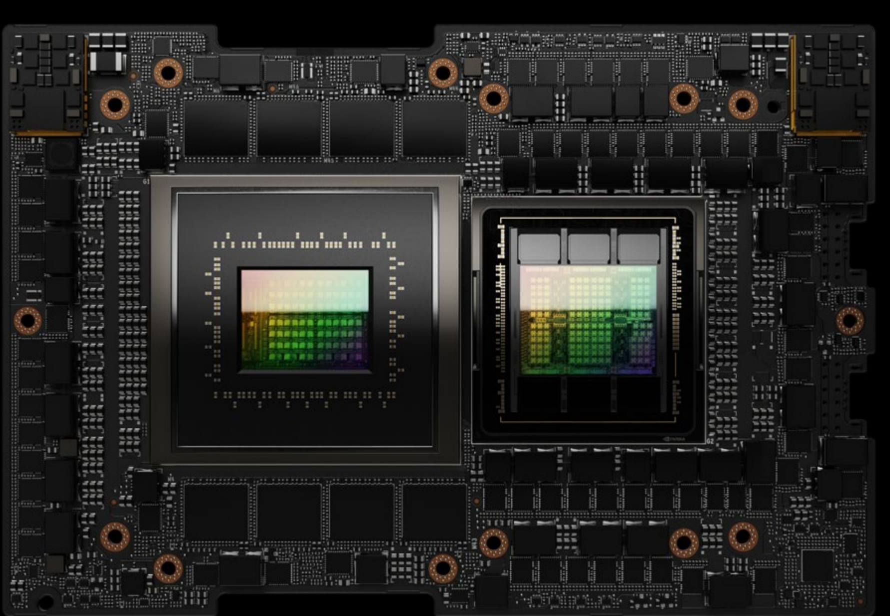
Partners Supercharge HPC and AI

DELL Technologies

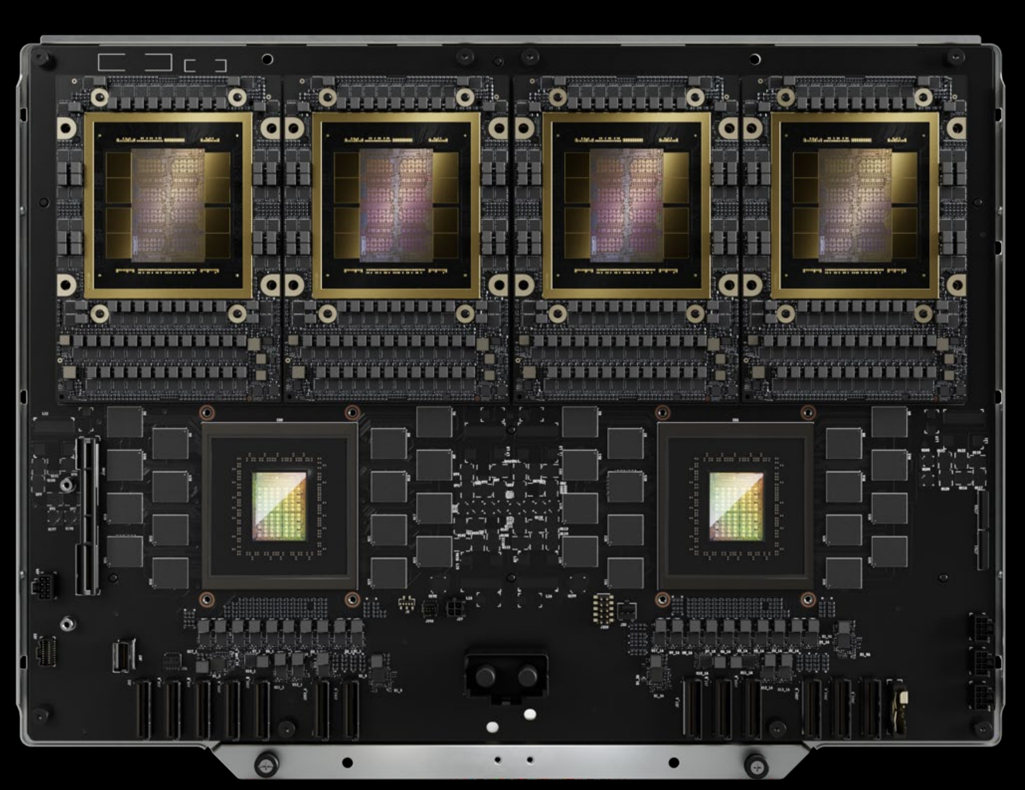
GIGABYTE™

Inventec

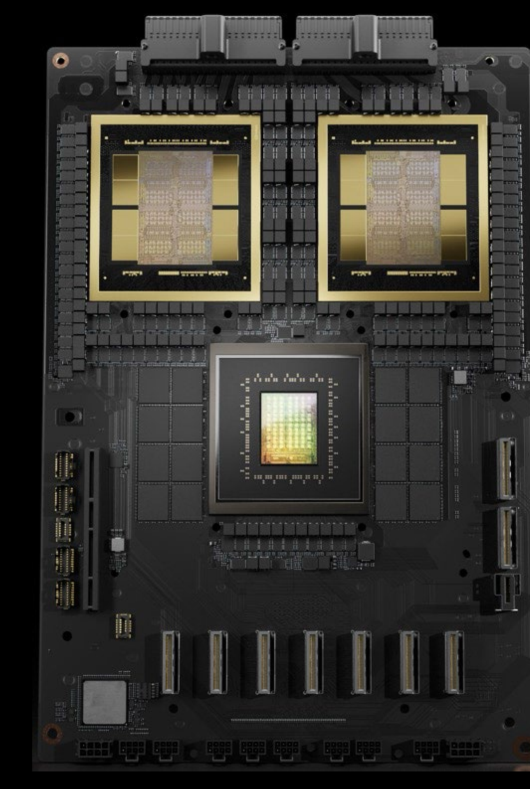
PEGATRON



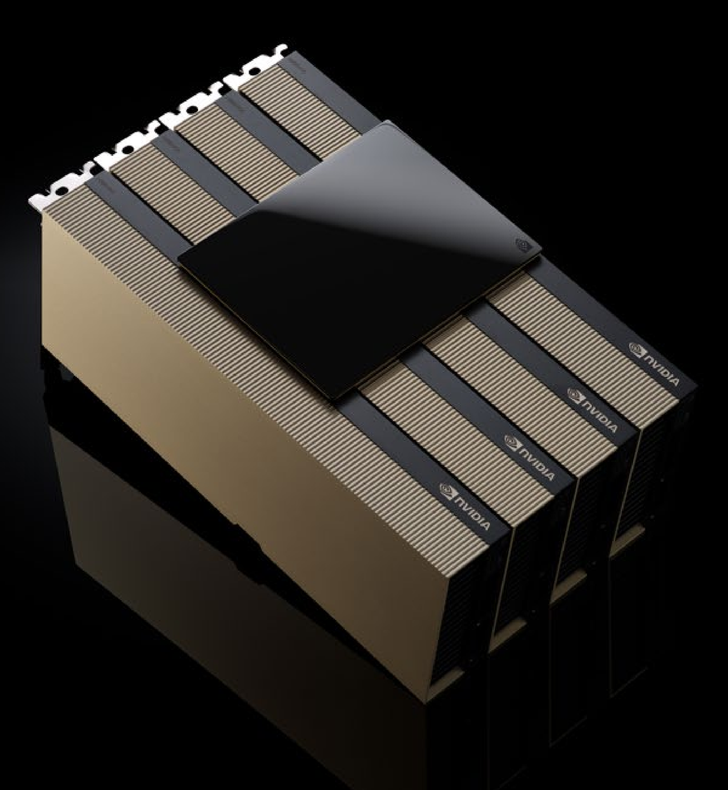
Grace Hopper Superchip



NVIDIA GB200 Grace Blackwell NVL4 Superchip



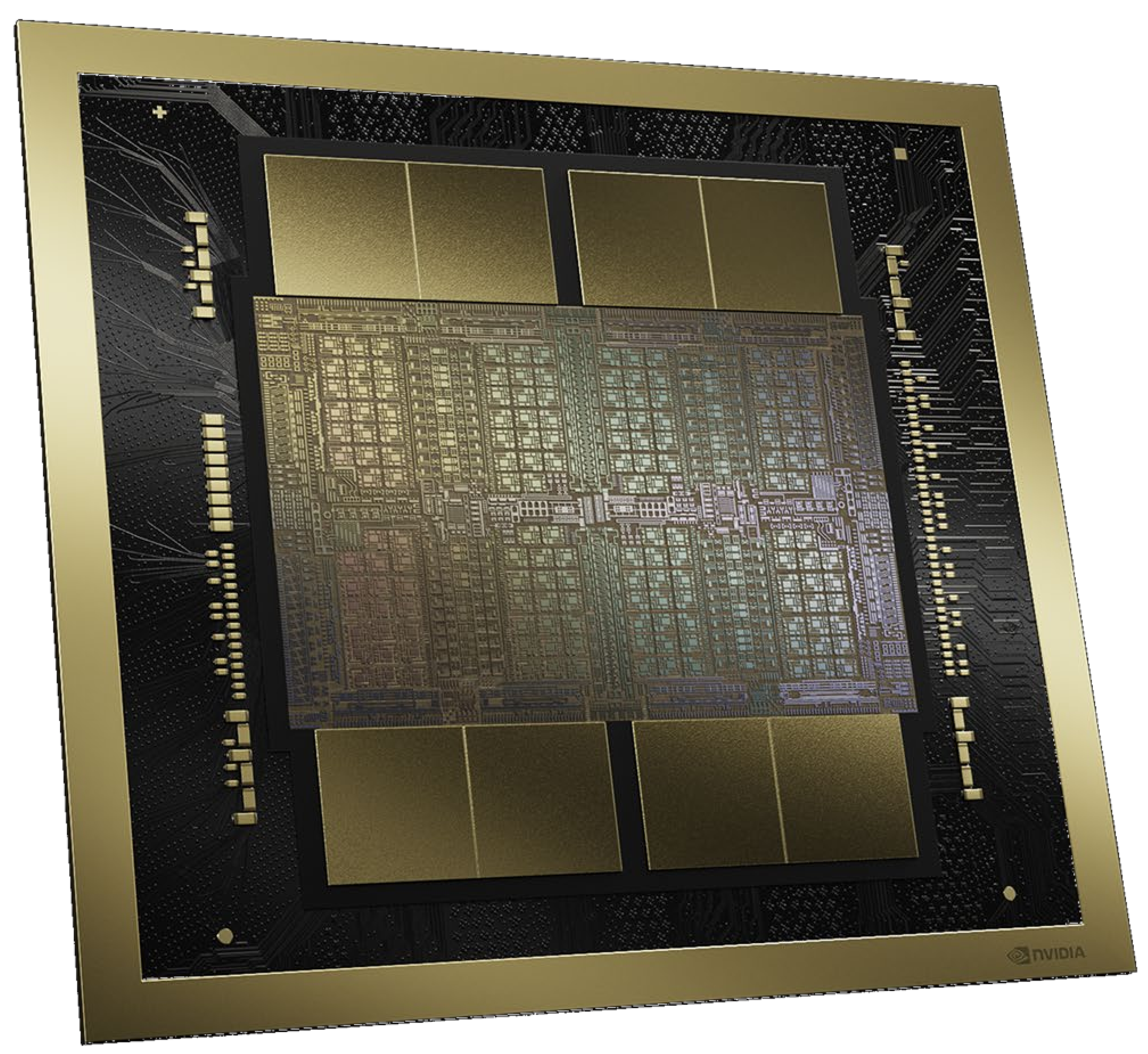
NVIDIA GB200 Grace Blackwell Superchip



H200 NVL

NVIDIA Blackwell

The Engine of the New Industrial Revolution

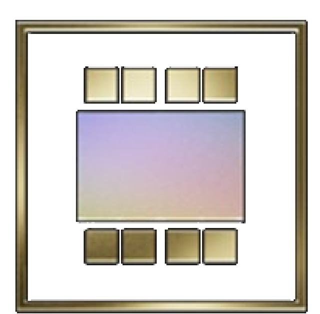


Built to Democratize Trillion-Parameter AI

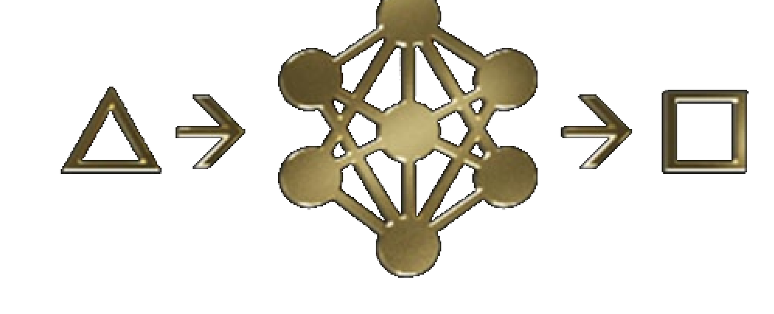
20 PetaFLOPS of AI performance on a single GPU

4X Training | 30X Inference | 25X Energy Efficiency & TCO

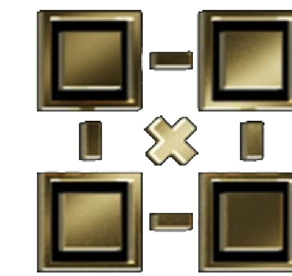
Expanding AI Datacenter Scale to beyond100K GPUs



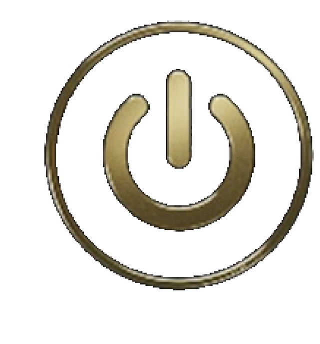
AI SUPERCHIP
208B Transistors



2nd GEN TRANSFORMER ENGINE
FP4/FP6 Tensor Core



5th GENERATION NVLINK
Scales to 576 GPUs



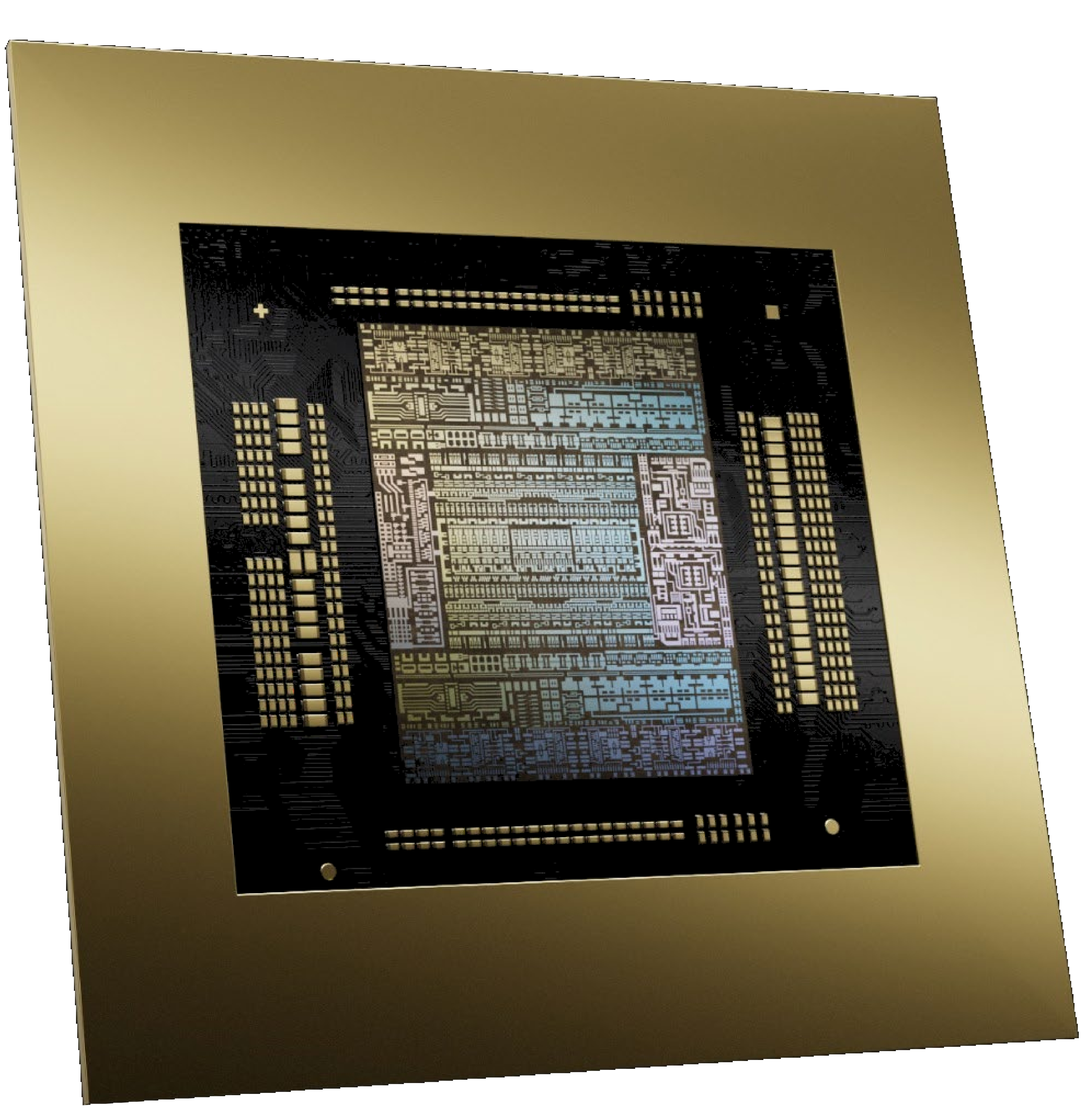
RAS ENGINE
100% In-System
Self-Test

SECURE AI
Full Performance
Encryption & TEE

DECOMPRESSION ENGINE
800 GB/s

Announcing Fifth Generation NVLink and NVLink Switch Chip

Efficient Scaling for Trillion Parameter Models



7.2 TB/s Full all-to-all Bidirectional Bandwidth

Sharp v4 plus FP8

3.6 TF In-Network Compute

Expanding NVLink up to 576 GPU NVLink Domain

18X Faster than Today's Multi-Node Interconnect

GB200 NVL72

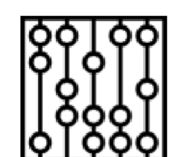
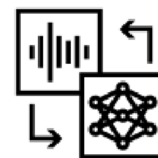
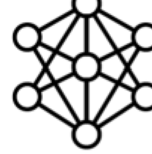
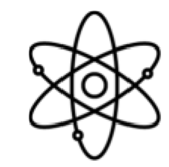
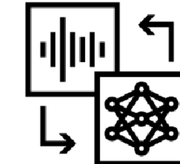
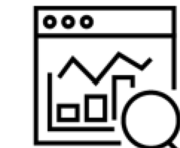
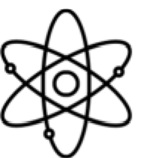
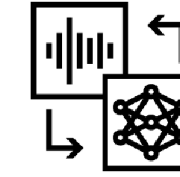
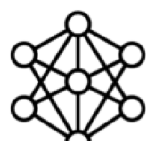
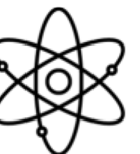
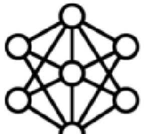
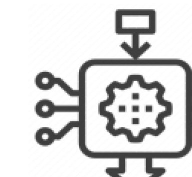
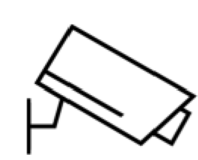
Delivers New Unit of Compute

GB200 NVL72

- 36 GRACE CPUs
- 72 BLACKWELL GPUs
- Fully Connected NVLink Switch Rack

Training	720 PFLOPs
Inference	1,440 PFLOPs
NVL Model Size	**27T params**
Multi-Node All-to-All	130 TB/s
Multi-Node All-Reduce	260 TB/s

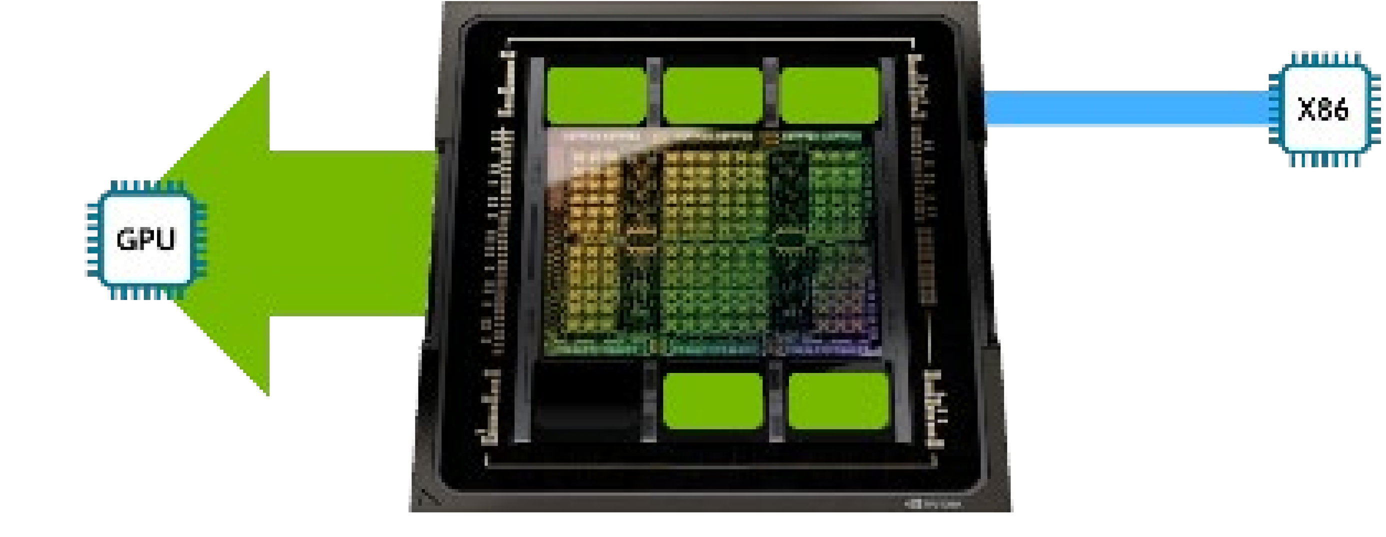
Choose The Right *Solution* For The Job

Scale-up, CPU+GPU & HGX Products		Scale-out, CPU+GPU & PCIe Products			
Real-Time Trillion-Parameter Models LLM & MoE	**Highest Compute Performance HPC & AI**	**Most Versatile Platform Diverse Workloads**	**Most Efficient Compute LLM Inference, AI + HPC**	**Fast Universal AI + Graphics Text to Image/Video**	**AI Video Entry-tier Universal & Edge AI**
 Graph Neural Networks  Massive Scale Model Training & Inference	  AI Training, Inference & Scientific Research Data Analytics	 LLM Inference & Retrieval-Augmented Generation   Data Analytics, Vector Database & HPC	 LLM Inference & Retrieval-Augmented Generation   AI, HPC & Data Analytics	Text to Image/Video AI Multi-modal Generative AI  Fine Tune Training/Inference, GenAI + Omniverse	  Edge AI, Inference + Video and AI Mobile Cloud Gaming + vWS
405B - 1T+	70B-405B	70B-175B	70B-175B	13B-70B	Up to 7B
Rack Power: ~120 / 70kW	Node Power: ~11 / 5.5kW	Node Power: ~3kW	Node Power: ~8kW	Node Power: ~3kW	Node Power: ≤2kW
NVLink Domain 72	NVLink Domain 8 & 4	NVLink Domain 2	NVLink Domain 4 & 2	NA	NA
Max GPUs per NVLink Domain Max Performance & Capability	4-8 GPUs per Baseboard Highest Compute Performance AI, HPC & Data Analytics	1 – 2 GPUs per node Best Inference TCO Adv. Arch for AI & HPC	4-8 GPUs per node 2nd Generation MIG 5-year NVIDIA AI Enterprise	Fastest RT Graphics Largest Render Models	1-16 GPUs per node Video & Graphics Compact & Versatile
GB200 NVL72 1000W, 480GB, 144GB	**HGX H200 | H100** 700W, 141GB | 80GB	**GH200 NVL2** 1000W, 480GB, 144GB	**H200 NVL | H100 NVL** 600W, 141GB | 400W, 94GB NVL2 & 4 | NVL2	**L40S** 350W | 48GB 2-Slot FHFL	**L4** 72W | 24GB 1-Slot HHHL

Network	Scale-up (GB200 NVL72)	HGX / GH200 / H200 NVL / L40S	L4
EW InfiniBand	**Quantum-2/CX7**	**Quantum-2/ConnectX-7**	Ethernet/Wi-Fi
EW Ethernet	**N/A**	**Spectrum-X / B3140H SuperNIC**	
NS MGMT, IO	**BlueField-3 DPU**	**BlueField-3 DPU**	

X86 + HOPPER

Architectures & *Connectivity*



PCIe

NEW

H100 up to NVL2	94GB HBM3
H200 up to NVL4	141GB HBM3e

HGX 4-way

NEW

HGX H100 4-GPU	80GB HBM3 3.4TB/s
HGX H200 4-GPU	141GB HBM3e 4.8TB/s

HGX 8-way

OEM and DGX options

NEW

HGX H100 8-GPU	80GB HBM3 3.4TB/s
HGX H200 8-GPU	141GB HBM3e 4.8TB/s

NVIDIA GH200 Grace Hopper Superchip

Built for the New Era of Accelerated Computing and Generative AI

Most versatile compute

Best performance across CPU, GPU or memory intensive applications

Easy to deploy and scale out

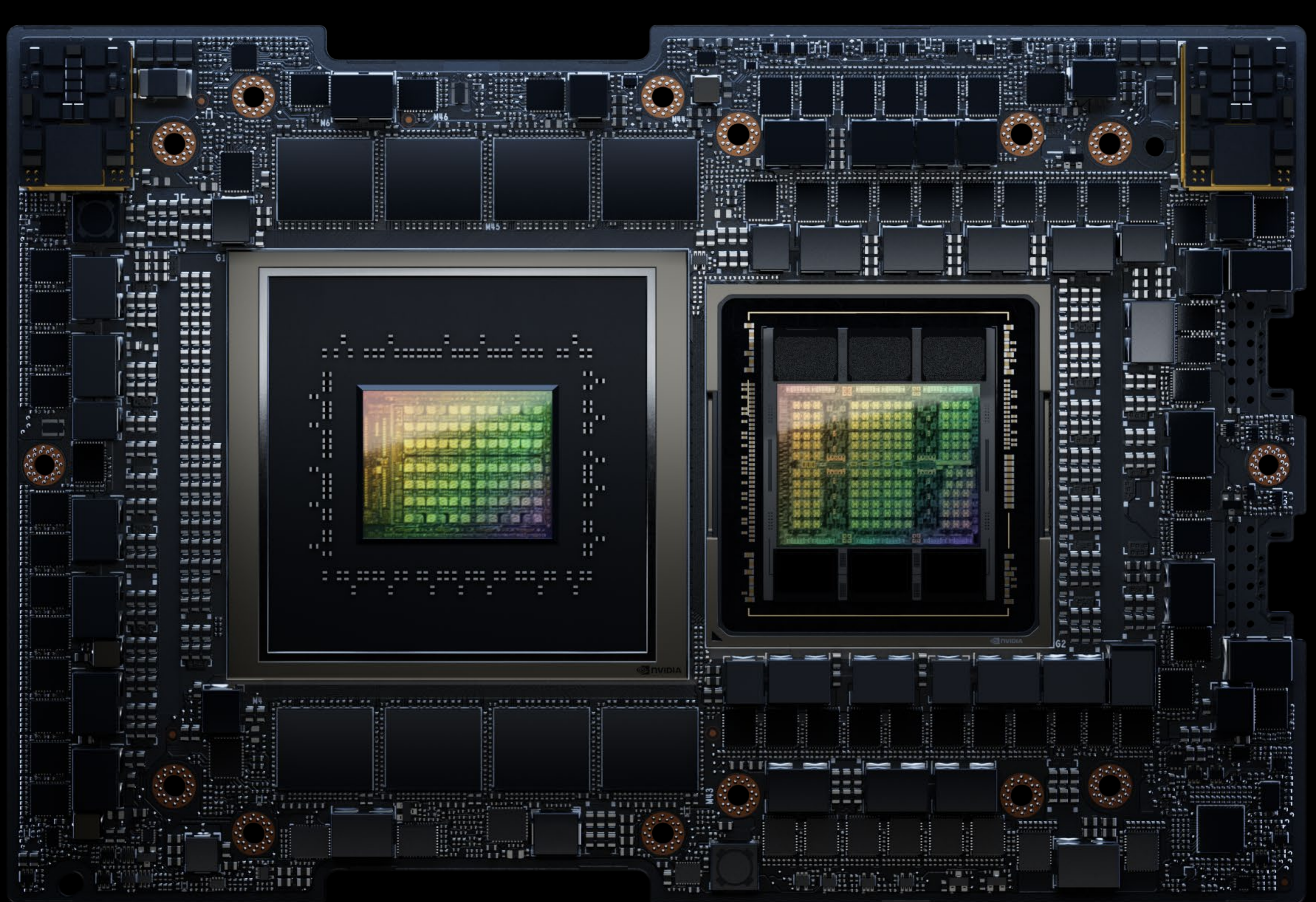
1 CPU:1 GPU node simple to manage and schedule for for HPC, enterprise, and cloud

Best Perf/TCO for diverse workloads

Maximize data center utilization and power efficiency

Continued Innovation

Grace and Blackwell in 2025



900GB/s NVLink-C2C | 624GB High-Speed Memory
4 PF AI Perf | 72 Arm Cores

Grace Hopper Powers AI Supercomputing Datacenters

Grace Hopper Will Deliver 200 Exaflops of AI performance for Groundbreaking Research

Cumulative AI FLOPS

Cumulative AI Performance (ExaFLOPS of AI)

400
350
300
250
200
150
100
50
0

2015 2016 2017 2018 2019 2020 2021 2022 2023 2024 2025

Piz Daint
Tsubame3
Summit
Pangea
Selene
Perlmutter
Polaris
Leonardo
MareNostrum-5
Azure ND H100 v5
Eos
Alps
Vista
Venado
EXA1 HE
JEDI
DeltaAI
Isambard AI
Miyabi
OFP-II
Helios
BioHive2
GENCI

Isambard AI
VENADO
ALPS

200 ExaFLOPS
AI Grace Hopper in Supercomputing Centers

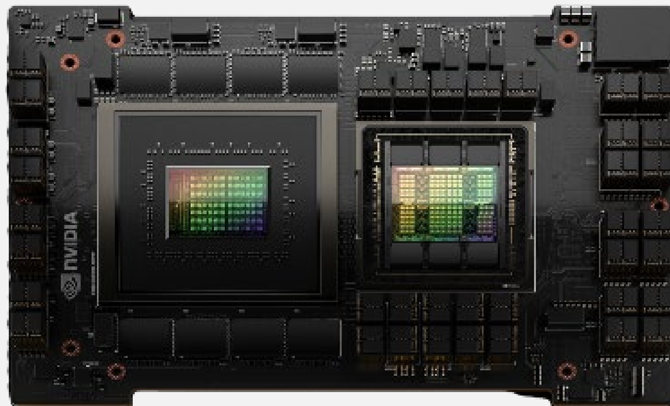
65%
of Hopper are Grace Hopper

2X
More energy efficient

GRACE GPU-GPU NVLINK

Architectures & Cost of *Connectivity*

1-Way
1:1



GH200 96GB

480 GB LP5x
96GB HBM3

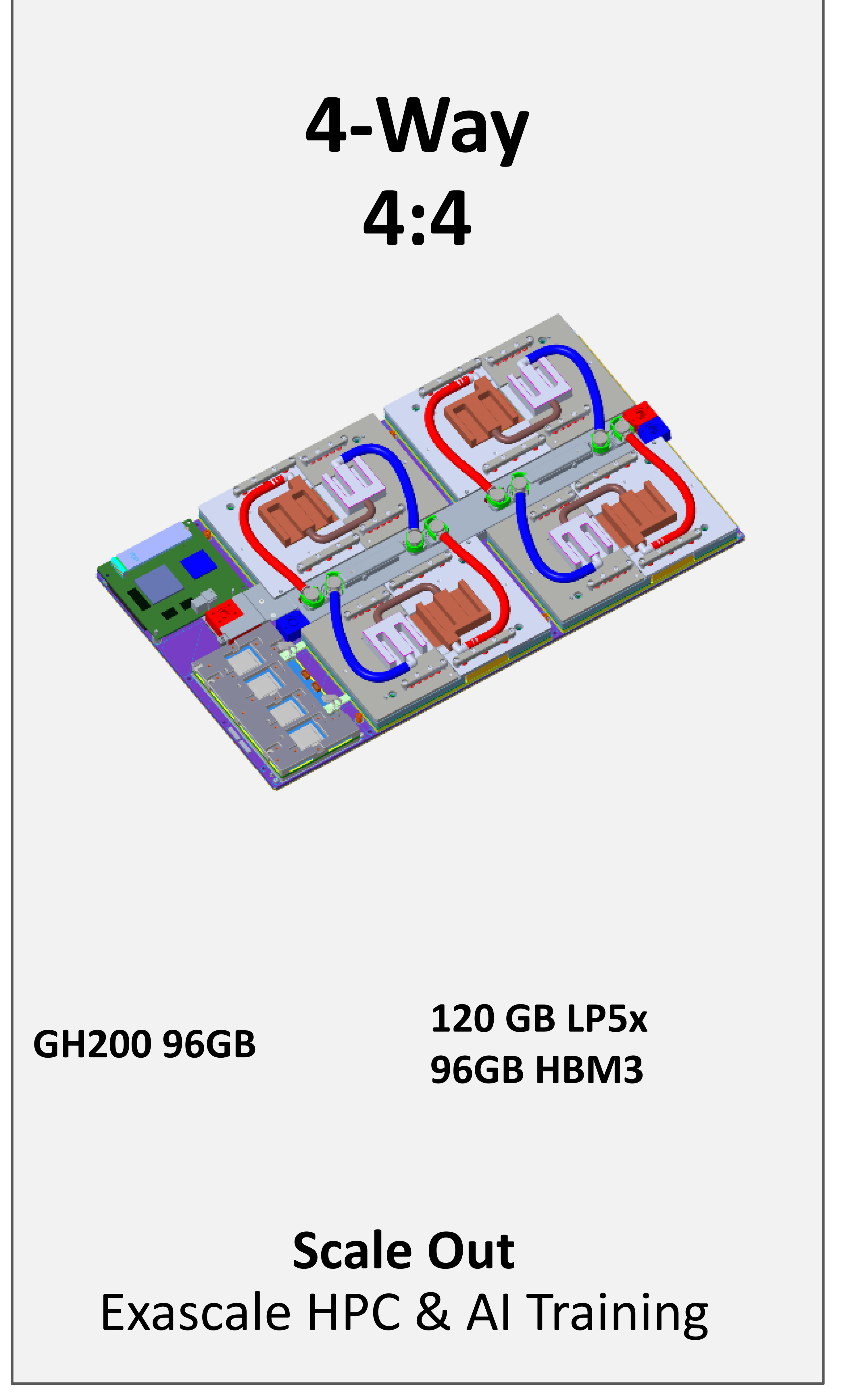
Scale Out
AI Inference

2-Way
2:2

GH200 NVL2

480 GB LP5x
144GB HBM3e

Scale Out
AI Inference



4-Way
4:4

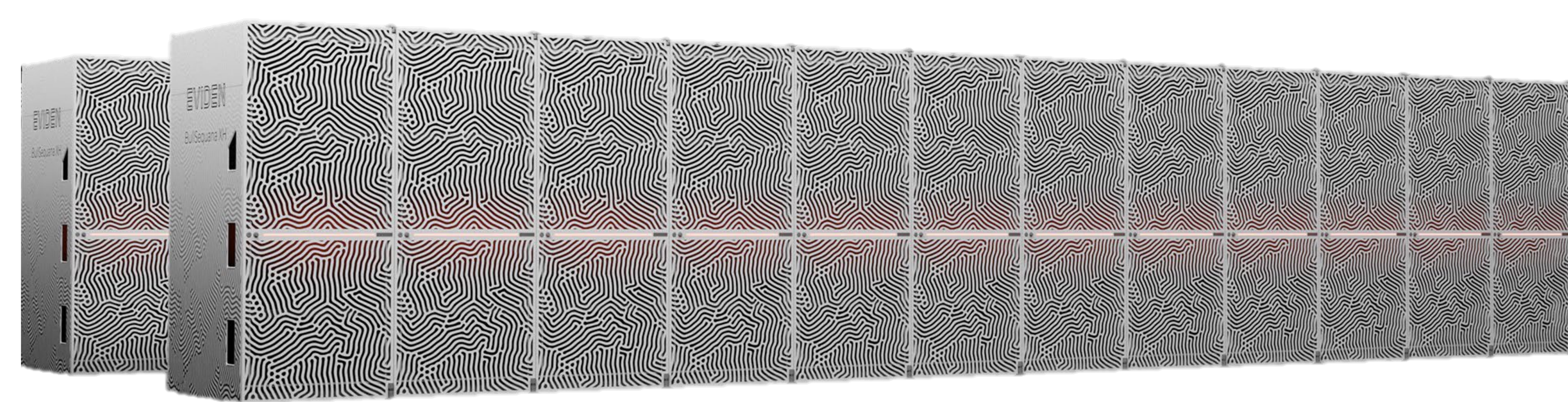
GH200 96GB

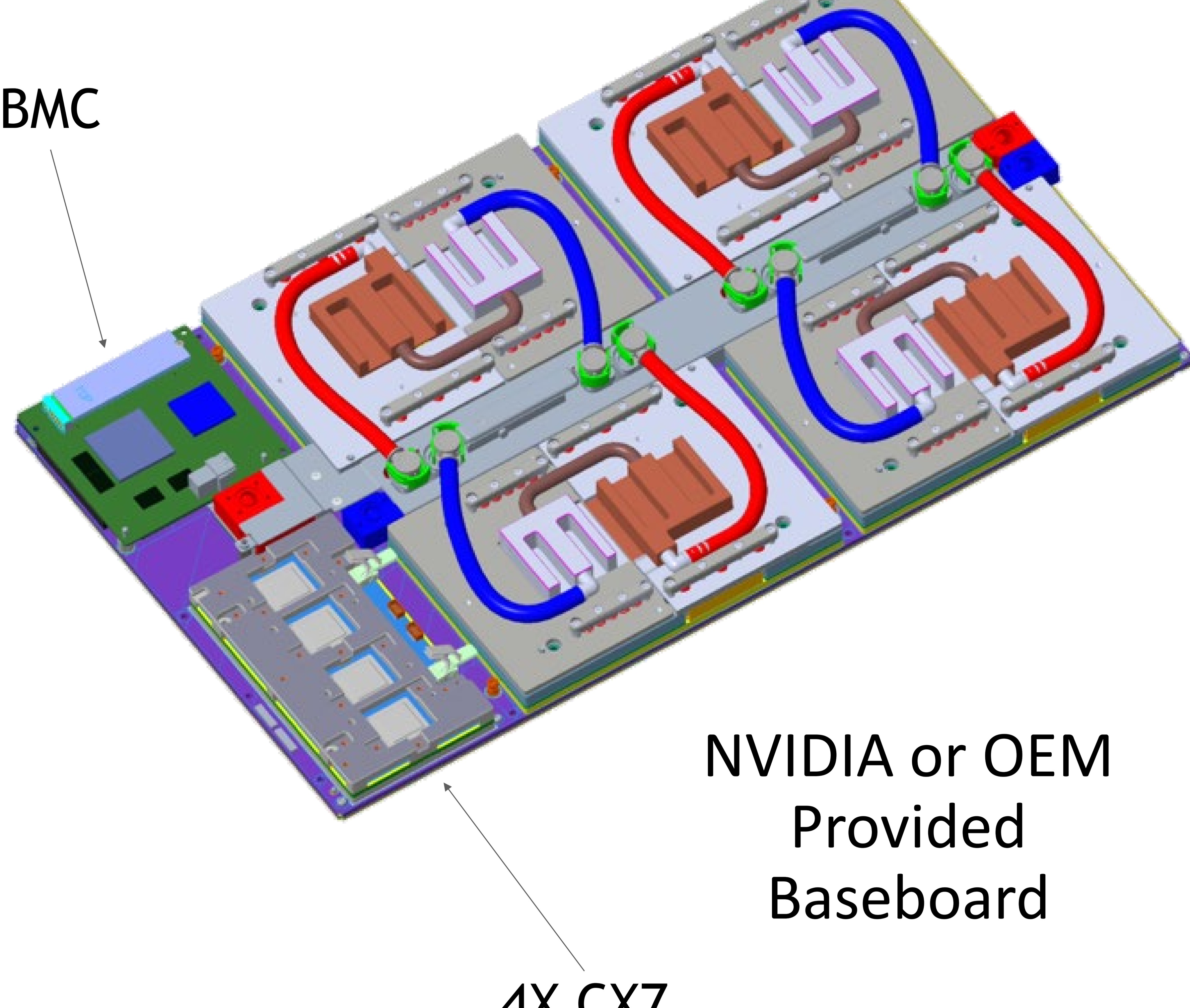
120 GB LP5x
96GB HBM3

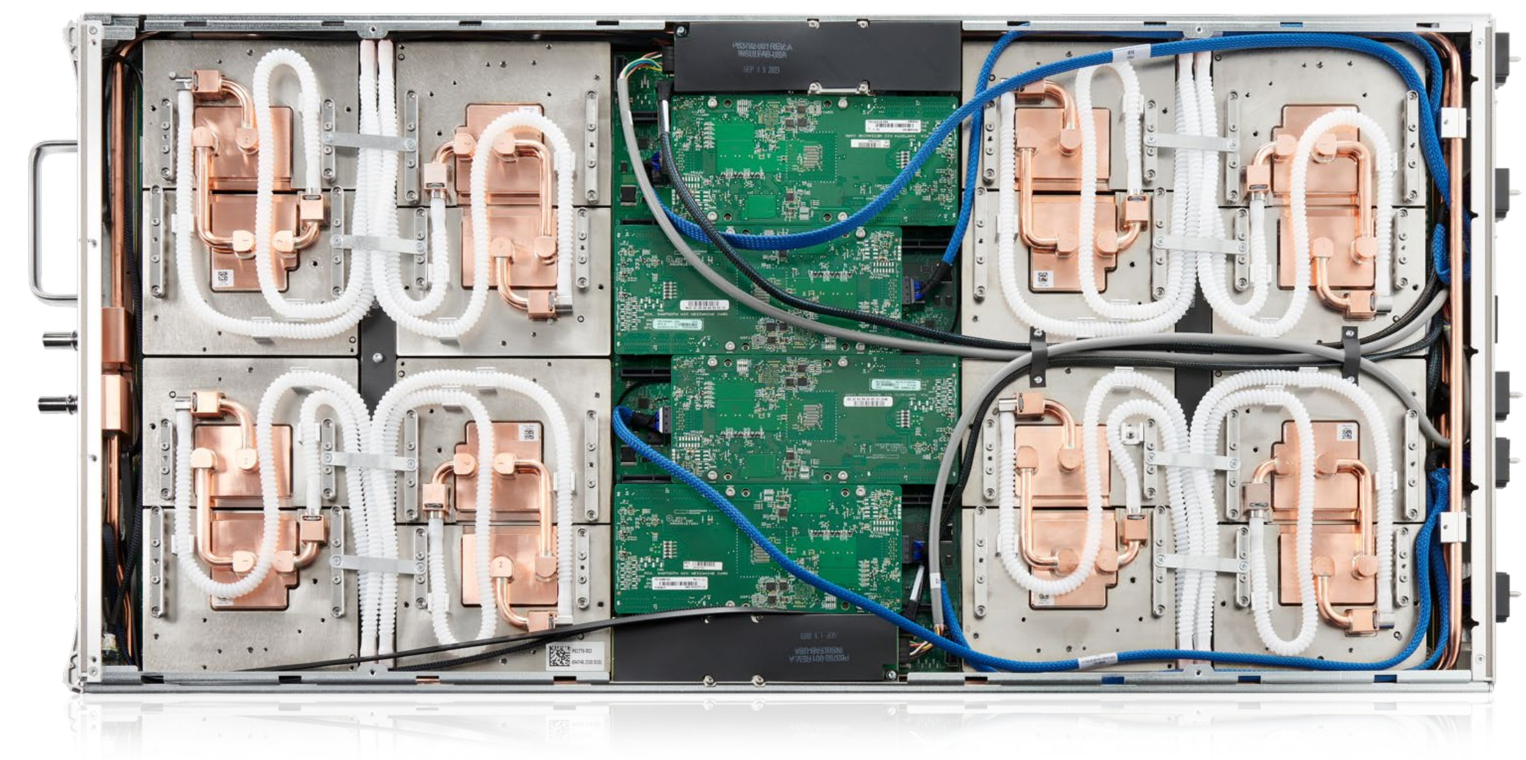
Scale Out
Exascale HPC & AI Training

Grace Hopper Superchip 4-Way Design

The choice for the world's fastest supercomputers





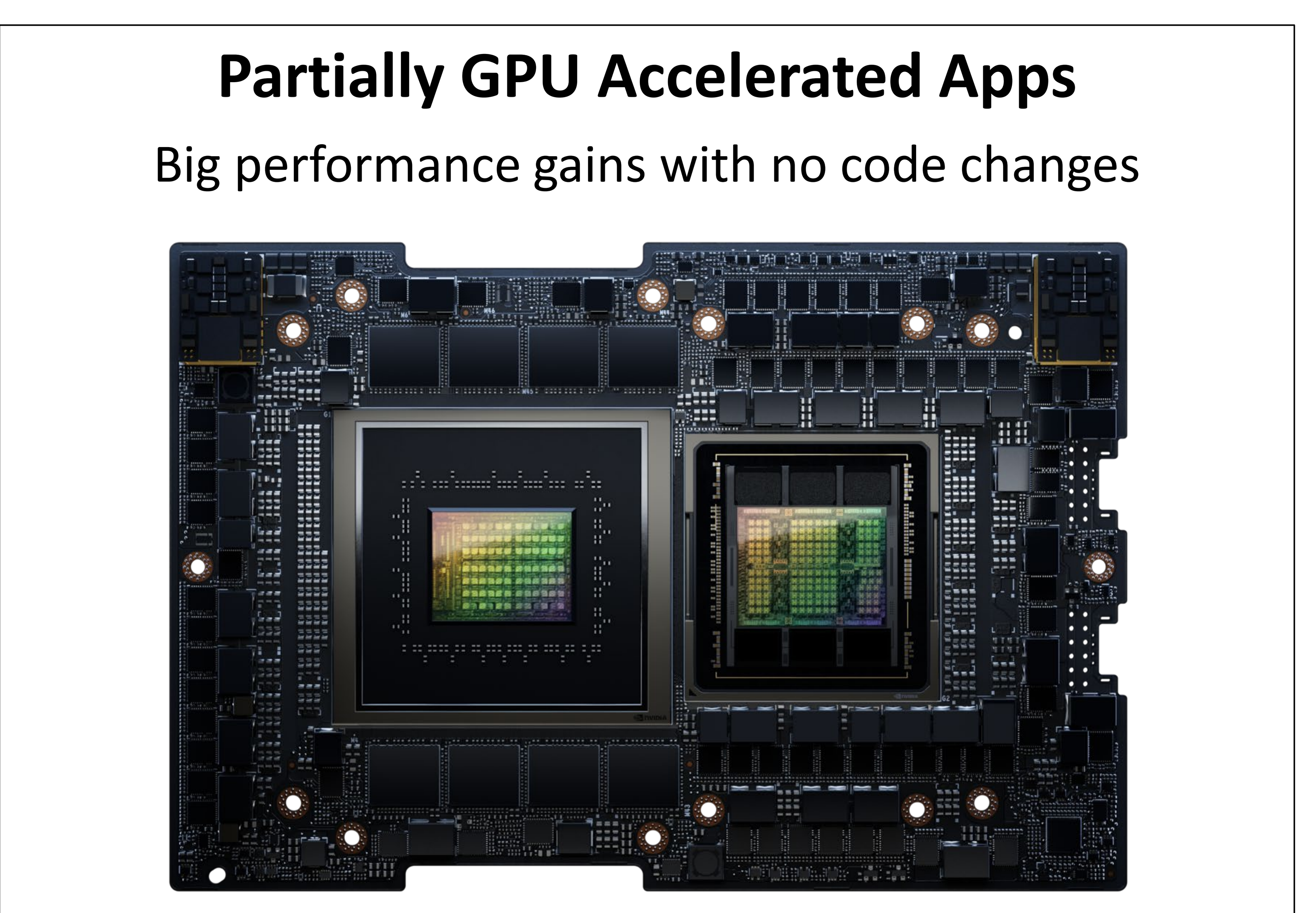


GH200 Grace Hopper HPC Platform

Unified Memory and Cache Coherence for Next Gen HPC Performance

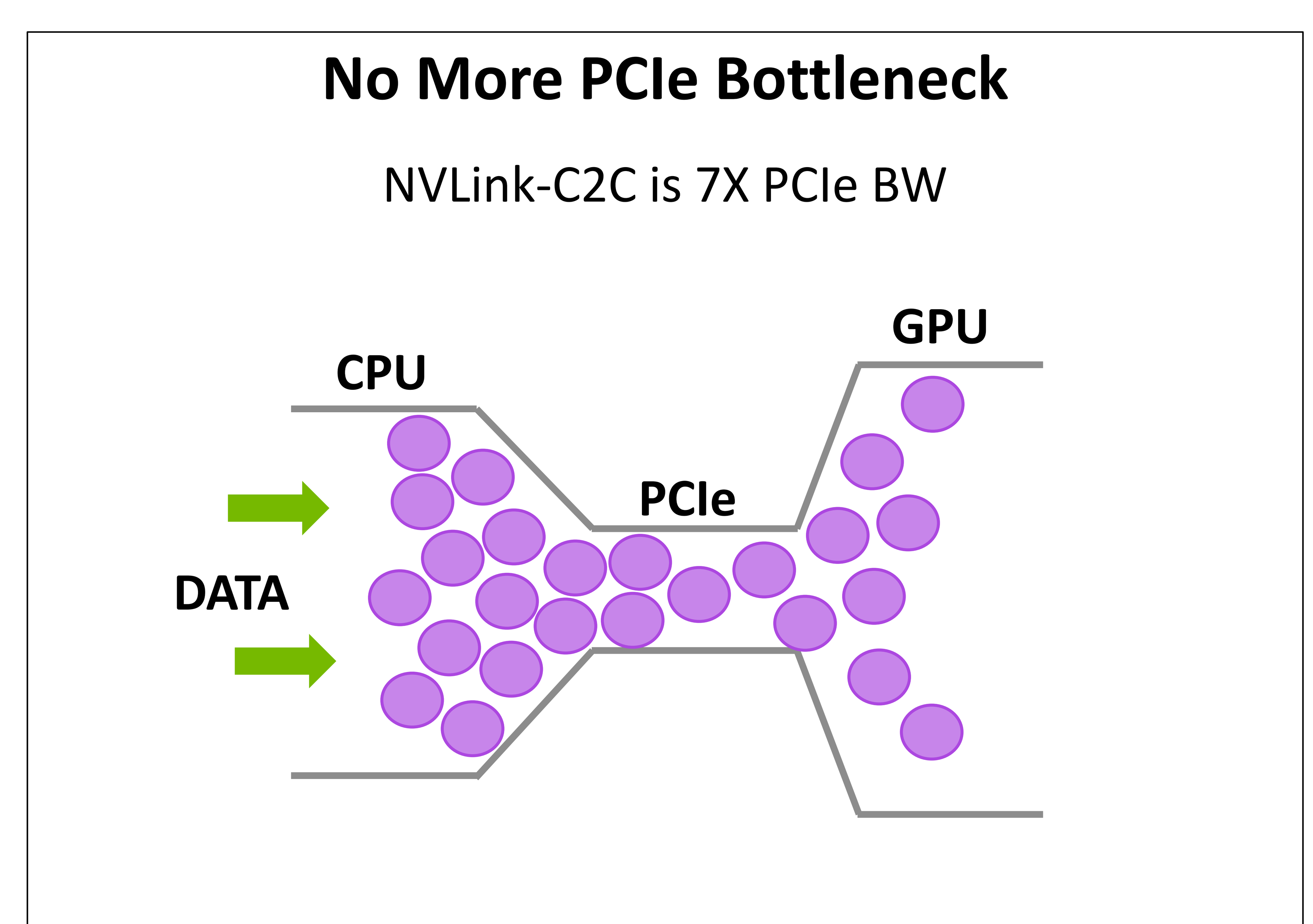
Partially GPU Accelerated Apps

Big performance gains with no code changes



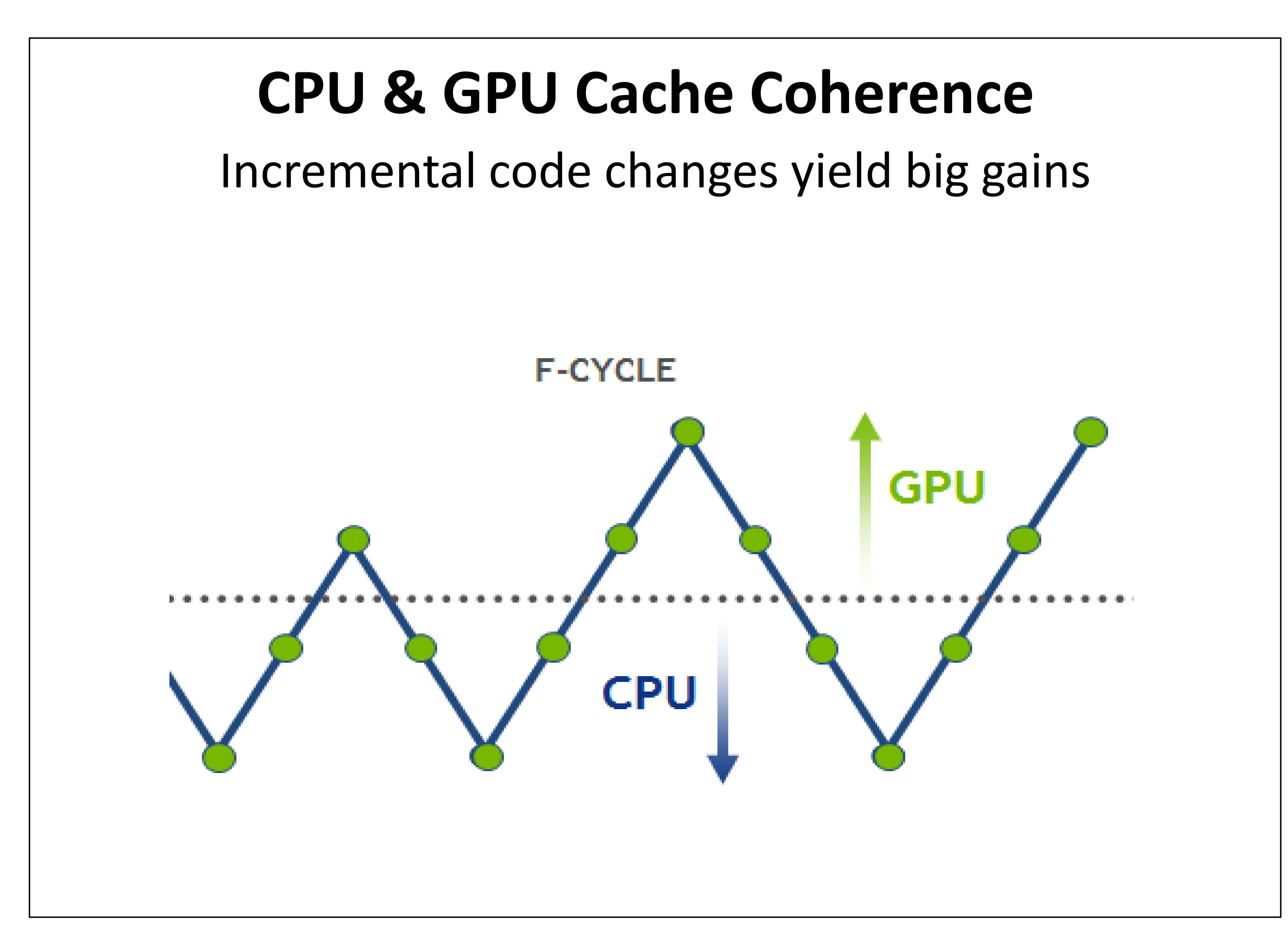
No More PCIe Bottleneck

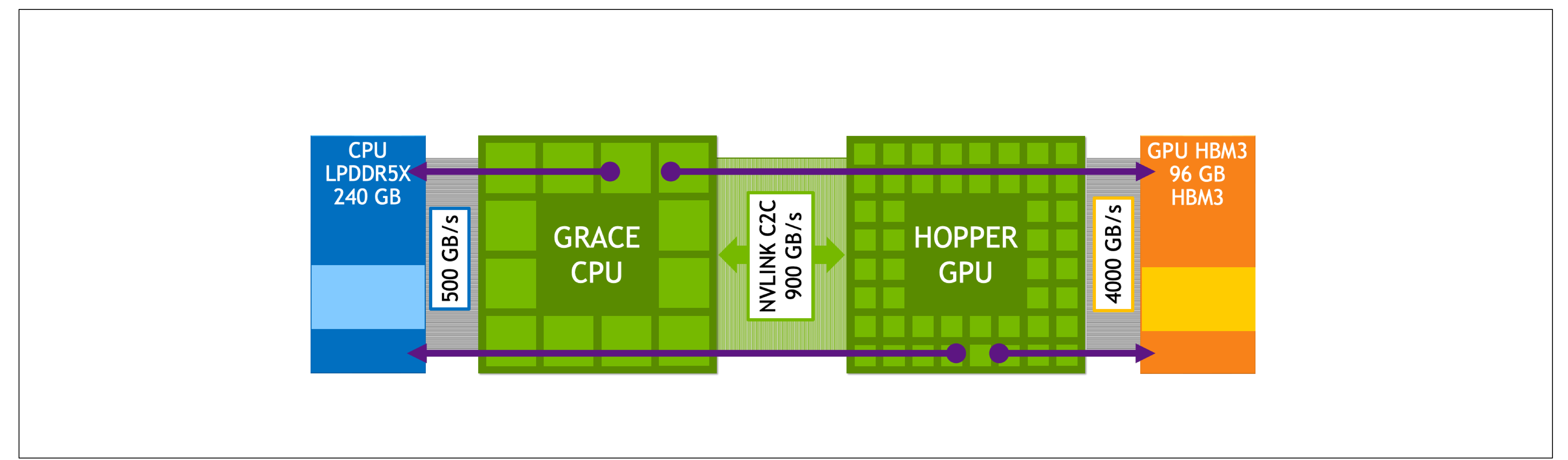
NVLink-C2C is 7X PCIe BW

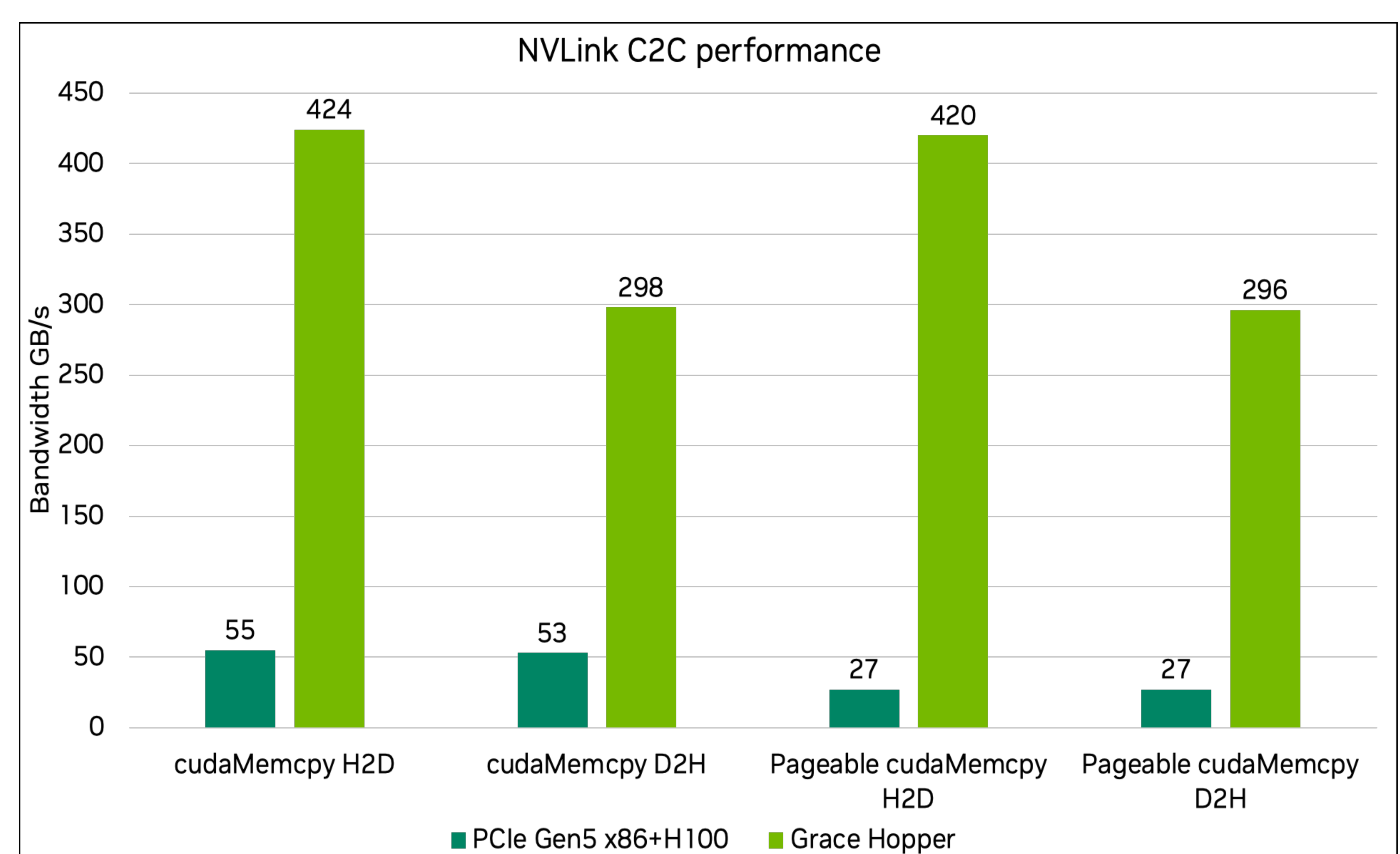


CPU & GPU Cache Coherence

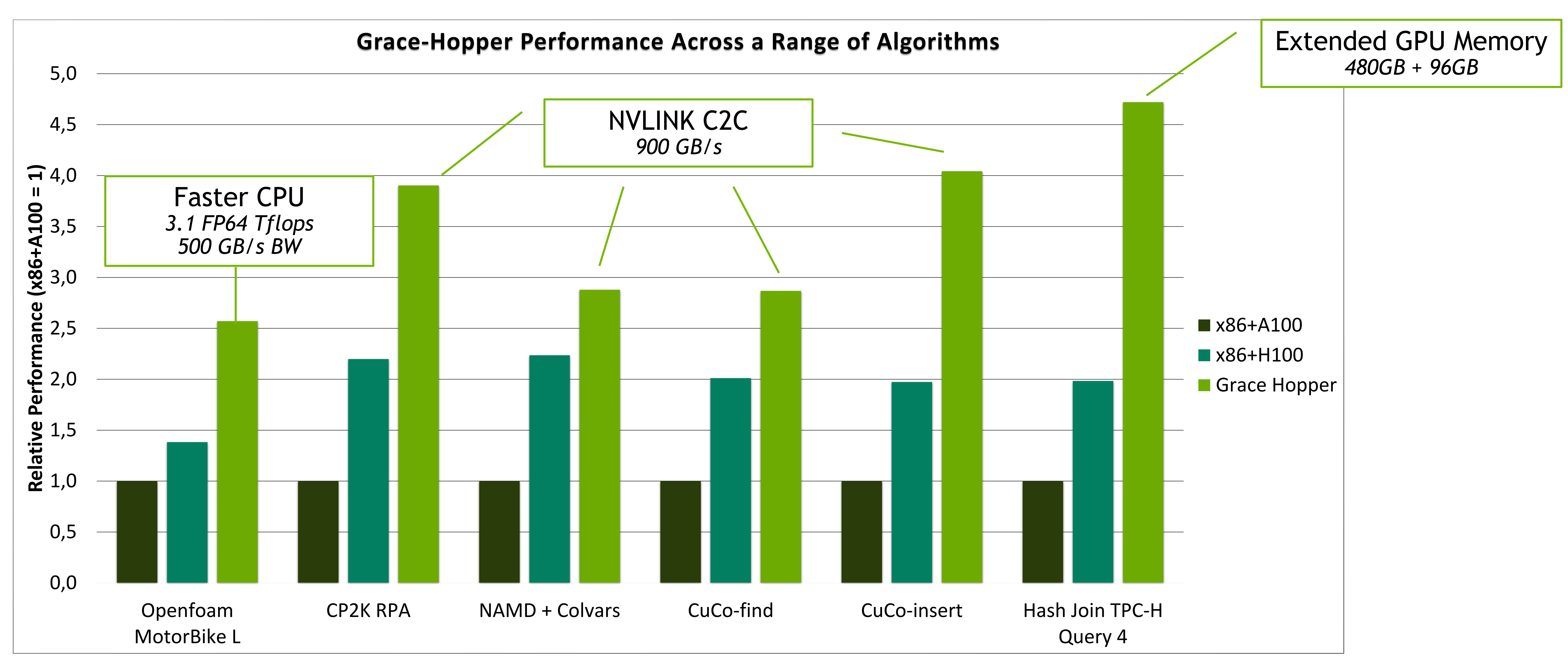
Incremental code changes yield big gains







Grace-Hopper Superchip Workload Performance



NVIDIA CONFIDENTIAL. DO NOT DISTRIBUTE.

Hopper architecture

HOPPER H100 TENSOR CORE GPU

80B Transistors, TSMC 4N

NEW HOPPER SM ARCHITECTURE

- 2x faster FP32 & FP64 FMA
- 256 KB L1$ / Shared Memory
- New 4th Gen Tensor Core
- New DPX instruction set
- New Tensor Memory Accelerator
  - Fully asynchronous data movement
- New Thread Block Clusters
  - Turn locality into efficiency

Tensor Cores

- Specialized high-performance compute cores for matrix multiply and accumulate (MMA) math operations for AI and HPC applications.
- Operating in parallel across SMs in one NVIDIA GPU deliver massive increases in throughput and efficiency compared to standard floating-point (FP), integer (INT), and fused multiply-accumulate (FMA) operations.
- Support for a wide range of data types (fp64, fp32, tf32, fp16, bfloat16, fp8, int8) and mixed precision
- New Transformer Engine designed specifically to accelerate Transformer model training and inference (chooses dynamically between FP8 and 16-bit calculations)
- Tensor Memory Accelerator feeds the H100 Tensor Cores with transfers large blocks of data and multi-dimensional tensors from global memory to shared memory and vice-versa.

NVIDIA CONFIDENTIAL. DO NOT DISTRIBUTE.

FP8 TENSOR CORE

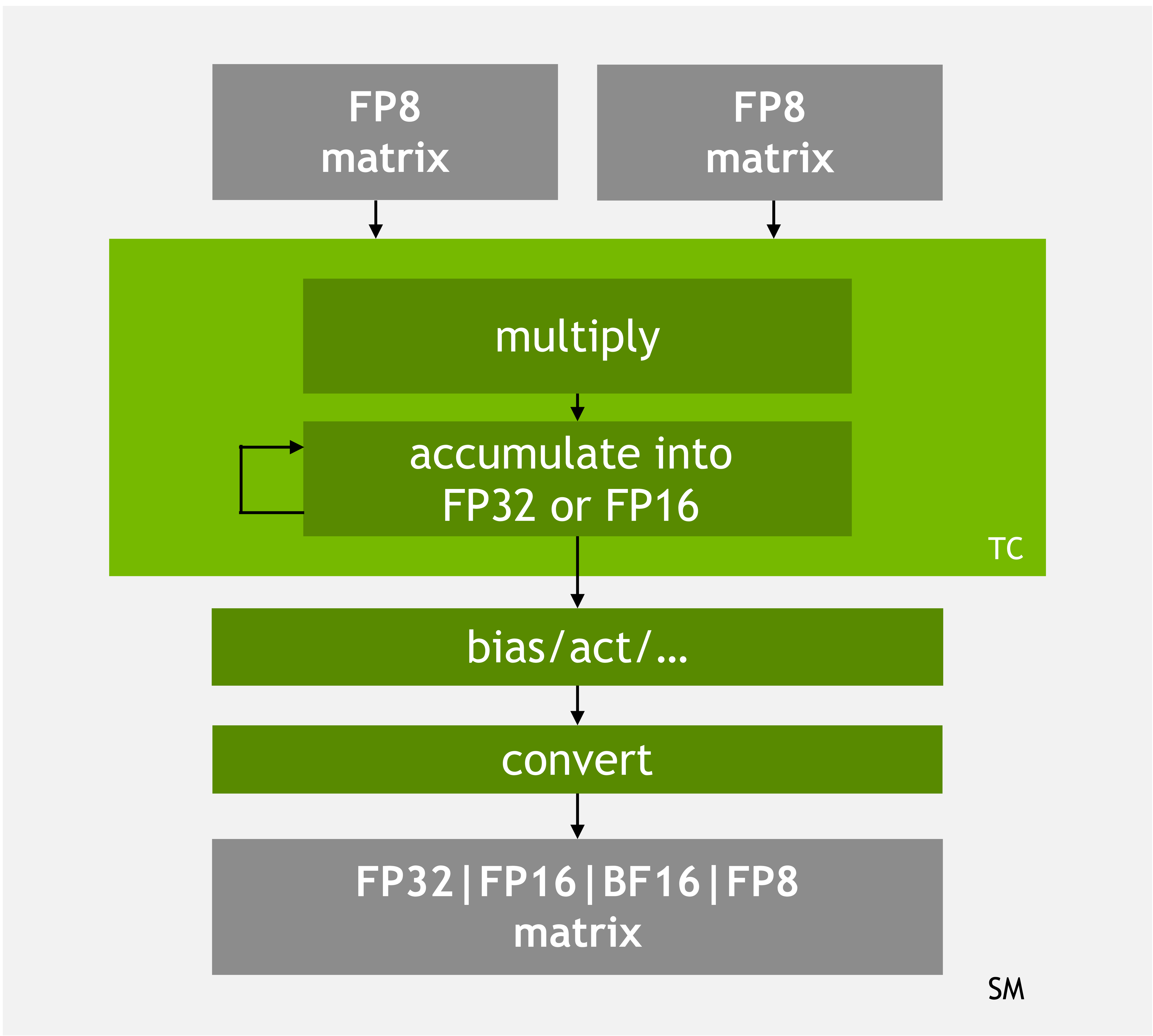
Allocate 1 bit to either range or precision

Range
Precision
sign
exponent
mantissa

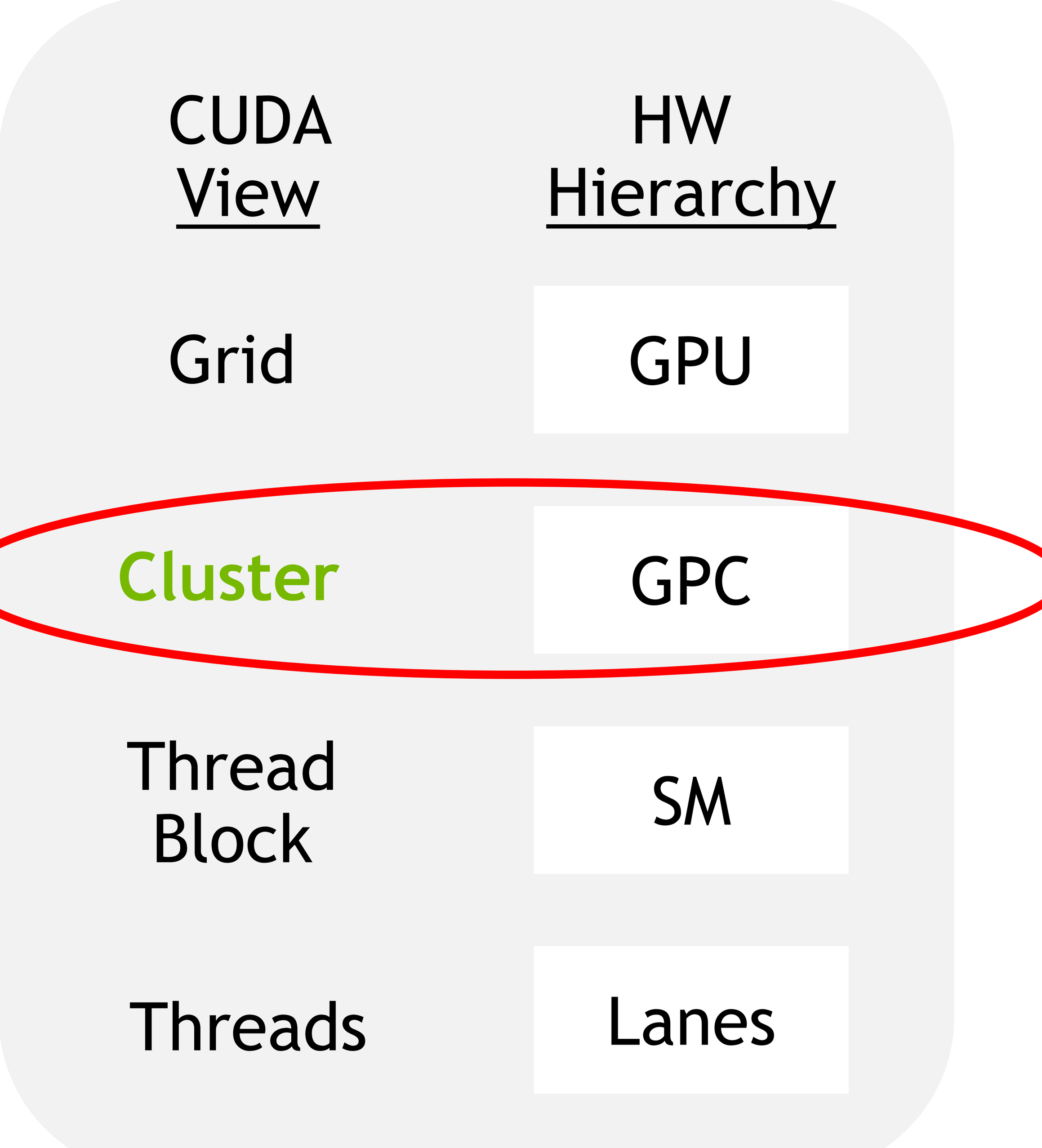
FP32 — s — e8 — m23
BF16 — s — e8 — m7
FP16 — s — e5 — m10
FP8 (E5M2) — s — e5 — m2
FP8 (E4M3) — s — e4 — m3

2x throughput & half footprint of FP16/BF16

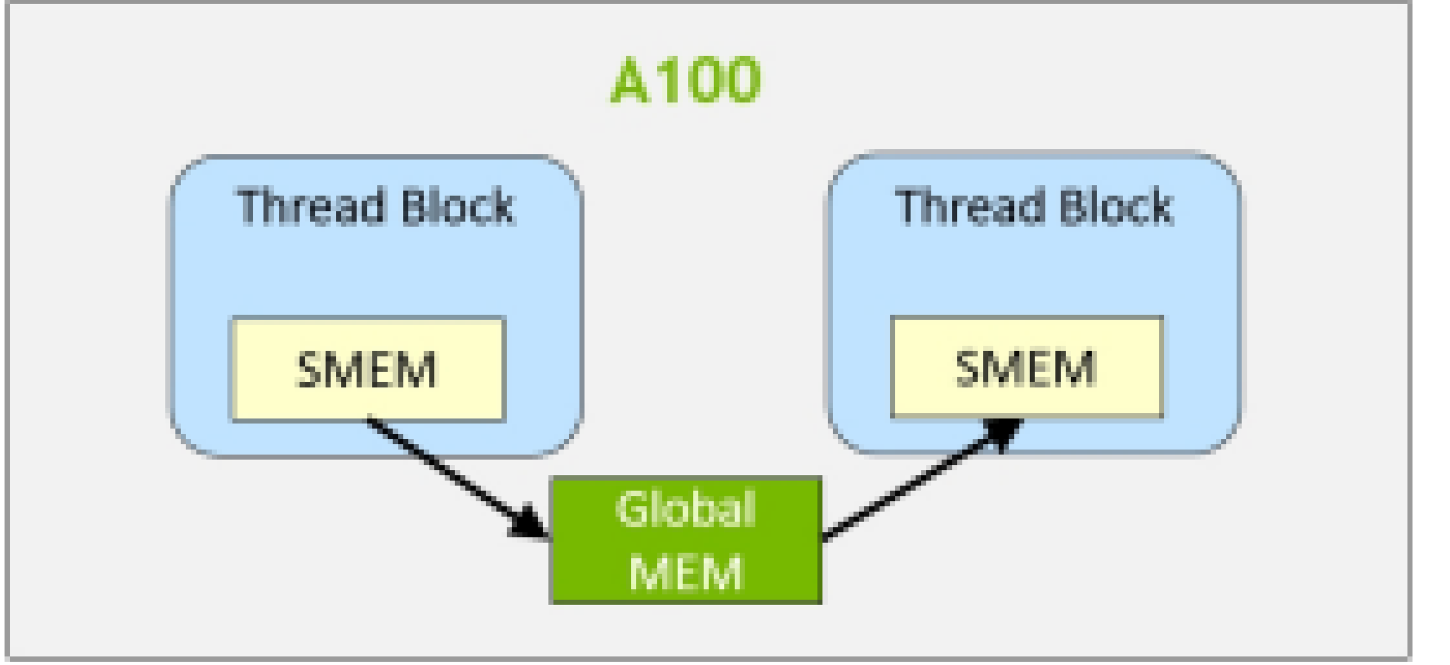
Support for multiple accumulator and output types

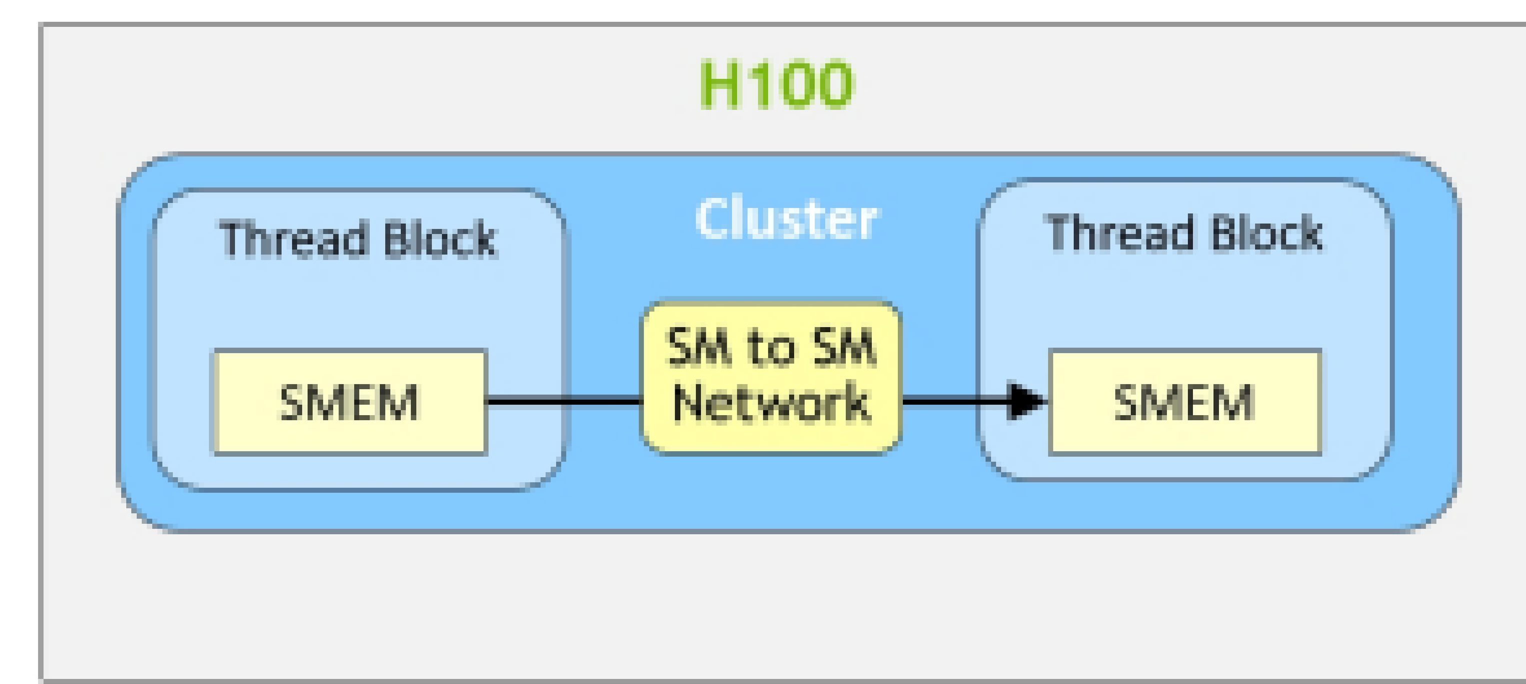


Thread Block Clusters



- New feature introduces programming locality within clusters of SMs
- Shared memory blocks of SMs within a GPU Processing Cluster (GPC) can communicate directly (w/o going to HBM)
- Leveraged with CUDA cooperative groups API





For details, see "NVIDIA H100 Tensor Core GPU Architecture" white paper available for download

TENSOR MEMORY ACCELERATOR UNIT

ASYNC MEM COPY USING TMA

HW-accelerated mem_copies

Global <=> Shared Mem

Shared Mem <=> Shared Mem for Clusters

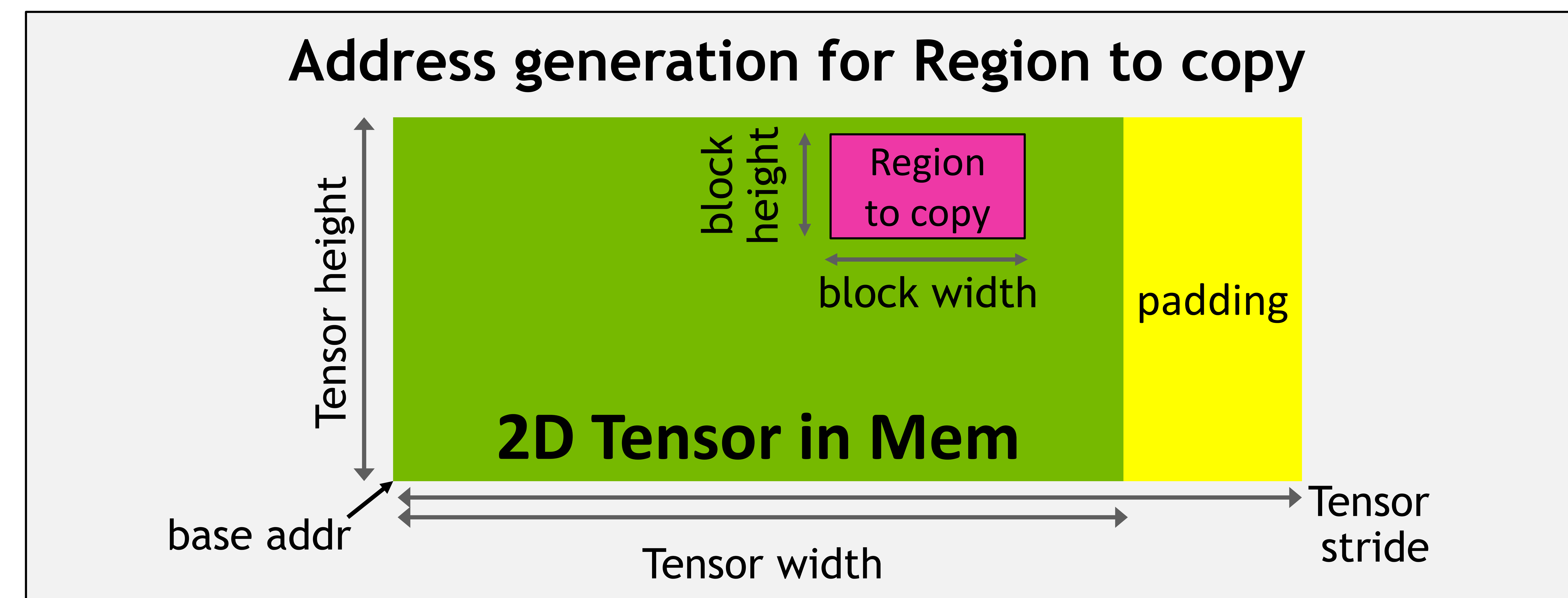
Address generation for 1D to 5D Tensors

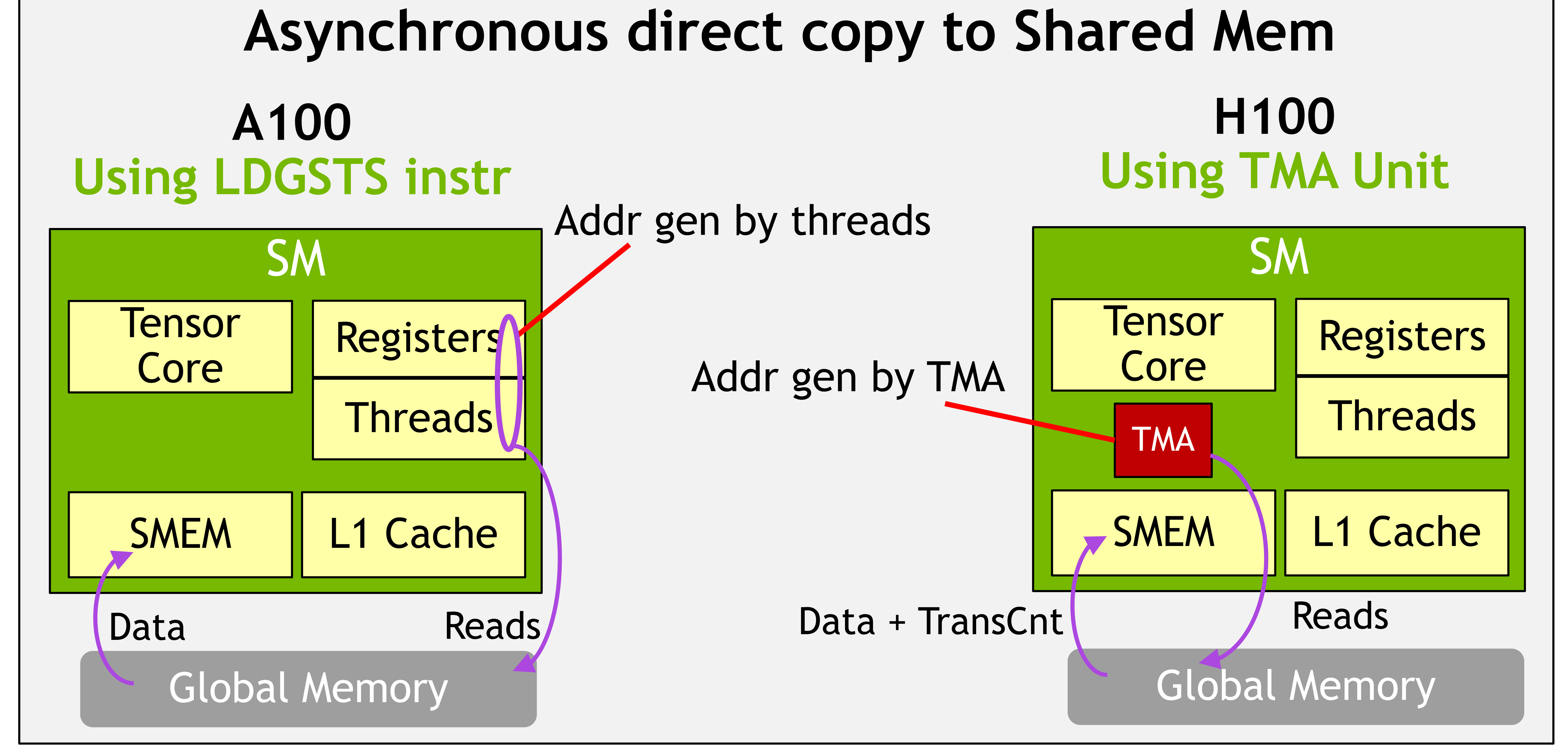
Simplified programming model

Fully asynchronous with threads

No addr gen or data movement overhead

Synchronize with transaction barrier





Software

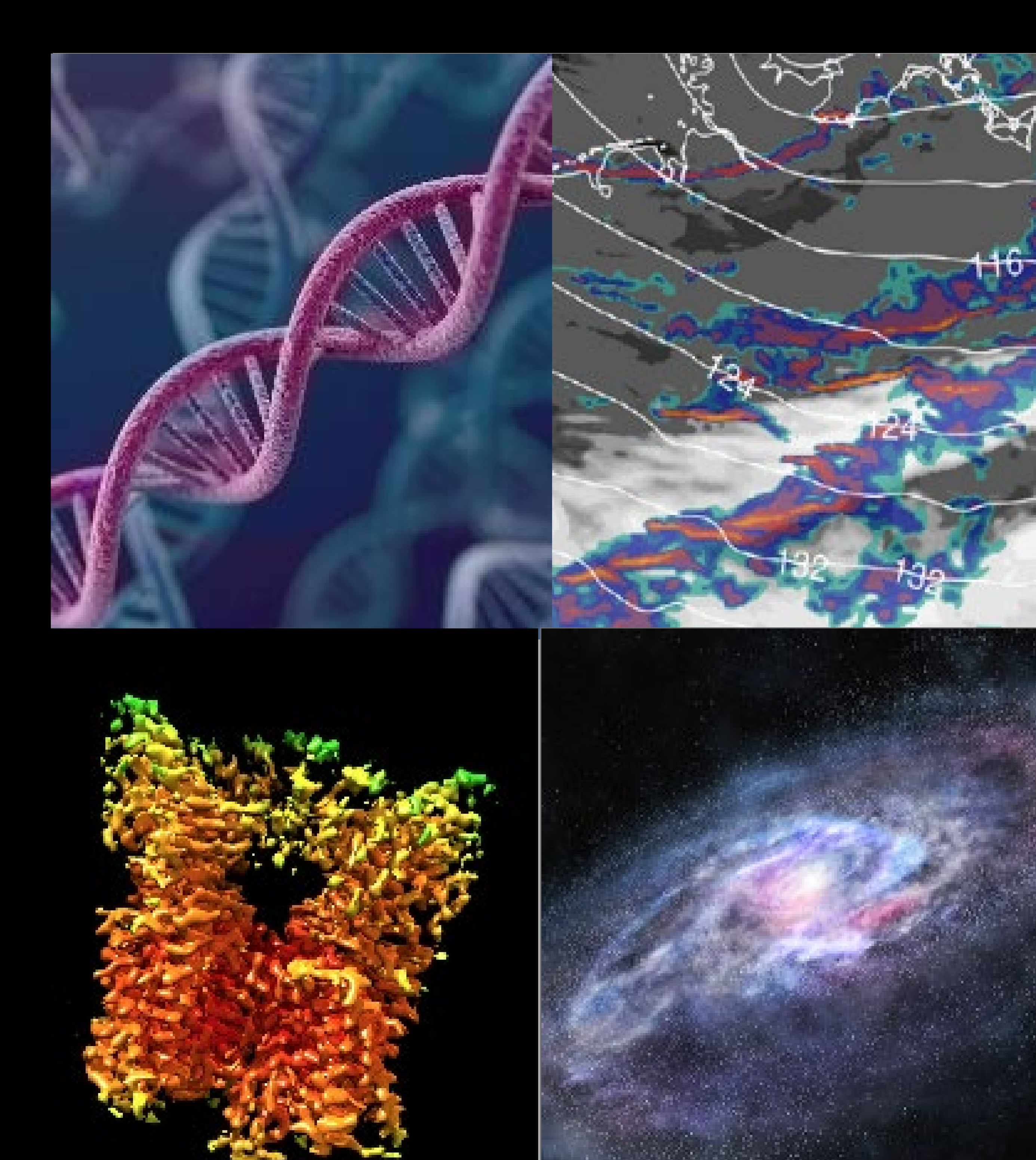
NVIDIA
CUDA

CUDA: NVIDIA's Computing Platform

Used Everywhere

Gaming

Media & Entertainment



Science

http://developer.nvidia.com/cuda-downloads

CUDA TOOLKIT

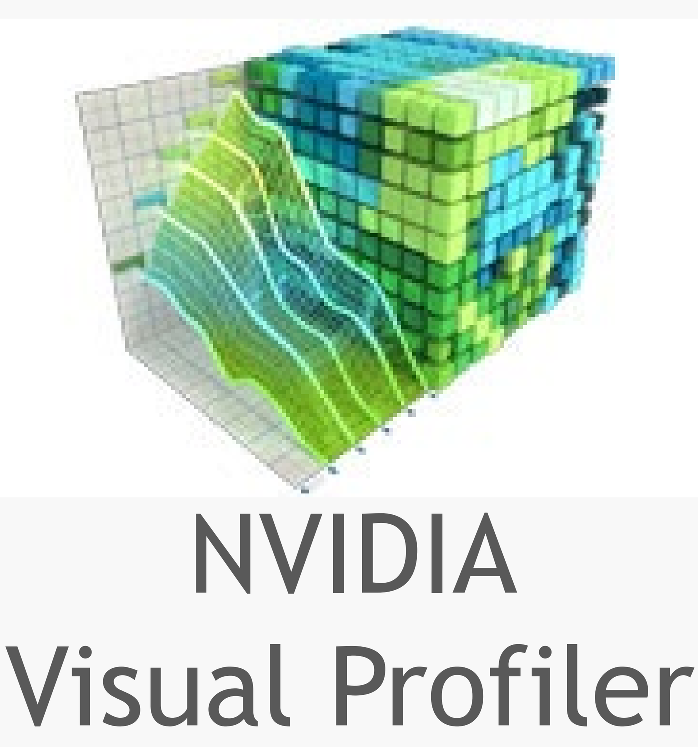
Libraries, Languages and Development Tools for GPU Computing

Programming Approaches

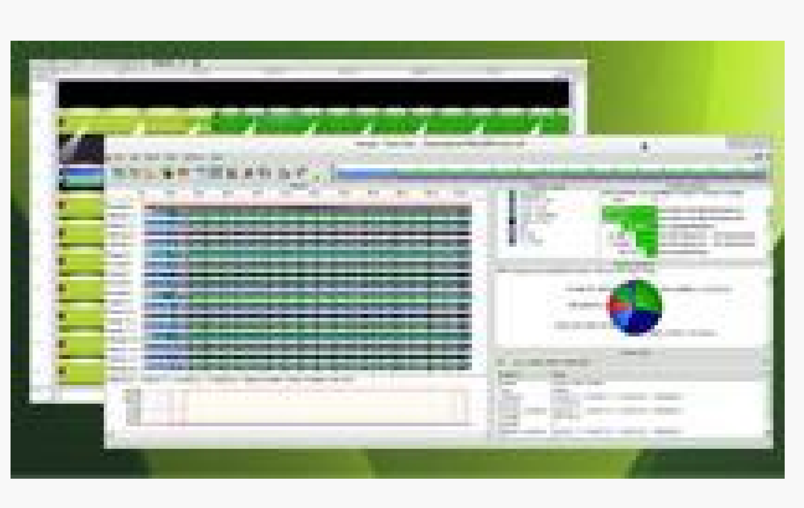
Libraries — "Drop-in" Acceleration

Programming Languages — Maximum Flexibility

Development Environment



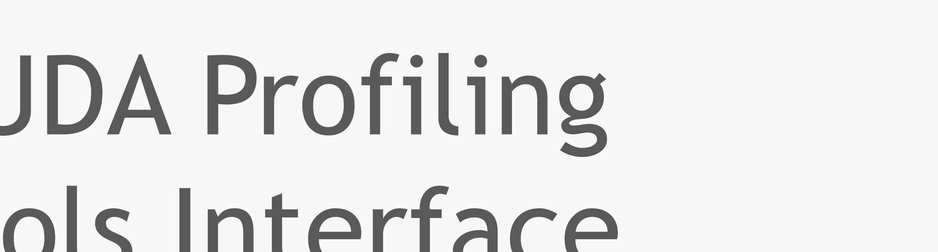
NVIDIA Visual Profiler



CUDA Profiling Tools Interface

CUDA-GDB Debugger

CUDA MEMCHECK



Language Support

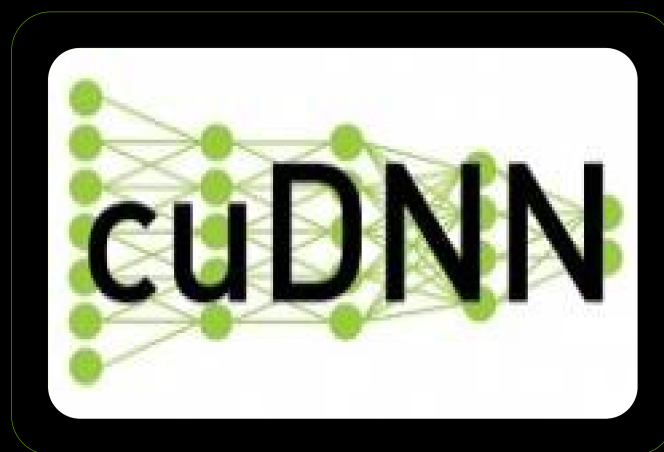
C C++ Fortran

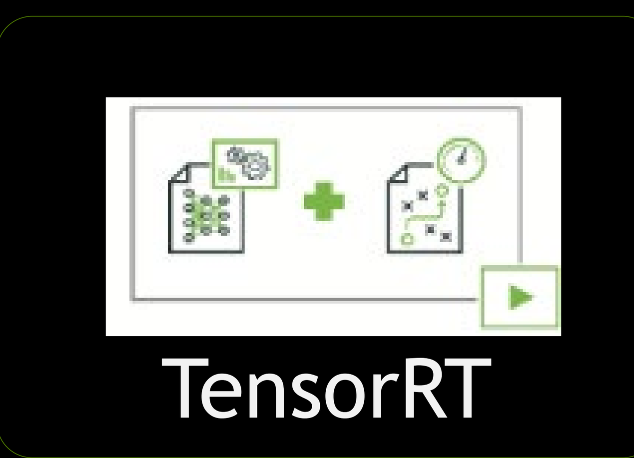
Compile new languages to CUDA

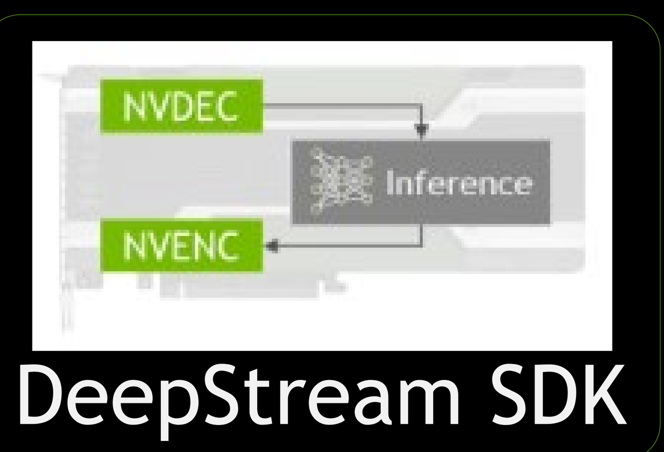
GPU Accelerated Libraries

"Drop-In" Acceleration For Your Applications

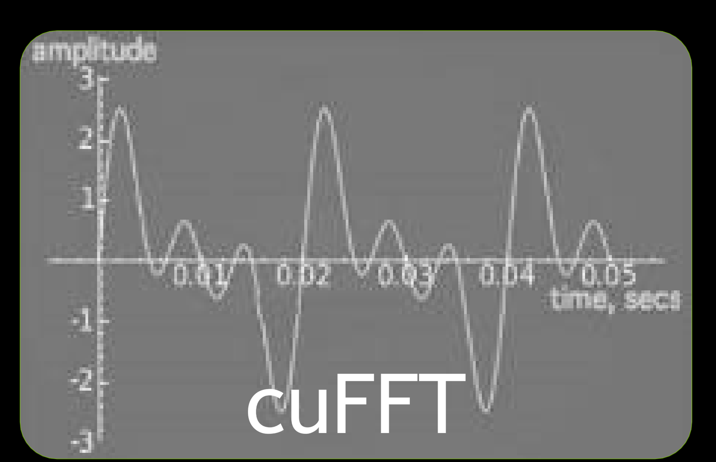
DEEP LEARNING



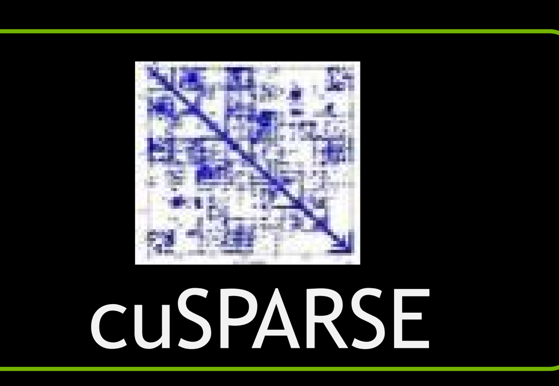


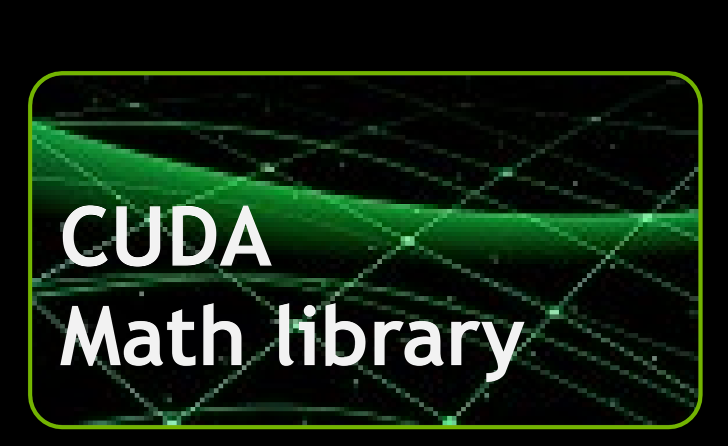


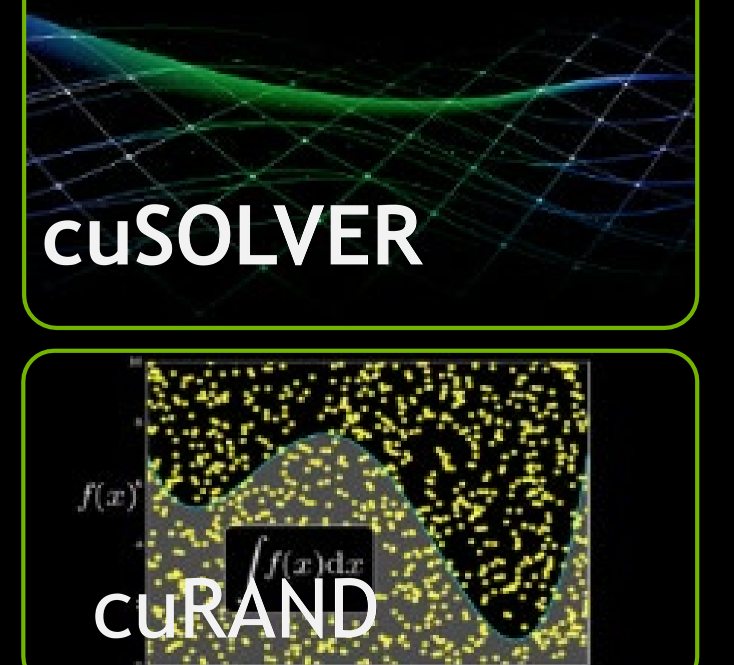
SIGNAL, IMAGE & VIDEO



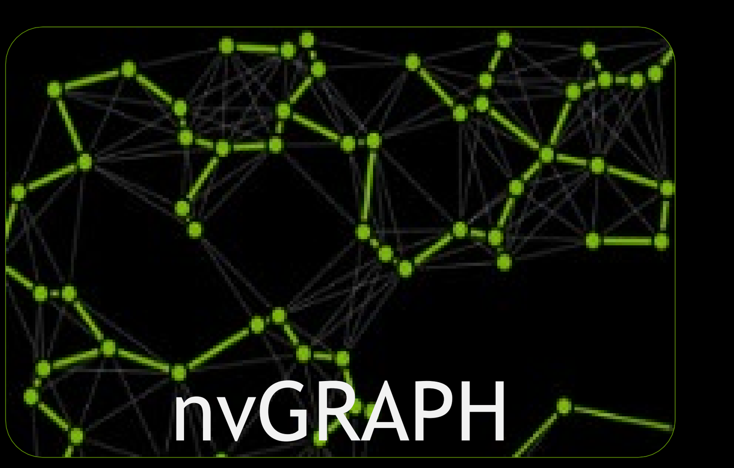
LINEAR ALGEBRA

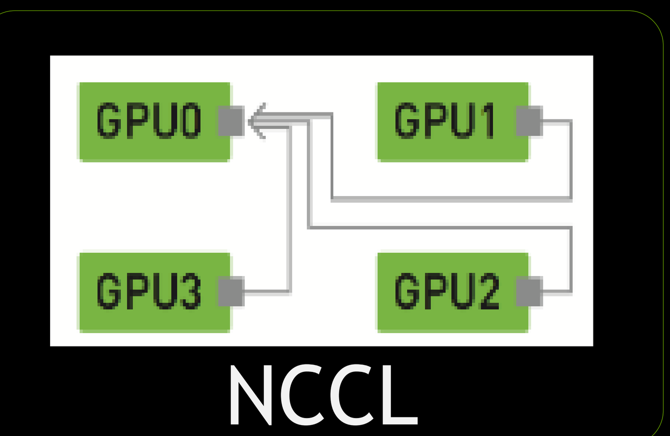






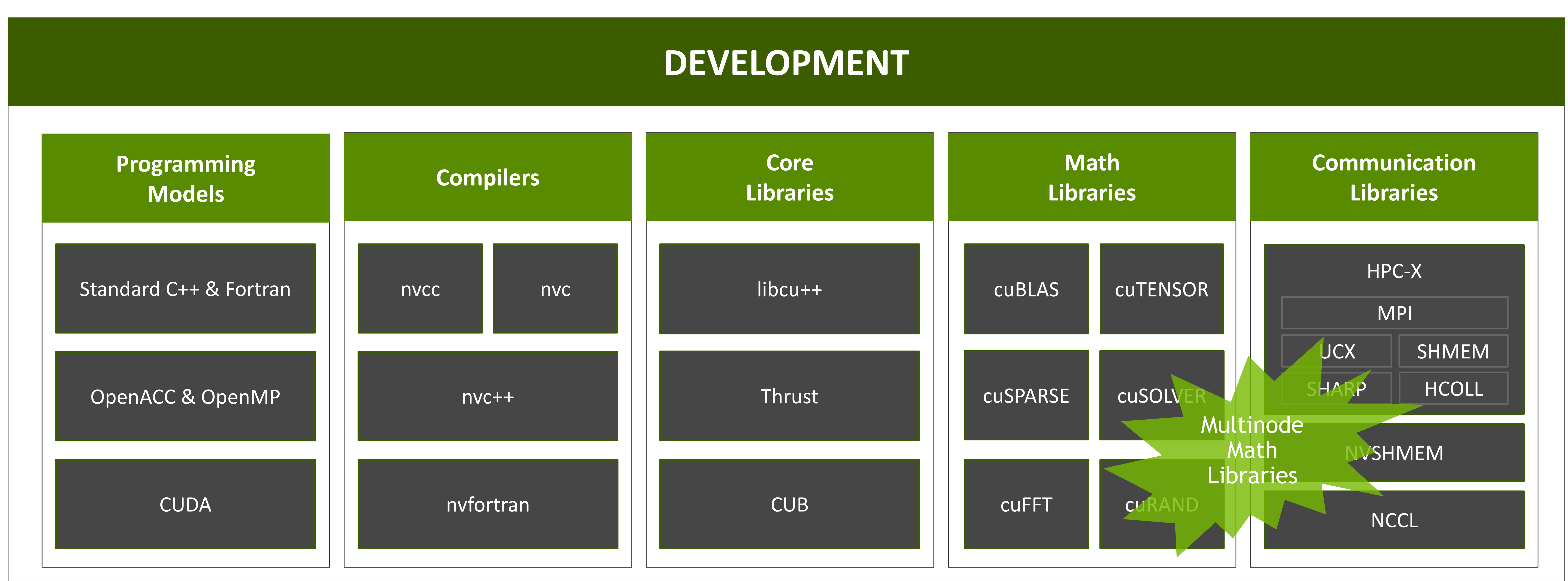
PARALLEL ALGORITHMS

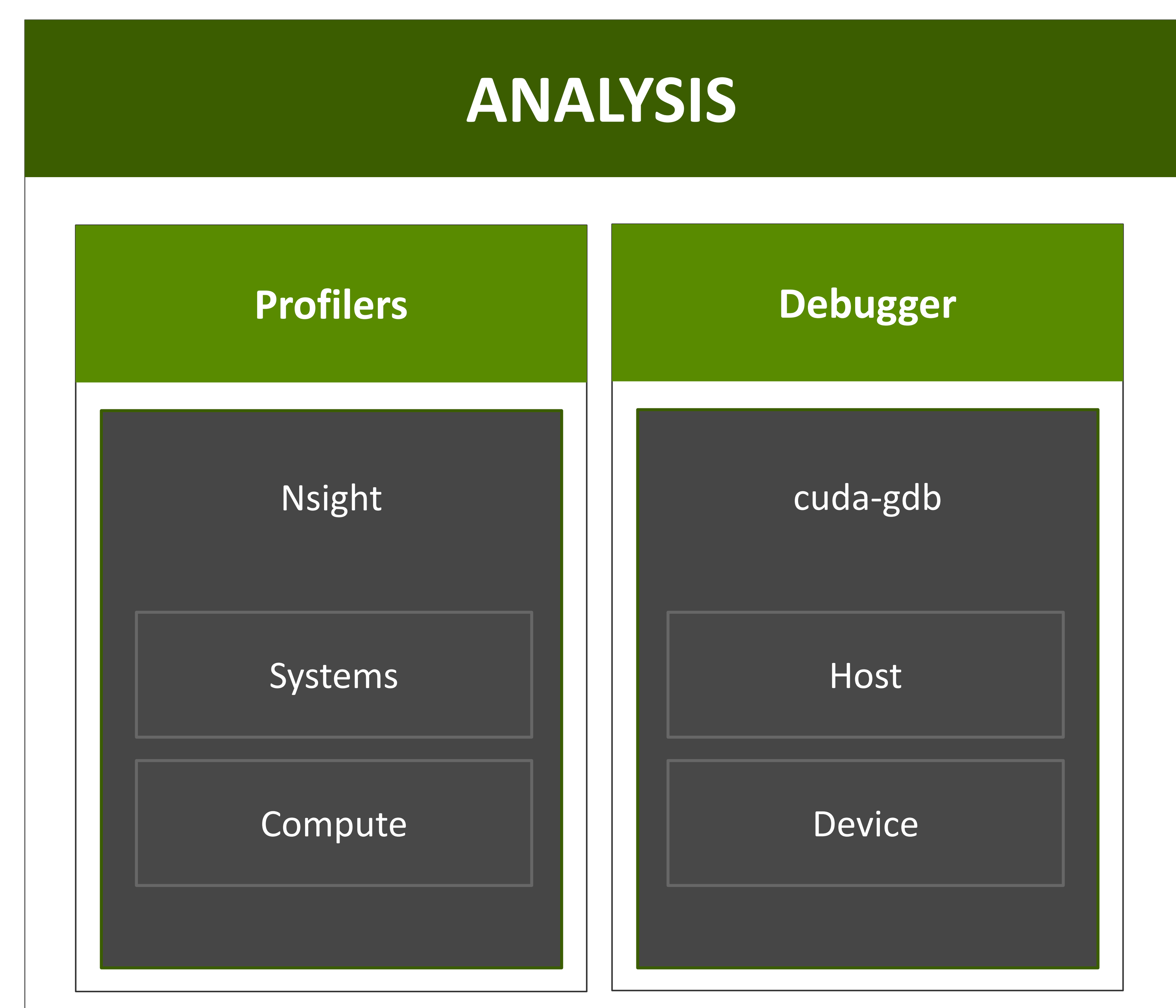




NVIDIA HPC SDK

Available at developer.nvidia.com/hpc-sdk, on NGC, via Spack, and in the Cloud





Develop for the NVIDIA Platform: GPU, CPU and Interconnect

Libraries | Accelerated C++ and Fortran | Directives | CUDA

x86_64 | Arm | OpenPOWER

7-8 Releases Per Year | Freely Available

OK, but, What Does It Mean?

How GPU Acceleration Works

Application Code

Rest of Sequential CPU Code

Compute-Intensive Functions

5% of Code
95% of Execution

GPU

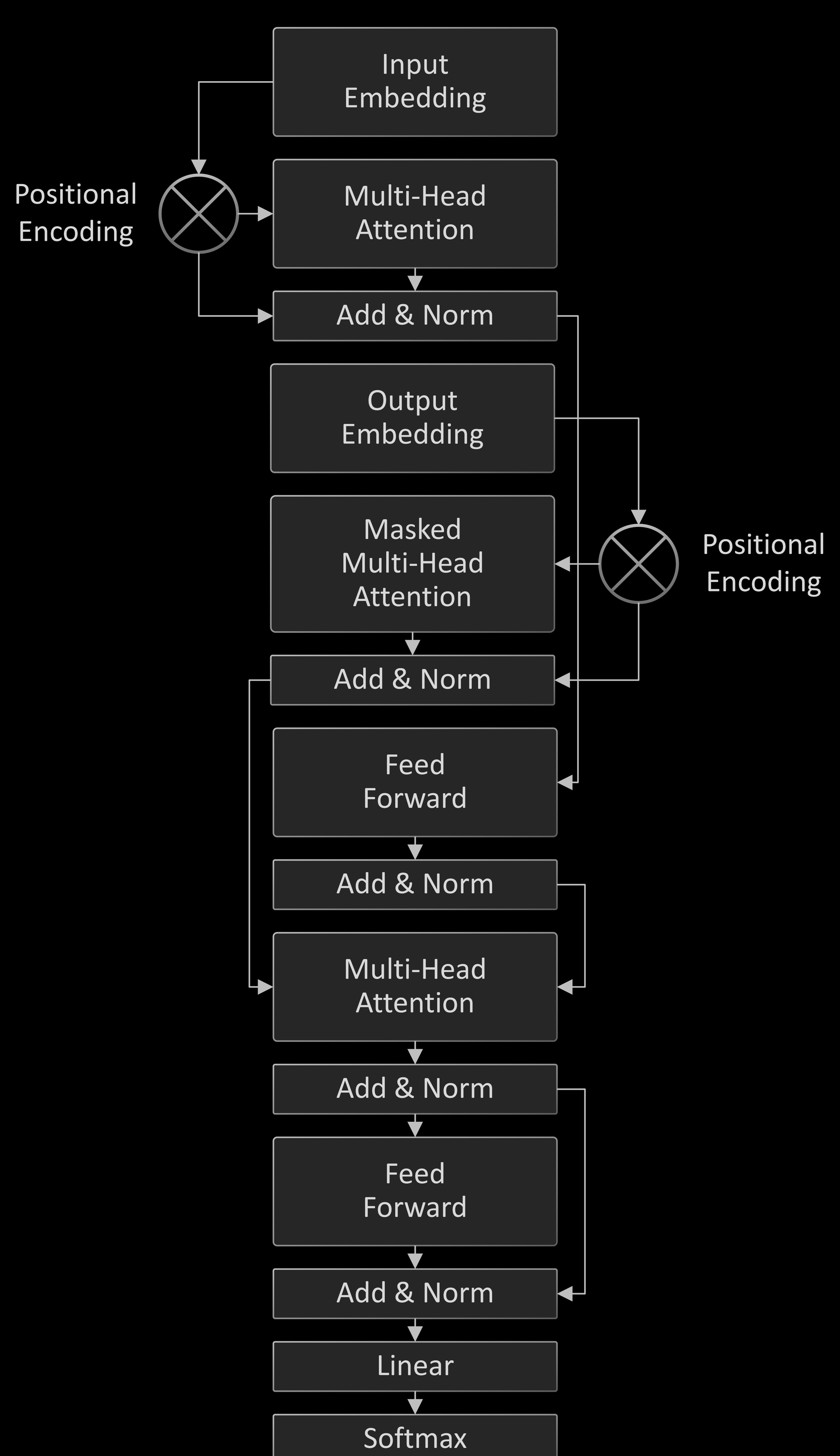
CPU

95% of Code
5% of Execution

+

Piece-by-Piece, not All-or-Nothing

Incrementally accelerate key components of an application



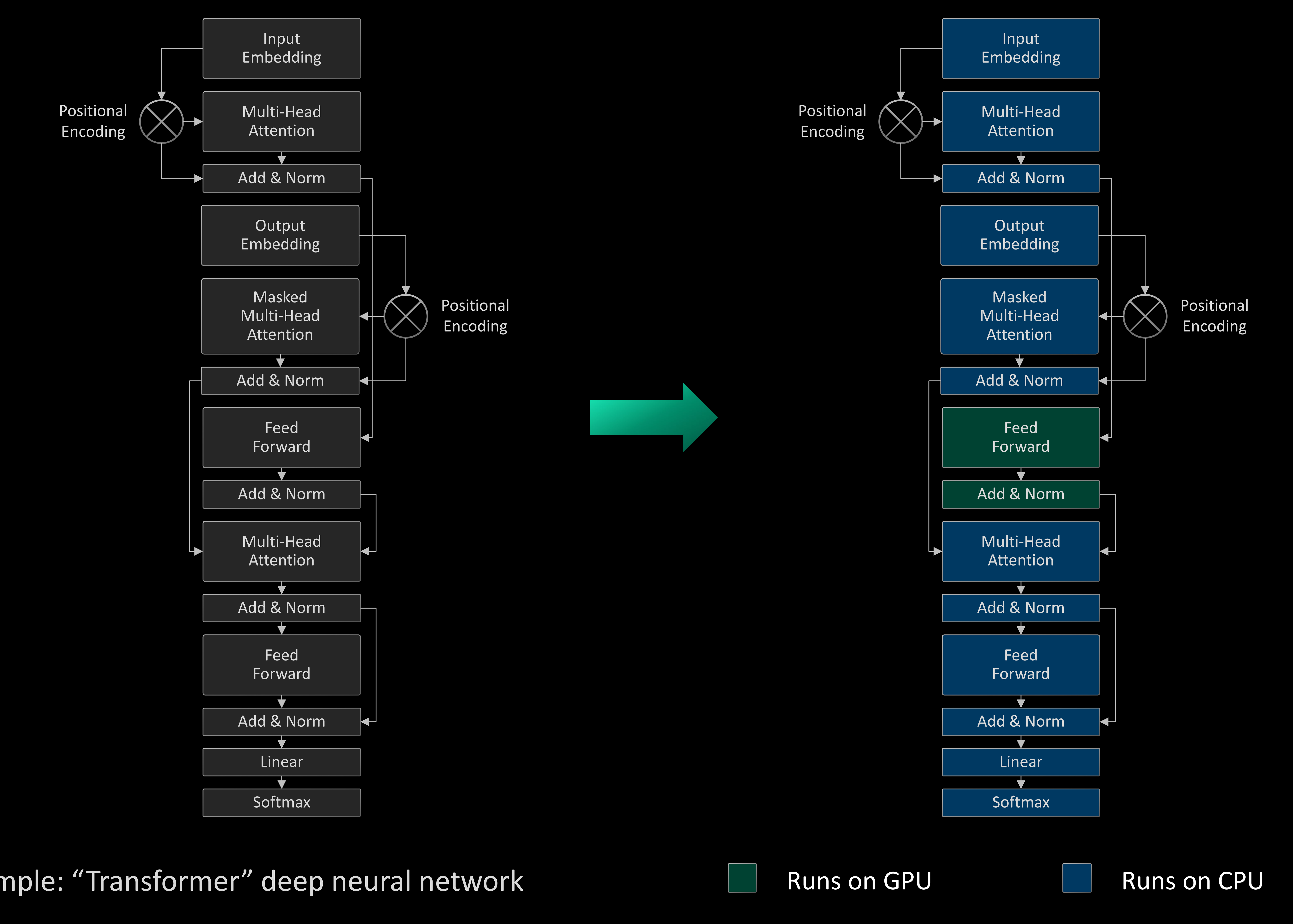
Example: “Transformer” deep neural network

Real applications are complex

No need to port the whole thing in one go

Piece-by-Piece, not All-or-Nothing

Incrementally accelerate key components of an application



Example: “Transformer” deep neural network

CUDA includes heterogeneous profiling tools to help evaluate which components to port next

Piece-by-Piece, not All-or-Nothing

Incrementally accelerate key components of an application

Input Embedding
Positional Encoding
Multi-Head Attention
Add & Norm
Output Embedding
Masked Multi-Head Attention
Positional Encoding
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Feed Forward
Add & Norm
Linear
Softmax

Example: "Transformer" deep neural network

Runs on GPU

Runs on CPU

Piece-by-Piece, not All-or-Nothing

Incrementally accelerate key components of an application

Input Embedding
Positional Encoding
Multi-Head Attention
Add & Norm
Output Embedding
Masked Multi-Head Attention
Positional Encoding
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Feed Forward
Add & Norm
Linear
Softmax

Example: "Transformer" deep neural network

Runs on GPU

Runs on CPU

Piece-by-Piece, not All-or-Nothing

Incrementally accelerate key components of an application

Input Embedding
Positional Encoding
Multi-Head Attention
Add & Norm
Output Embedding
Masked Multi-Head Attention
Positional Encoding
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Feed Forward
Add & Norm
Linear
Softmax

Input Embedding
Positional Encoding
Multi-Head Attention
Add & Norm
Output Embedding
Masked Multi-Head Attention
Positional Encoding
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Feed Forward
Add & Norm
Linear
Softmax

Input Embedding
Positional Encoding
Multi-Head Attention
Add & Norm
Feed Forward
Add & Norm
Output Embedding
Masked Multi-Head Attention
Positional Encoding
Add & Norm
Multi-Head Attention
Add & Norm
Feed Forward
Add & Norm
Linear
Softmax

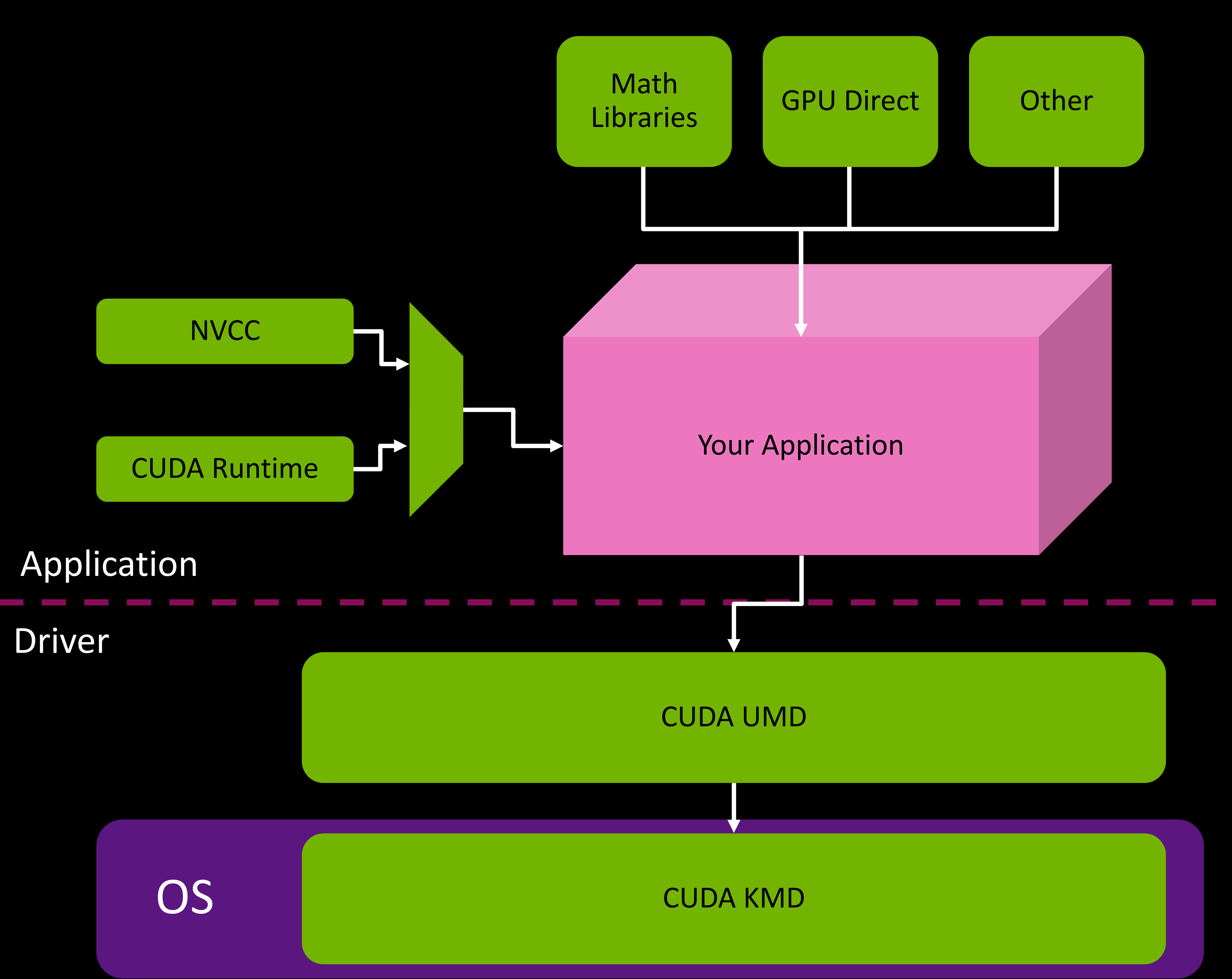
Example: “Transformer” deep neural network

Runs on GPU

Runs on CPU

What is CUDA

A Simplified View



- Domain-Specific Libraries
- CUDA Programming Environment
  - Compiler
  - CUDA programming model (will talk more about this)
- CUDA Runtime Libraries
- Driver:
  - **Kernel Mode Driver –** Lives in the OS, handles low-level hardware interaction
  - **User Mode Driver –** Integrates with your application, maps low-level CUDA API calls to your specific HW

But, Of Course, a Real Application is Complex

Many Components, Many Dependencies

Math Libraries
GPU Direct
Other
NVCC
CUDA Runtime
Your Application
Application
Driver
CUDA UMD
OS
CUDA KMD

Your Application
Mod A
Mod B
Mod C
Mod D
Mod E
PyTorch
PyTorch
cuDNN
cuFFT
cuDNN
cuBLAS
cuBLAS
CUDA UMD
OS
CUDA KMD

4 Ways To Accelerate Applications

Libraries	Directives (OpenACC, OpenMP)	Standard Languages	Native Programming Languages
"Drop-in" Acceleration	Easily Accelerate Applications	Portable Performance	Maximum Flexibility

4 Ways To Accelerate Applications

Applications

Libraries	Directives (OpenACC, OpenMP)	Standard Languages	Native Programming Languages
"Drop-in" Acceleration	Easily Accelerate Applications	Portable Performance	Maximum Flexibility

Libraries: Easy, High-Quality Acceleration

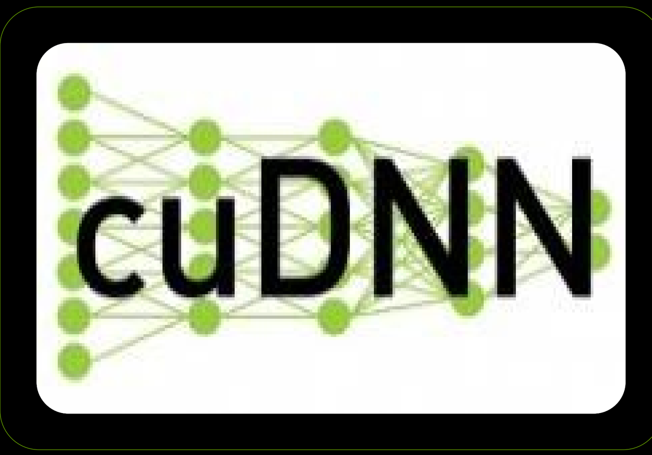
EASE OF USE	Using libraries enables GPU acceleration without in-depth knowledge of GPU programming
“DROP-IN”	Many GPU-accelerated libraries follow standard APIs, thus enabling acceleration with minimal code changes
QUALITY	Libraries offer high-quality implementations of functions encountered in a broad range of applications
PERFORMANCE	NVIDIA libraries are tuned by experts

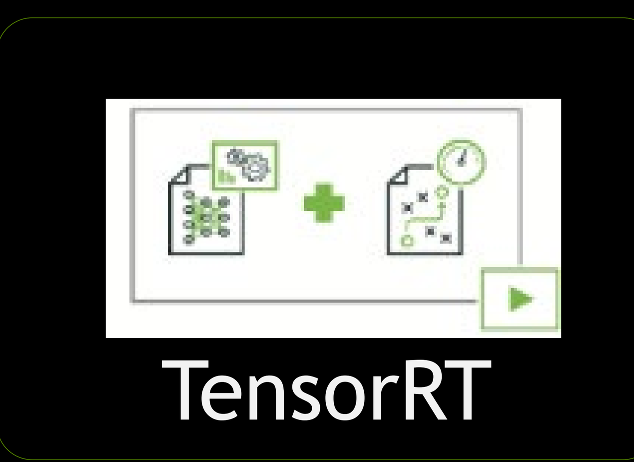
GPU Accelerated Libraries

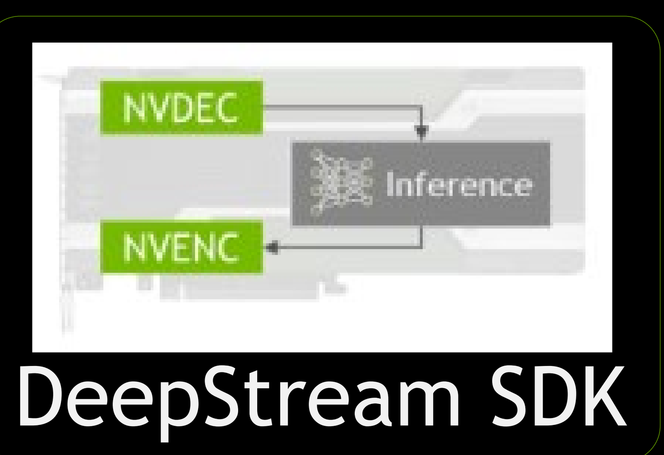
“Drop-In” Acceleration For Your Applications

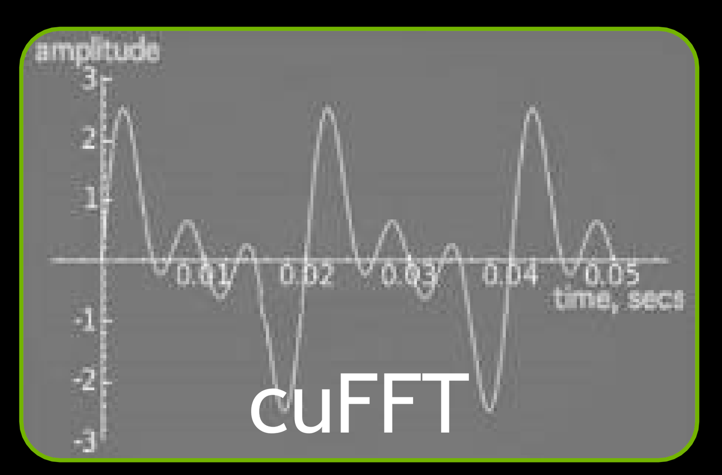
DEEP LEARNING

SIGNAL, IMAGE & VIDEO

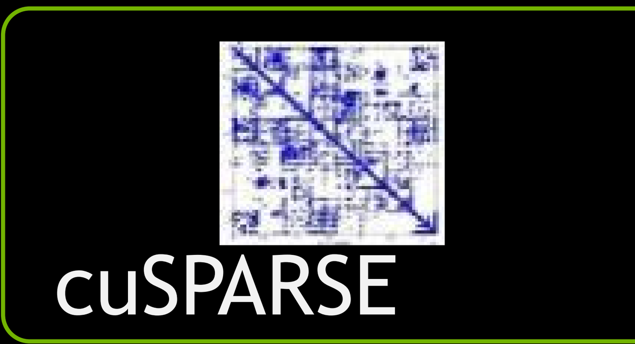


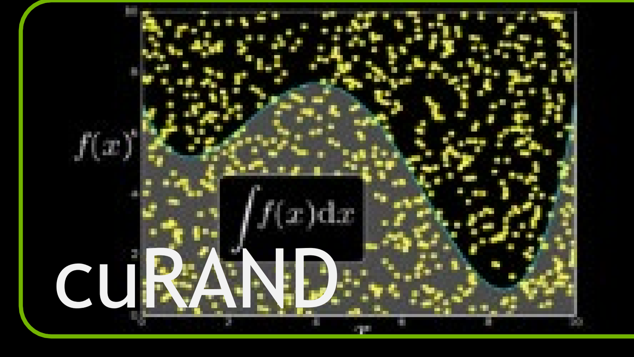




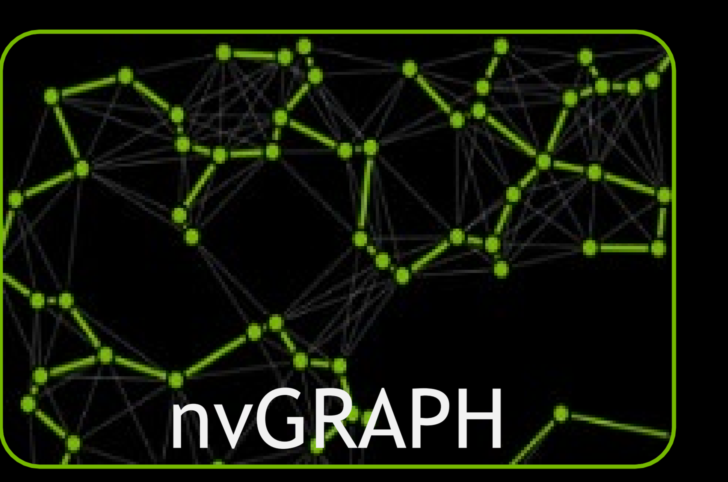


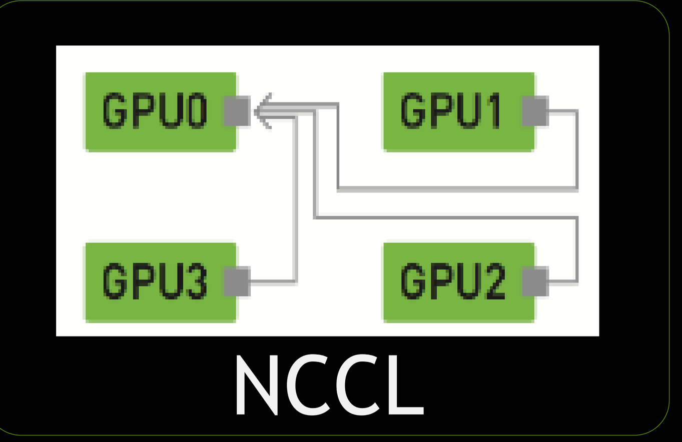
LINEAR ALGEBRA





PARALLEL ALGORITHMS





3 Steps To A CUDA-Accelerated Application

STEP 1	Substitute library calls with equivalent CUDA library calls `saxpy ( ... )` → `cublasSaxpy ( ... )`
STEP 2	Manage data locality with CUDA: `cudaMallocManaged,() cudaMalloc(), cudaMemcpy()` with CUBLAS: `cublasAlloc(), cublasSetVector()`
STEP 3	Rebuild and link the application with the CUDA-accelerated library `$ gcc myobj.o –l cublas`

Single Precision Alpha X Plus Y (SAXPY)

Part of Basic Linear Algebra Subroutines (BLAS) library

$$z = \alpha x + y$$

x, y, z : vector

α : scalar

Drop-In Acceleration With CUDA Maths Libraries

In two easy steps

```
int N = 1 << 20;            // 1M elements

x = (float *)malloc(N * sizeof(float));
y = (float *)malloc(N * sizeof(float));
initData(x, y);

// Perform SAXPY on 1M elements: y[]=a*x[]+y[]
saxpy(N, 2.0, x, 1, y, 1);

useResult(y);
```

Original Code

```
int N = 1 << 20;            // 1M elements

x = (float *)malloc(N * sizeof(float));
y = (float *)malloc(N * sizeof(float));
initData(x, y);

// Perform SAXPY on 1M elements: y[]=a*x[]+y[]
saxpy(N, 2.0, x, 1, y, 1);

useResult(y);
```

GPU-Accelerated Code

Drop-In Acceleration With CUDA Maths Libraries

Step 1: Update memory allocation to be CUDA-aware

```
int N = 1 << 20;            // 1M elements

x = (float *)malloc(N * sizeof(float));
y = (float *)malloc(N * sizeof(float));
initData(x, y);

// Perform SAXPY on 1M elements: y[]=a*x[]+y[]
saxpy(N, 2.0, x, 1, y, 1);

useResult(y);
```

Original Code

```
int N = 1 << 20;            // 1M elements

cudaMallocManaged(&x, N * sizeof(float));
cudaMallocManaged(&y, N * sizeof(float));
initData(x, y);

// Perform SAXPY on 1M elements: y[]=a*x[]+y[]
saxpy(N, 2.0, x, 1, y, 1);

useResult(y);
```

GPU-Accelerated Code

Here, we use Unified Memory which automatically migrates
between host (CPU) and device (GPU) as needed by the program

Drop-In Acceleration With CUDA Maths Libraries

Step 2: Call CUDA library version of API

```
int N = 1 << 20;              // 1M elements

x = (float *)malloc(N * sizeof(float));
y = (float *)malloc(N * sizeof(float));
initData(x, y);

// Perform SAXPY on 1M elements: y[]=a*x[]+y[]
saxpy(N, 2.0, x, 1, y, 1);

useResult(y);
```

Original Code

```
int N = 1 << 20;              // 1M elements

cudaMallocManaged(&x, N * sizeof(float));
cudaMallocManaged(&y, N * sizeof(float));
initData(x, y);

// Perform SAXPY on 1M elements: y[]=a*x[]+y[]
cublasSaxpy(N, 2.0, x, 1, y, 1);

useResult(y);
```

GPU-Accelerated Code

Here, we use Unified Memory which automatically migrates between host (CPU) and device (GPU) as needed by the program

SIX WAYS TO SAXPY

Programming Languages for GPU Computing

Single Precision Alpha X Plus Y (SAXPY)

Part of Basic Linear Algebra Subroutines (BLAS) library

$$z = \alpha x + y$$

x, y, z : vector

α : scalar

GPU SAXPY in multiple languages and libraries

A selection of possibilities, not a tutorial

1

OpenACC Compiler Directives

Serial C code

```
void saxpy(int n,
           float a,
           float *x,
           float *y)
{

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

// Perform SAXPY on 1M elements
int N = 1<<20;
saxpy(N, 2.0, x, y);
```

Parallel C code with OpenACC

```
void saxpy(int n,
           float a,
           float *x,
           float *y)
{
#pragma acc kernels
  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}

// Perform SAXPY on 1M elements
int N = 1<<20;
saxpy(N, 2.0, x, y);
```

2

cuBLAS Library

Serial BLAS code

```
int N = 1<<20;

...

// Use your choice of blas library

// Perform SAXPY on 1M elements
blas_saxpy(N, 2.0, x, 1, y, 1);
```

Parallel cuBLAS code

```
int N = 1<<20;

cublasInit();
cublasSetVector(N, sizeof(x[0]), x, 1, d_x, 1);
cublasSetVector(N, sizeof(y[0]), y, 1, d_y, 1);

// Perform SAXPY on 1M elements
cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);

cublasGetVector(N, sizeof(y[0]), d_y, 1, y, 1);

cublasShutdown();
```

You can also call cuBLAS from Fortran, C++, Python and other languages

3

CUDA C++

Serial C code

```
void saxpy(int n, float a,
           float *x, float *y)
{
  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}


// Perform SAXPY on 1M elements
int N = 1<<20;
saxpy(N, 2.0, x, y);
```

CUDA C++ code

```
__global__
void saxpy(int n, float a,
           float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}


// Perform SAXPY on 1M elements
int N = 1<<20;
saxpy<<< 4096, 256 >>>(N, 2.0, d_x, d_y);
```

4

CUDA C++ Core Libraries (CCCL)

Serial Standard C++ code

```
int N = 1<<20;
std::vector<float> x(N), y(N);


...

// Perform SAXPY on 1M elements
std::transform(x.begin(), x.end(),
               y.begin(), y.end(),
               2.0f * _1 + _2);
```

CUDA C++ code

```
int N = 1<<20;
thrust::host_vector<float> x(N), y(N);
thrust::device_vector<float> d_x = x;
thrust::device_vector<float> d_y = y;
...

// Perform SAXPY on 1M elements
thrust::transform(d_x.begin(), d_x.end(),
                  d_y.begin(),d_y.begin(),
                  2.0f * _1 + _2)
```

5

Standard C++ Parallel Algorithms (stdpar)

Serial Standard C++ code

```
int N = 1<<20;
std::vector<float> x(N), y(N);


...

// Perform SAXPY on 1M elements
std::transform(x.begin(), x.end(),
               y.begin(), y.end(),
               2.0f * _1 + _2);
```

CUDA C++ code

```
int N = 1<<20;
std::vector<float> x(N), y(N), out;
out.reserve(N);

...

// Perform SAXPY on 1M elements
std::transform(
         std::execution::par_unseq,
         x.begin(), x.end(), y.begin(), y.end(),
         std::back_inserter(out),
         [](int a, int b) {
                  return 2.0f * a + b;
         });
```

6

Python

Standard Python

```
import numpy as np


def saxpy(a, x, y):
  return [a * xi + yi
          for xi, yi in zip(x, y)]

x = np.arange(2**20, dtype=np.float32)
y = np.arange(2**20, dtype=np.float32)







cpu_result = saxpy(2.0, x, y)
```

Numba Parallel Python

```
import numpy as np
from numba import vectorize

@vectorize(['float32(float32, float32,
             float32)'], target='cuda')
def saxpy(a, x, y):
  return a * x + y

N = 1048576

# Initialize arrays
A = np.ones(N, dtype=np.float32)
B = np.ones(A.shape, dtype=A.dtype)
C = np.empty_like(A, dtype=A.dtype)

# Add arrays on GPU
C = saxpy(2.0, X, Y)
```

Anatomy of a CUDA binary

Hello world example

```
__global__ void kernel() {
    printf("Hello, CUDA\n");
}

void main() {
    kernel<<< 1, 1 >>>();
    cudaDeviceSynchronize();
}
```

```
nvcc example.cu -o example
```

"example" executable

Anatomy of a CUDA binary

Hello world example

```
__global__ void kernel() {
    printf("Hello, CUDA\n");
}

void main() {
    kernel<<< 1, 1 >>>();
    cudaDeviceSynchronize();
}
```

```
nvcc example.cu -o example
```

main() — CPU code section

kernel() — GPU code section

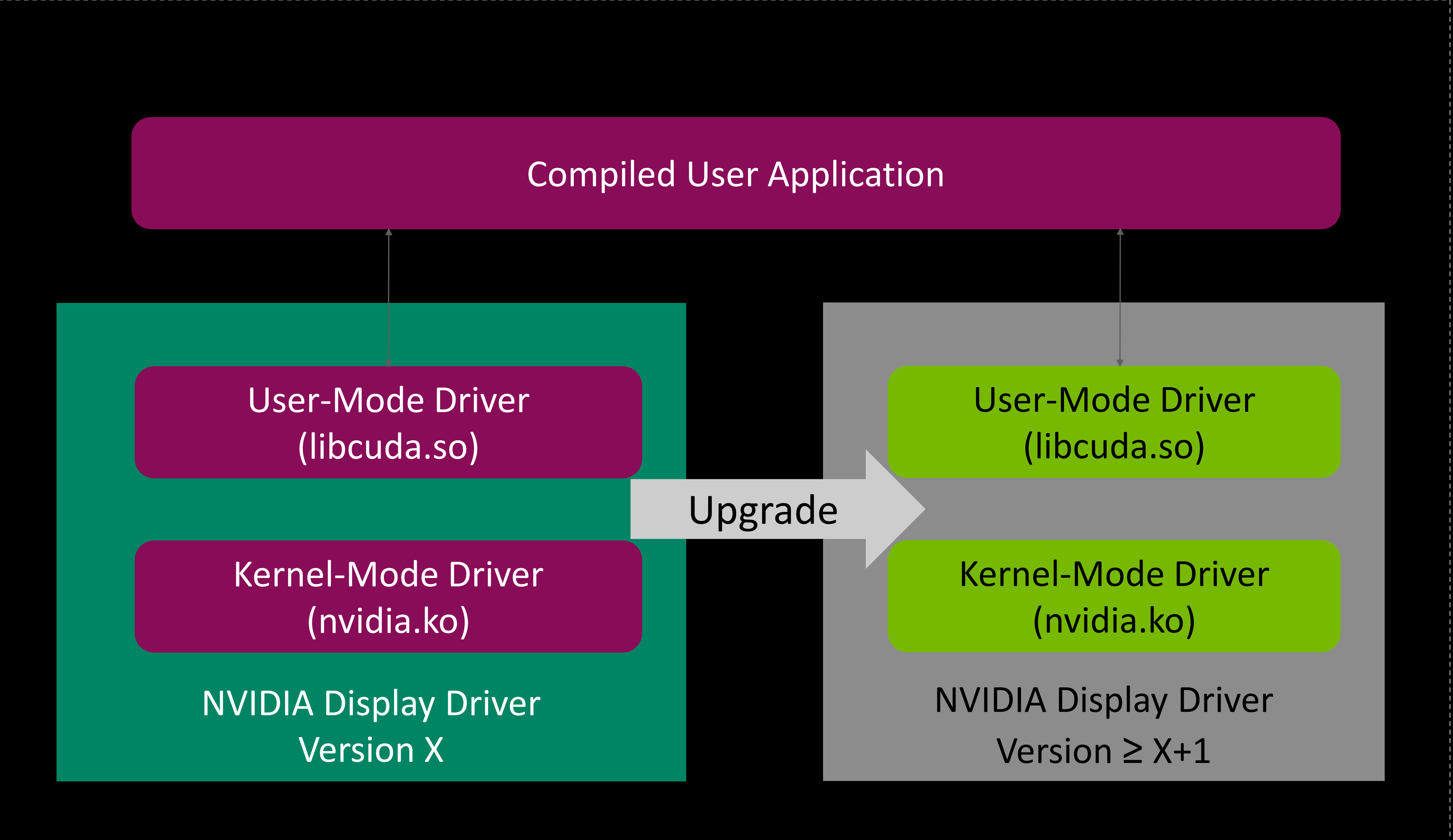
example.ptx — PTX Compatibility Layer

How Do We Keep Things Working
Together?

CUDA Compatibility

“Backward Compatibility” Software Considerations

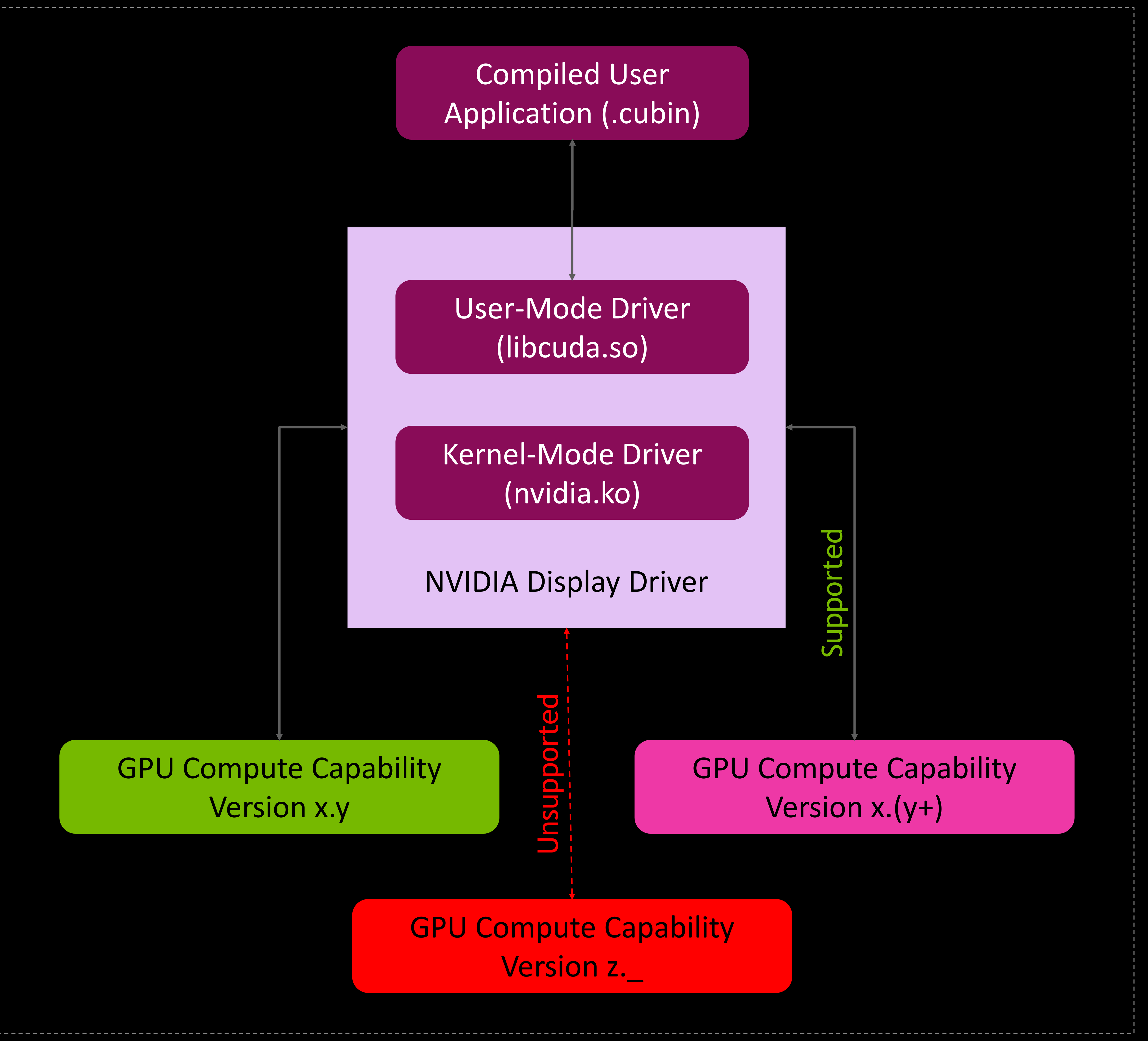
- The simplest use case:
  - “Your compiled application will work forever on NVIDIA GPUs, regardless of installed driver”
- All newer GPU drivers will be binary-compatible with older binaries
  - Requires statically linking libraries like the CUDA runtime
- Recompiling from *source* may require API changes
  - Only binary compatibility is guaranteed



CUDA Compatibility

“SM/Compute Compatibility” Hardware Considerations (Binaries)

- Binaries are built for a specific GPU family, PTX is used to target additional families
  - Each architecture supports a given ISA, or **compute capability**
  - PTX enables compatibility between architectures
- Compiled applications target a specific CC, with *some* compatibility within a family (newer but not older)
- Supported:
  - CC 8.0 cubin runs on CC 8.6 (A100 → A40)
- Unsupported:
  - CC 8.6 cubin cannot run on CC 8.0 (A40 → A100)
  - CC 8.0 cubin cannot run on CC 7.0 (A100 → V100)
  - CC 7.0 cubin cannot run on CC 8.0 (V100 → A100)

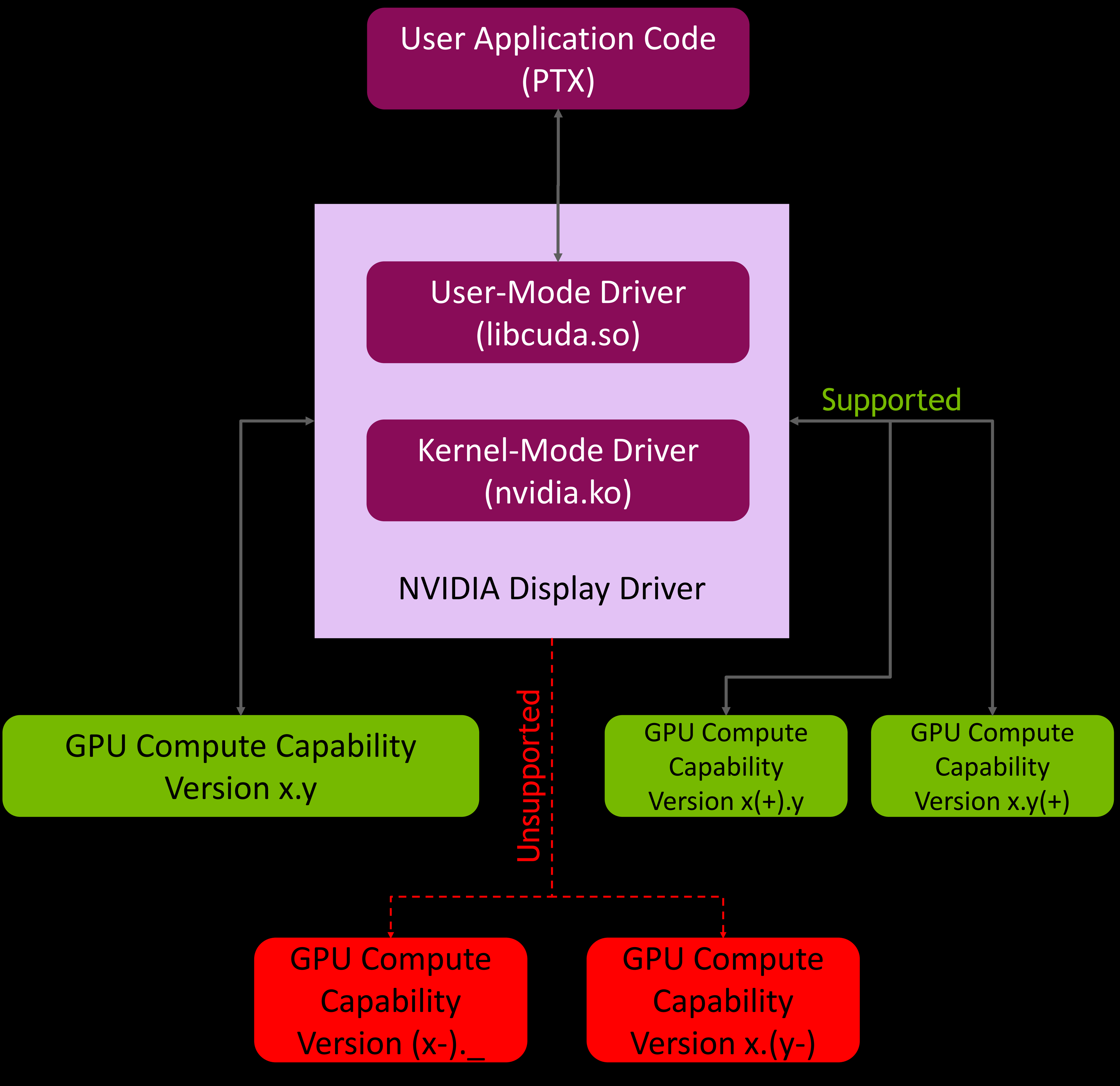


https://docs.nvidia.com/cuda/pdf/NVIDIA_Ampere_GPU_Architecture_Compatibility_Guide.pdf

CUDA compatibility

“PTX Compatibility” Hardware Considerations

- **PTX Code** is compatible with future versions, both Major and Minor
- Supported PTX Migration:
  - CC 8.0 PTX runs on CC 8.6 (A100 PTX → A40)
  - CC 7.0 PTX runs on CC 8._ (V100 PTX → A100)
- Unsupported PTX Migration:
  - CC 8.6 PTX cannot run on CC 8.0 (A40 PTX → A100)
  - CC 8.0 PTX cannot run on CC 7.0 (A100 PTX → V100)



https://docs.nvidia.com/cuda/pdf/NVIDIA_Ampere_GPU_Architecture_Compatibility_Guide.pdf

CUDA COMPATIBILITY

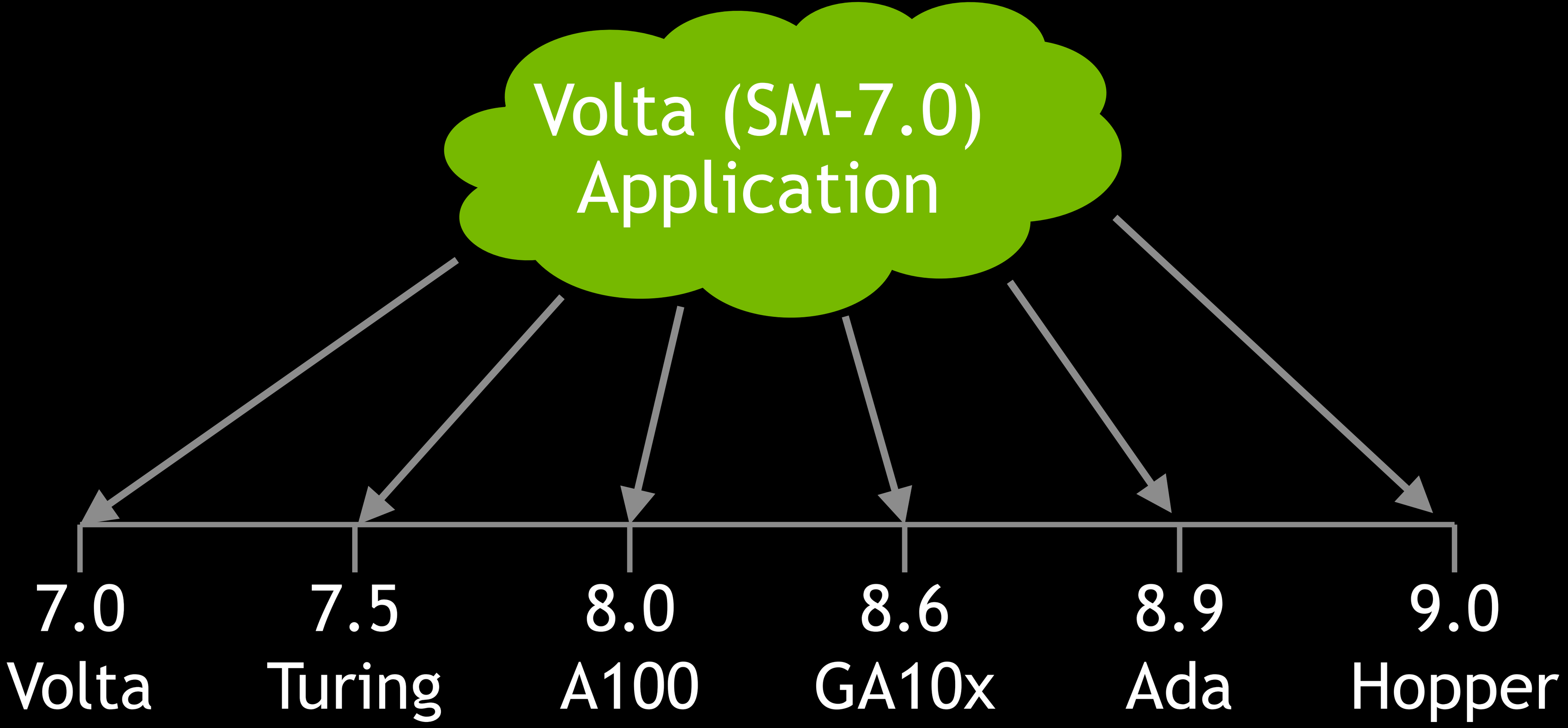
"PTX Compatibility" Hardware Considerations

"CUDA Everywhere"

Code for one GPU runs on all GPUs with newer SM version

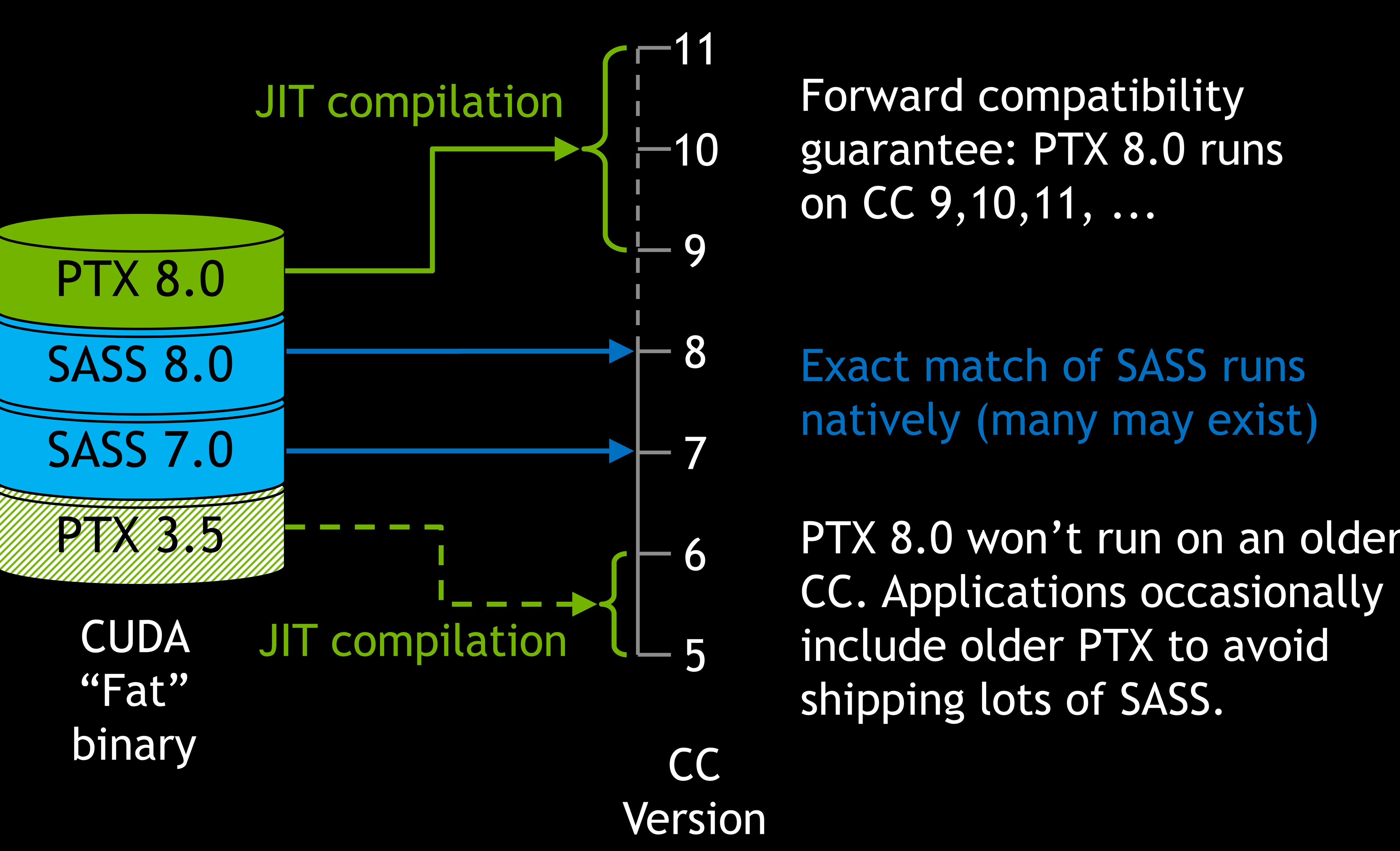
CUDA applications are portable between all chip classes (100, 10x, 20x, 21x, 10b, etc.)

- All current features supported on all future architectures
- Performance & capacities vary (e.g. SM count)
- A few features much slower but still functional (e.g. FP64)



Volta applications "just work" on Turing/Ampere/Hopper
Datacenter libraries "just work" for Quadro, GeForce, etc.

Portability depends on PTX Just-In-Time Compilation



Forward compatibility guarantee: PTX 8.0 runs on CC 9,10,11, ...

Exact match of SASS runs natively (many may exist)

PTX 8.0 won't run on an older CC. Applications occasionally include older PTX to avoid shipping lots of SASS.

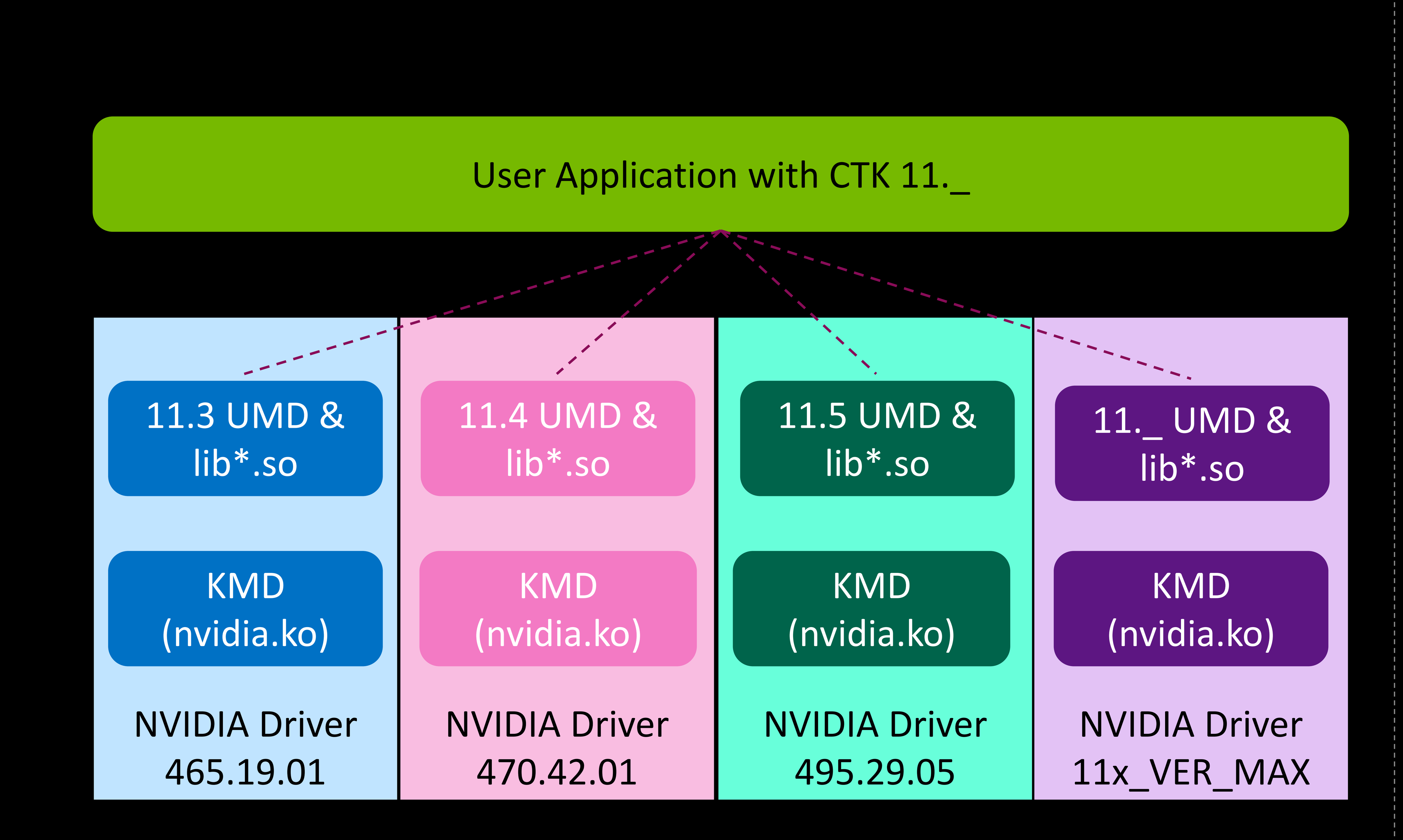
SASS is pre-compiled binary code native to a specific GPU architecture - multiple versions may be packaged together

PTX is assembly code JIT compiled by CUDA when an application is run on a new GPU for which there is no SASS

CUDA compatibility

"Minor-Version Compatibility" (Previously "Enhanced Compatibility")

- Applications created within a major-release of CUDA may run on a system with the minimum driver version
  - E.g., 11.x CTK requires **450.80.02**
- Works with:
  - Newer driver than CTK
  - Newer CTK than driver
- New CTK features that require a new driver will return errors
  - Programmers must write code to check if features exist and if libraries are supported (e.g., cublas must match cudnn)*
- PTX JIT unsupported (matching driver required)

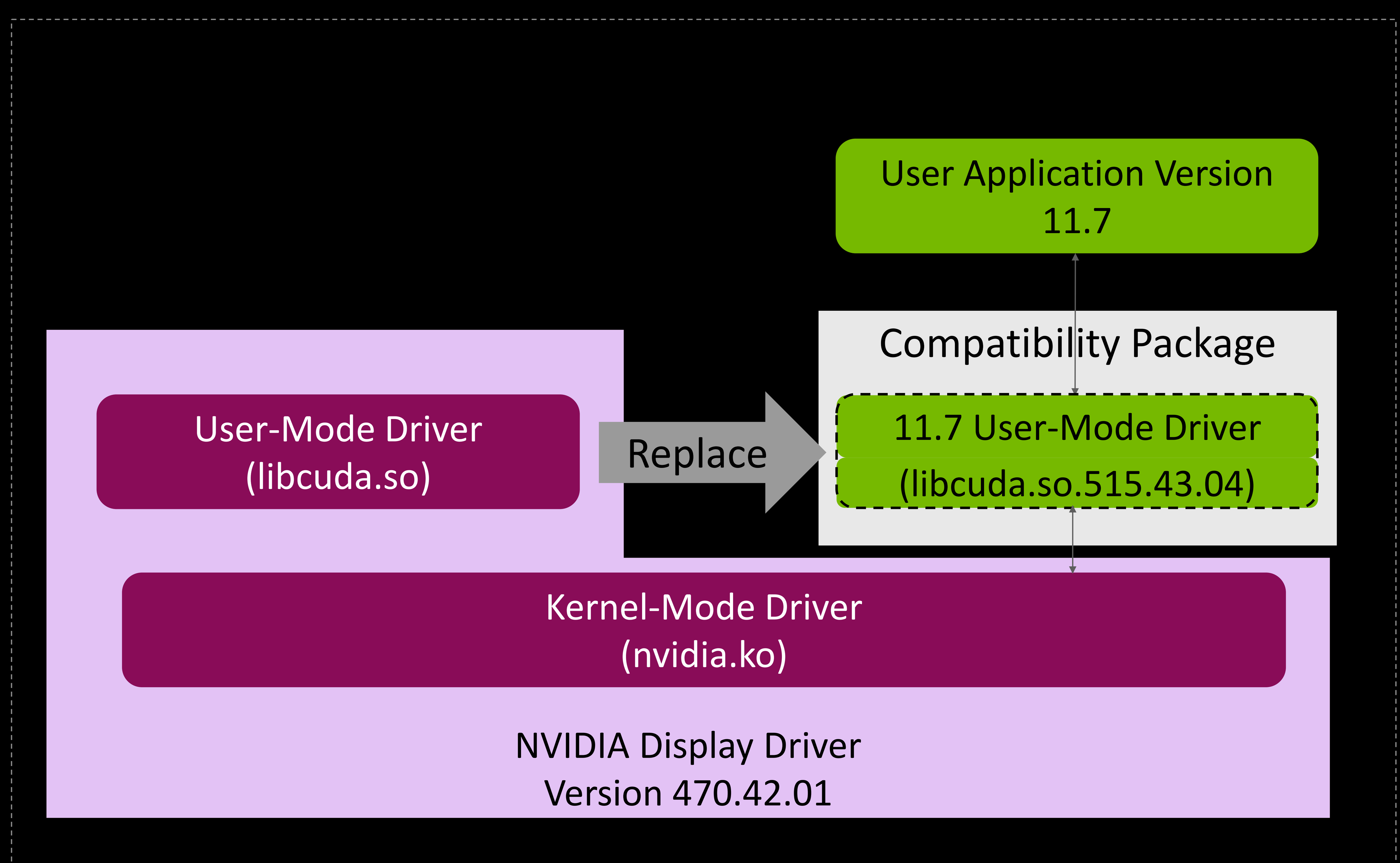


* See NVIDIA "Best Practices" Documentation

CUDA Compatibility

"Forward Compatibility"

- Using a CUDA toolkit with higher-versioned UMD with a lower-versioned KMD
- Deployment & upgrade of Drivers may be very disruptive, especially in CSP and enterprise datacenters
- Can be used across major and minor versions of CTK
- Compatibility Package to be installed, includes user-mode driver (among other files)
  - Via symbolic links, multiple compatibility versions can be installed together in a single system
- Programmers must check for supported features & supported hardware
- Supports PTX JIT compilation



Key Takeaways

- CUDA applications are compatible – forever
- CUDA programs within a major version – *generally* are compatible
- CUDA applications run against older drivers – with compatibility shims
  - Matters in e.g. containers, data center environments

Multi GPU Multi Node programming

Example: Jacobi solver

Solves the 2D-Laplace Equation on a rectangle

$$\Delta \boldsymbol{u}(\boldsymbol{x},\boldsymbol{y})=\boldsymbol{0}\ \forall\ (\boldsymbol{x},\boldsymbol{y})\in\Omega\backslash\boldsymbol{\delta}\Omega$$

Dirichlet boundary conditions (constant values on boundaries) on left and right boundary

Periodic boundary conditions on top and bottom boundary

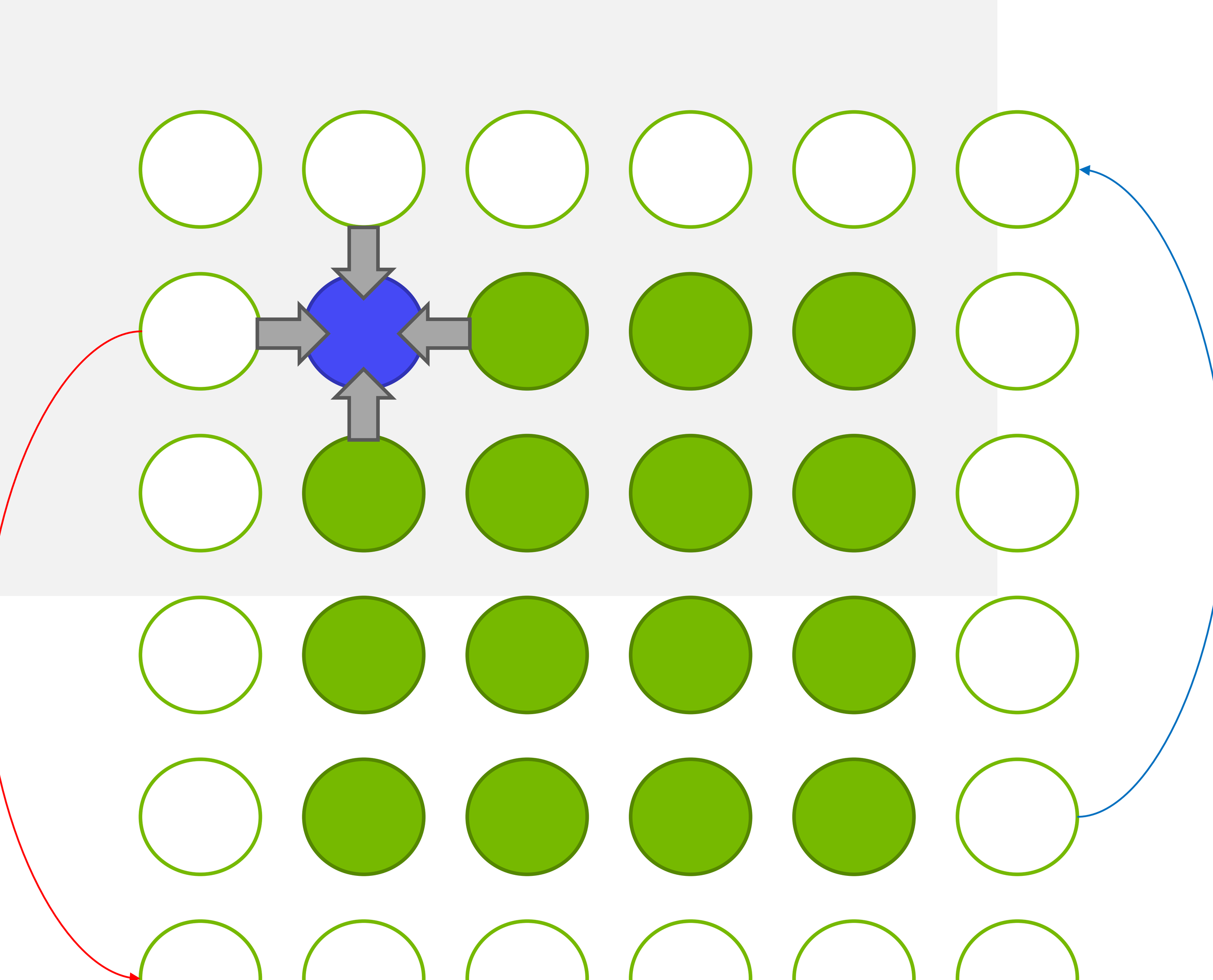
Example: Jacobi Solver

Single GPU

While not converged

Do Jacobi step:

```
for( int iy = 1        ; iy < ny-1  ; iy++ )
for( int ix = 1        ; ix < nx-1  ; ix++ )
    a_new[iy*nx+ix] = -0.25 *
                  -( a[ iy   *nx+(ix+1)] + a[ iy   *nx+ix-1]
                   + a[(iy-1)*nx+ ix   ] + a[(iy+1)*nx+ix  ] );
```



Apply periodic boundary conditions

Swap a_new and a

Next iteration

Example: Jacobi Solver

Multi GPU

While not converged

Do Jacobi step:

```
for( int iy = iy_start; iy < iy_end; iy++ )
for( int ix = 1        ; ix < nx-1  ; ix++ )
    a_new[iy*nx+ix] = -0.25 *
                    -( a[ iy   *nx+(ix+1)] + a[ iy   *nx+ix-1]
                     + a[(iy-1)*nx+ ix   ] + a[(iy+1)*nx+ix  ] );
```

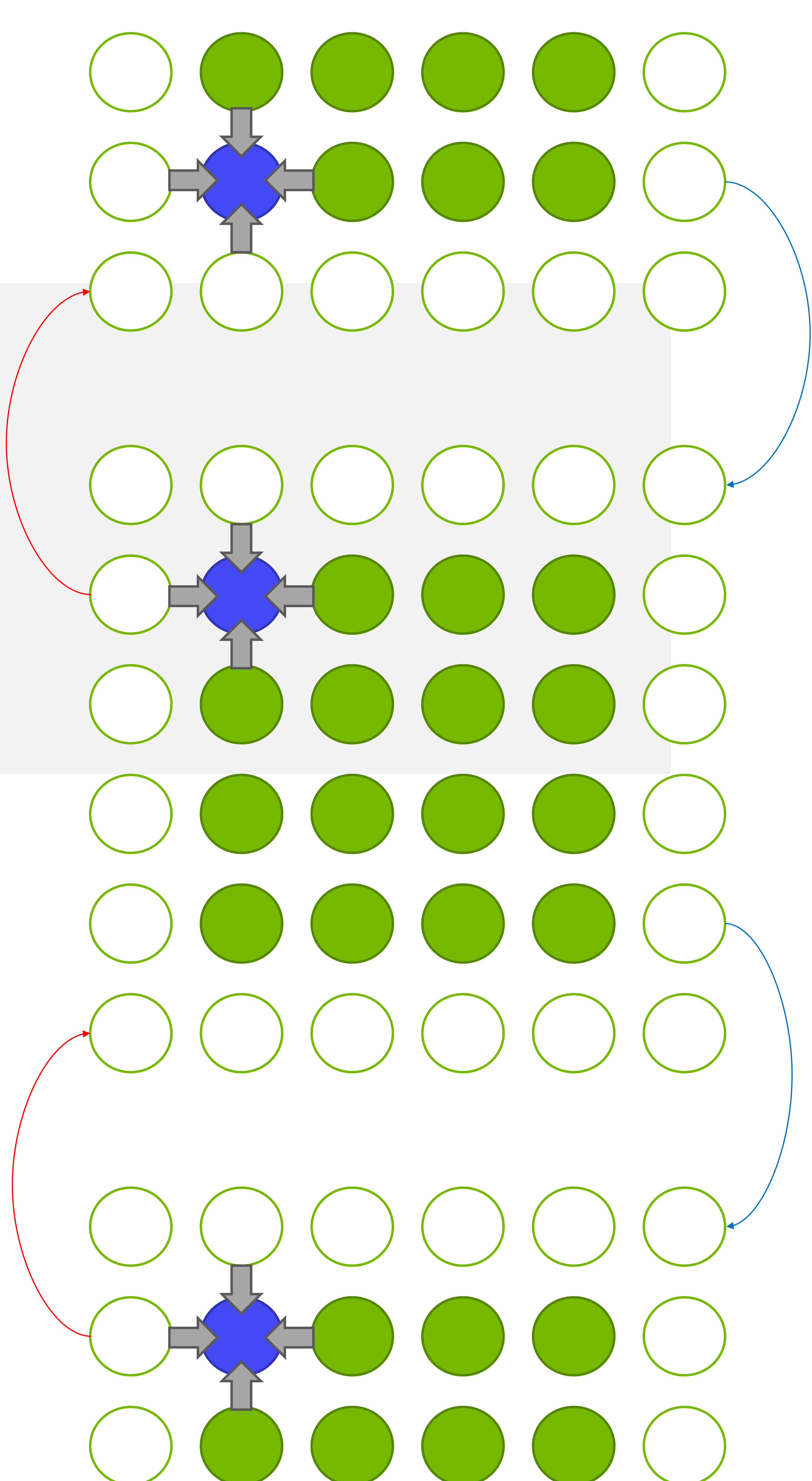
Apply periodic boundary conditions

Halo exchange

One-step with ring exchange

Swap a_new and a

Next iteration



Message Passing Interface - MPI

- Standard to exchange data between processes via messages
  - Defines API to exchanges messages
    - Point to Point: e.g. `MPI_Send`, `MPI_Recv`
    - Collectives: e.g. `MPI_Reduce`
- Multiple implementations (open source and commercial)
  - Bindings for C/C++, Fortran, Python, …
  - E.g. MPICH, OpenMPI, MVAPICH, IBM Platform MPI, Cray MPT, …

MPI - Skeleton

```
#include <mpi.h>
int main(int argc, char *argv[]) {
    int rank,size;
    /* Initialize the MPI library */
    MPI_Init(&argc,&argv);
    /* Determine the calling process rank and total number of ranks */
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    /* Call MPI routines like MPI_Send, MPI_Recv, ... */
    ...
    /* Shutdown MPI library */
    MPI_Finalize();
    return 0;
}
```

MPI

Launching

```
$ mpirun -np 4 ./myapp <args>
```

rank = 0 | rank = 1 | rank = 2 | rank = 3

myapp | myapp | myapp | myapp

MEM MEM GPU CPU PCIe Switch IB (×4)

Multi Process Multi GPU Programming

Using CUDA-aware MPI

Handle GPU affinity on multi-GPU nodes:

```
int local_rank = -1;
MPI_Comm_rank(local_comm,&local_rank);
int num_devices = 0;
cudaGetDeviceCount(&num_devices);
cudaSetDevice(local_rank % num_devices);
```

(Use `MPI_Comm_split_type(MPI_COMM_WORLD, MPI_COMM_TYPE_SHARED, rank, info, &local_comm);` to get `local_comm`.)

Multi Process Multi GPU Programming

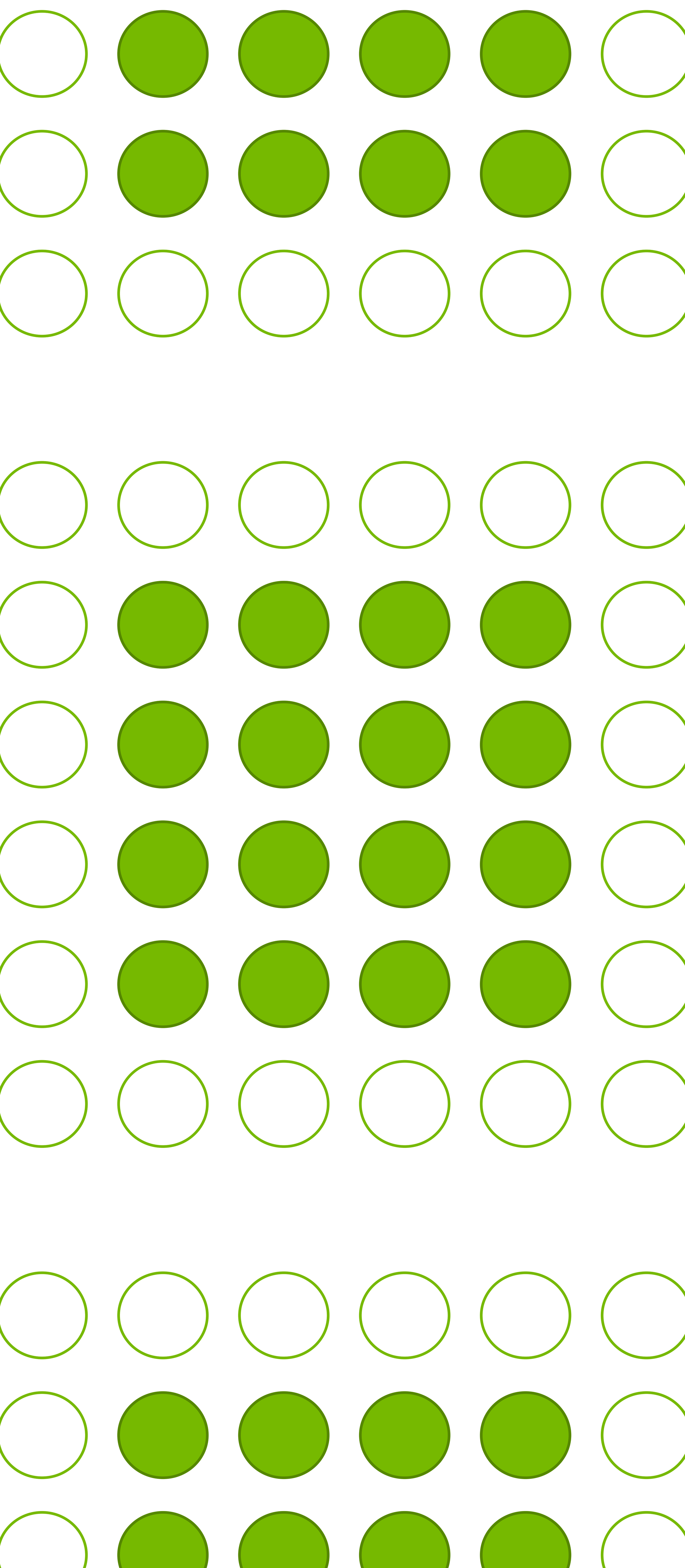
Using CUDA-aware MPI

```
while (l2_norm > tol && iter < iter_max) {
    cudaMemsetAsync(l2_norm_d, 0, sizeof(real), compute_stream);
    launch_jacobi_kernel(a_new, a, l2_norm_d, iy_start, iy_end, nx, compute_stream);
    cudaEventRecord(compute_done, compute_stream);
    cudaMemcpyAsync(l2_norm_h, l2_norm_d, sizeof(real), cudaMemcpyDeviceToHost, compute_stream);

    cudaEventSynchronize(compute_done);
    const int top = rank > 0 ? rank - 1 : (size - 1);
    const int bottom = (rank + 1) % size;
    // Top/Bottom Halo exchange -> next slide


    cudaStreamSynchronize(compute_stream);
    MPI_CALL(MPI_Allreduce(l2_norm_h, &l2_norm, 1, MPI_REAL_TYPE, MPI_SUM, MPI_COMM_WORLD));
    l2_norm = std::sqrt(l2_norm);

    std::swap(a_new, a); iter++;
}
```

Example Jacobi

Top/Bottom Halo

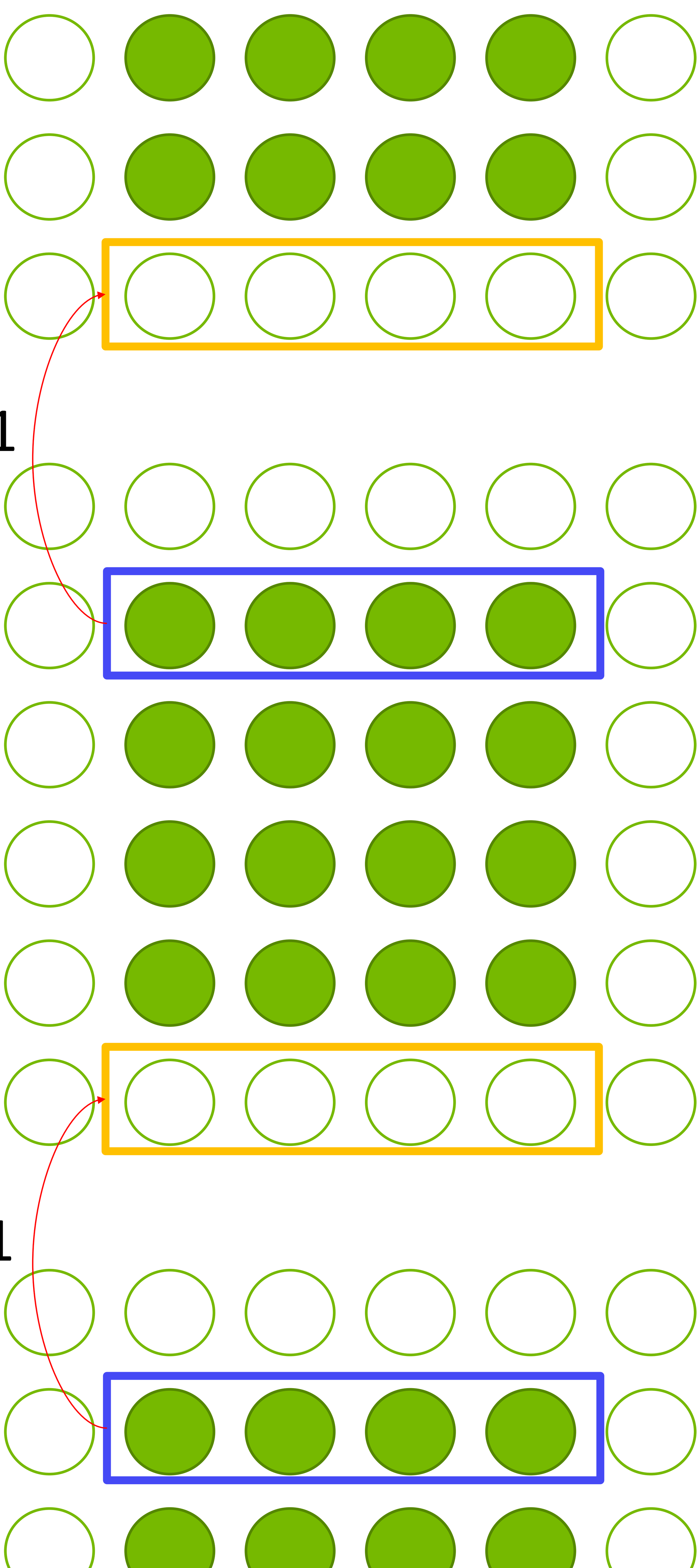
```
MPI_Sendrecv(a_new+iy_start*nx, nx, MPI_FLOAT, top , 0,
             a_new+(iy_end*nx), nx, MPI_FLOAT, bottom, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```



Example Jacobi

Top/Bottom Halo

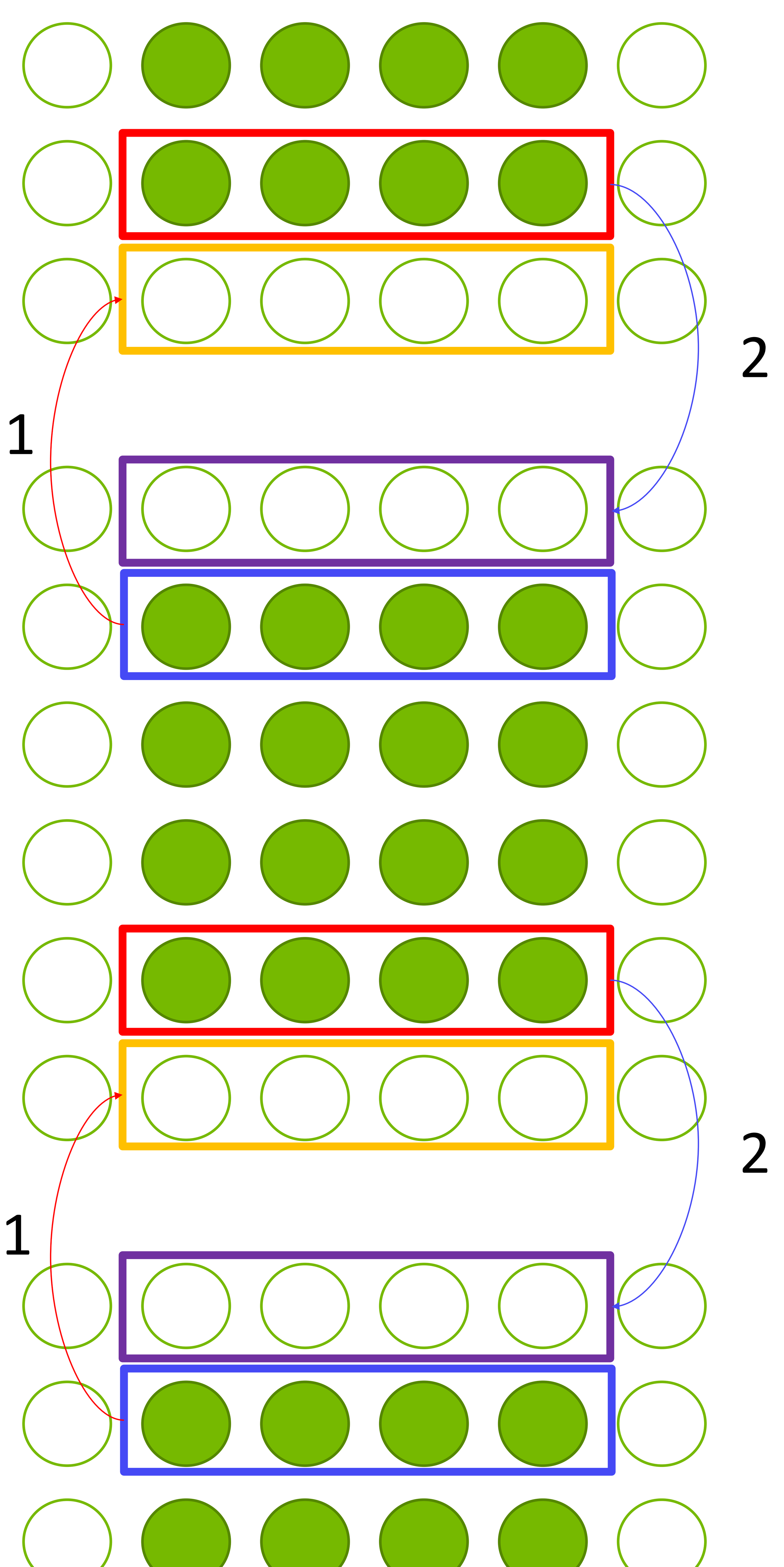
```
MPI_Sendrecv(a_new+iy_start*nx, nx, MPI_FLOAT, top , 0,
             a_new+(iy_end*nx), nx, MPI_FLOAT, bottom, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```



Example Jacobi

Top/Bottom Halo

```
MPI_Sendrecv(a_new+iy_start*nx, nx, MPI_FLOAT, top , 0,
             a_new+(iy_end*nx), nx, MPI_FLOAT, bottom, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);


MPI_Sendrecv(a_new+(iy_end-1)*nx, nx, MPI_FLOAT, bottom, 0,
             a_new              , nx, MPI_FLOAT, top, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```



Benchmark Setup

DGX H100

- CUDA Driver 535.129.03
- NVIDIA HPC SDK container: Tag `nvcr.io/nvidia/nvhpc:24.1-devel-cuda12.3-ubuntu22.04`
- GPUs@1980 Mhz
- Reported Runtime is the minimum of 5 repetitions
- For all runs CPU and GPU affinities have been tuned: See `bench.sh` in https://github.com/NVIDIA/multi-gpu-programming-models

Example: Jacobi Solver

Single GPU performance vs. problem size – NVIDIA H100 80GB HBM3

Problem size nx=ny	1024	2048	3072	4096	5120	6144	7168	8192	9216	10240	11264	12288	13312	14336	15360	16384	17408	18432	19456	20480

Performane [Mcells/s]: 0, 20000, 40000, 60000, 80000, 100000, 120000, 140000, 160000, 180000, 200000

Efficiency [%]: 0,00%, 10,00%, 20,00%, 30,00%, 40,00%, 50,00%, 60,00%, 70,00%, 80,00%, 90,00%, 100,00%

Performance [Mcells/s] — Efficiency [%]

Benchmarksetup: DGX H100, CUDA Driver 535.129.03, NVIDIA HPC SDK container: `nvcr.io/nvidia/nvhpc:24.1-devel-cuda12.3-ubuntu22.04`, GPUs@1980Mhz AC, Reported Runtime is the minimum of 5 repetitions

Multi GPU Jacobi Runtime And Parallel Efficiency

MPI on DGX H100 – 20480 x 20480, 1000 iterations

Benchmark setup: DGX H100, CUDA Driver 535.129.03, NVIDIA HPC SDK container: `nvcr.io/nvidia/nvhpc:24.1-devel-cuda12.3-ubuntu22.04`, GPUs@1980Mhz AC, Reported Runtime is the minimum of 5 repetitions

Multi GPU Jacobi Nsight Systems Timeline

MPI 8 NVIDIA H100 80GB HBM3 on DGX H100

Overlapping Communication and Computation

No Overlap

Process Whole Domain

COMM

Overlap

Process inner domain

Possible gain

Process boundary domain

Dependency

COMM

MPI

Overlapping Communication and Computation

```
launch_jacobi_kernel( a_new, a, l2_norm_d, iy_start, (iy_start+1), nx, push_top_stream );
launch_jacobi_kernel( a_new, a, l2_norm_d, (iy_end-1), iy_end, nx, push_bottom_stream );
launch_jacobi_kernel( a_new, a, l2_norm_d, (iy_start+1), (iy_end-1), nx, compute_stream );
const int top = rank > 0 ? rank - 1 : (size-1);
const int bottom = (rank+1)%size;
cudaStreamSynchronize( push_top_stream );
MPI_Sendrecv( a_new+iy_start*nx, nx, MPI_REAL_TYPE, top , 0,
              a_new+(iy_end*nx), nx, MPI_REAL_TYPE, bottom, 0,
              MPI_COMM_WORLD, MPI_STATUS_IGNORE );
cudaStreamSynchronize( push_bottom_stream );
MPI_Sendrecv( a_new+(iy_end-1)*nx, nx, MPI_REAL_TYPE, bottom, 0,
              a_new, nx, MPI_REAL_TYPE, top, 0, MPI_COMM_WORLD,
              MPI_STATUS_IGNORE );
```

Multi GPU Jacobi Nsight Systems Timeline

MPI Overlap 8 NVIDIA H100 80GB HBM3 on DGX H100

NVIDIA Nsight Systems 2024.1.1

File View Tools Help

Untitled 2 *

Timeline View | Options... | 1x | 112 messages

+672.12ms +672.14ms +672.16ms +672.18ms +672.2ms +672.22ms +672.24ms +672.26ms +672.28ms +672.3ms +672.32ms +672.34ms +672.36ms +672.38ms +672.4ms

[All Streams] — void jacobi_kernel<(int)32, (int)32>(float *, const float *, float *, int, int, int, bool)

MPI — MPI_Sendrecv [25.766 μs] MPI_Sendrecv [26.797 μs]

MPI — MPI_Sendrecv [24.695 μ... MPI_Sendrecv [25.656 μs]

MPI — MPI_Sendrecv [26.419 μs] MPI_Sendrecv [24.422 ...

MPI — MPI_Sendrecv [27.843 μs] MPI_Sendrecv [23.214...

MPI — MPI_Sendrecv [25.898 μs] MPI_Sendrecv [23.589 ...

MPI — MPI_Sendrecv [26.572 μs] MPI_Sendrecv [26.030 μs]

MPI — MPI_Sendrecv [27.258 μs] MPI_Sendrecv [27.260 μs]

MPI — MPI_Sendrecv [27.678 μs] MPI_Sendrecv [22.72...

NVTX — Jacobi solve [2.575 ms] — MPI [52.560 μs] — MPI [57.88...

CUDA API — cudaStreamSynchronize

Multi GPU Jacobi Parallel Efficiency

DGX H100 – 20480 x 20480, 1000 iterations

Parallel Efficiency [%]

100,00%
98,00%
96,00%
94,00%
92,00%
90,00%
88,00%
86,00%
84,00%
82,00%
80,00%

1 2 3 4 5 6 7 8

#GPUs

mpi mpi_overlap

Benchmarksetup: DGX H100, CUDA Driver 535.129.03, NVIDIA HPC SDK container: `nvcr.io/nvidia/nvhpc:24.1-devel-cuda12.3-ubuntu22.04`, GPUs@1980Mhz AC, Reported Runtime is the minimum of 5 repetitions

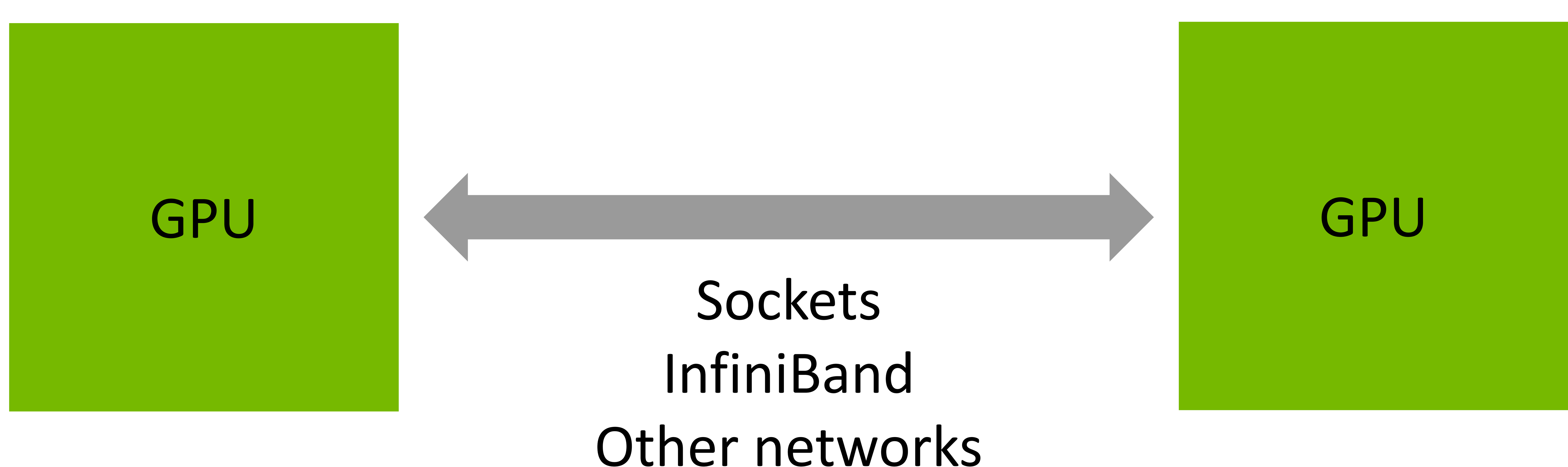
NCCL

Optimized inter-GPU communication

NCCL : NVIDIA Collective Communication Library

Communication library running on GPUs, for GPU buffers.

- Library for efficient communication with GPUs
- First: Collective Operations (e.g. Allreduce), as they are required for Deep Learning
- Since 2.8: Support for Send/Recv between GPUs
- Library running on GPU: Communication calls are translated to a GPU kernel (running on a stream)



Binaries : https://developer.nvidia.com/nccl and in NGC containers
Source code : https://github.com/nvidia/nccl
Perf tests : https://github.com/nvidia/nccl-tests

Multi GPU Jacobi Parallel Efficiency

DGX H100 – 20480 x 20480, 1000 iterations

Parallel Efficiency [%]

100,00%
98,00%
96,00%
94,00%
92,00%
90,00%
88,00%
86,00%
84,00%
82,00%
80,00%

1 2 3 4 5 6 7 8

#GPUs

mpi mpi_overlap nccl

Benchmarksetup: DGX H100, CUDA Driver 535.129.03, NVIDIA HPC SDK container: `nvcr.io/nvidia/nvhpc:24.1-devel-cuda12.3-ubuntu22.04`, GPUs@1980Mhz AC, Reported Runtime is the minimum of 5 repetitions

NCCL

Overlapping Communication and Computation

```
int leastPriority = 0;
int greatestPriority = leastPriority;
cudaDeviceGetStreamPriorityRange(&leastPriority, &greatestPriority);
cudaStream_t compute_stream,push_stream;
cudaStreamCreateWithPriority(&compute_stream, cudaStreamDefault, leastPriority);
cudaStreamCreateWithPriority(&push_stream, cudaStreamDefault, greatestPriority);

...

launch_jacobi_kernel(a_new, a, l2_norm_d, iy_start,       (iy_start + 1), nx, push_stream);
launch_jacobi_kernel(a_new, a, l2_norm_d, (iy_end - 1),   iy_end,         nx, push_stream);
launch_jacobi_kernel(a_new, a, l2_norm_d, (iy_start + 1), (iy_end - 1),   nx, compute_stream);
ncclGroupStart();
ncclRecv(a_new,                     nx, NCCL_REAL_TYPE, top, nccl_comm, push_stream)
ncclSend(a_new + (iy_end - 1) * nx, nx, NCCL_REAL_TYPE, btm, nccl_comm, push_stream);
ncclRecv(a_new + (iy_end * nx),     nx, NCCL_REAL_TYPE, btm, nccl_comm, push_stream);
ncclSend(a_new + iy_start * nx,     nx, NCCL_REAL_TYPE, top, nccl_comm, push_stream);
ncclGroupEnd();
```

Need to use CUDA high priority streams to avoid NCCL comms getting stuck behind compute.

Multi GPU Jacobi Parallel Efficiency

DGX H100 – 20480 x 20480, 1000 iterations

Parallel Efficiency [%]

100,00%
98,00%
96,00%
94,00%
92,00%
90,00%
88,00%
86,00%
84,00%
82,00%
80,00%

1 2 3 4 5 6 7 8

#GPUs

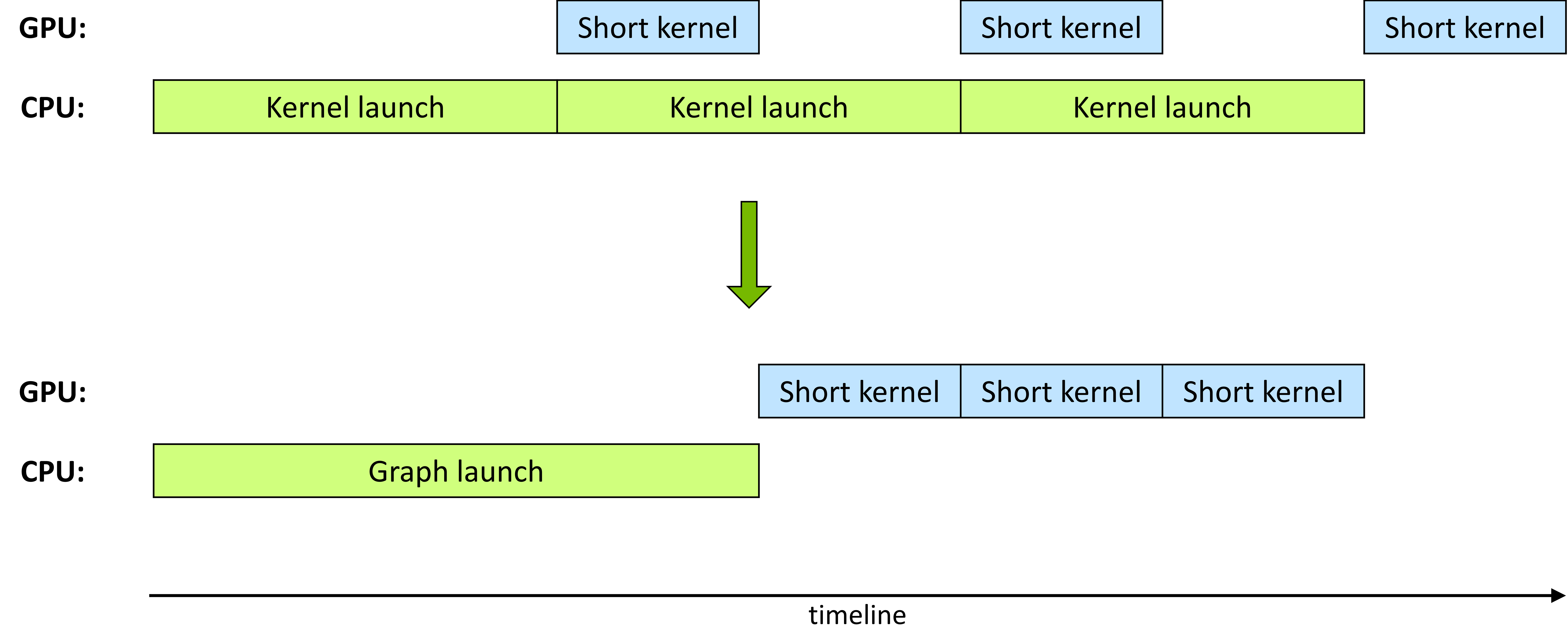
mpi mpi_overlap nccl nccl_overlap

Benchmarksetup: DGX H100, CUDA Driver 535.129.03, NVIDIA HPC SDK container: `nvcr.io/nvidia/nvhpc:24.1-devel-cuda12.3-ubuntu22.04`,
GPUs@1980Mhz AC, Reported Runtime is the minimum of 5 repetitions

CUDA Graphs

Reducing launch overhead

CUDA graphs reduce kernel launch latencies:



From Advanced Performance Optimization in CUDA [S62192] by Igor Terentyev more details on Graphs there.

Three-Stage Execution Model

Minimizes Execution Overheads – Pre-Initialize As Much As Possible

Define

Single Graph "Template"

Created in host code

or built up from libraries

Instantiate

Multiple "Executable Graphs"

Snapshot of templates

Sets up & initializes GPU execution structures
(create once, run many times)

Execute

Executable Graphs Running in CUDA Streams

Concurrency in graph **is not** limited by stream

Multi GPU Jacobi Parallel Efficiency

DGX H100 – 20480 x 20480, 1000 iterations

Parallel Efficiency [%]

100,00%
98,00%
96,00%
94,00%
92,00%
90,00%
88,00%
86,00%
84,00%
82,00%
80,00%

1 2 3 4 5 6 7 8

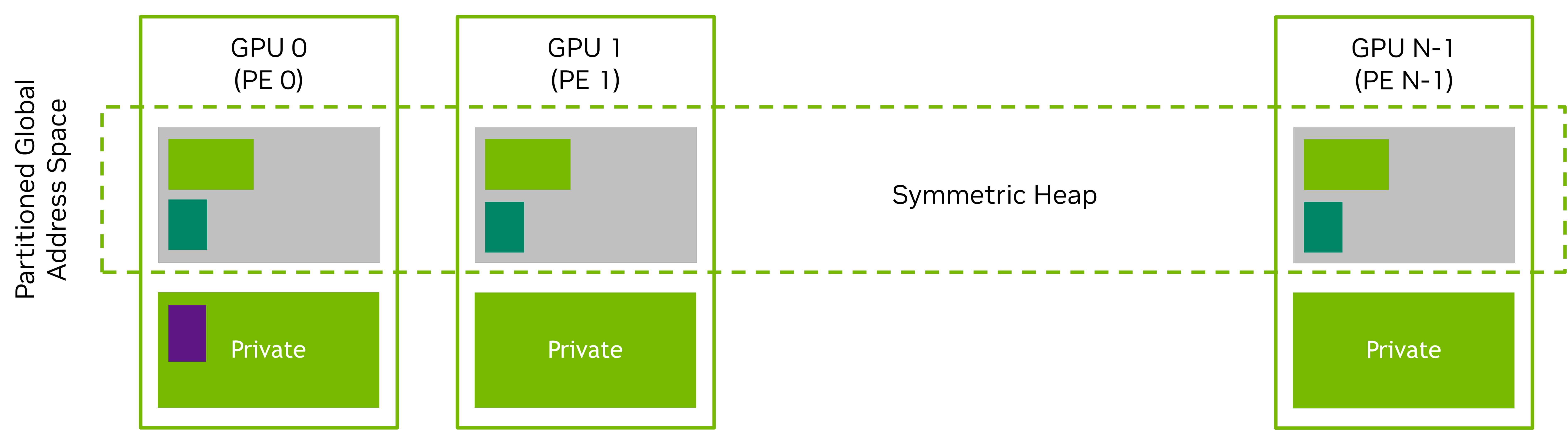
#GPUs

mpi, mpi_overlap, nccl, nccl_overlap, nccl_graphs

Benchmarksetup: DGX H100, CUDA Driver 535.129.03, NVIDIA HPC SDK container: `nvcr.io/nvidia/nvhpc:24.1-devel-cuda12.3-ubuntu22.04`, GPUs@1980Mhz AC, Reported Runtime is the minimum of 5 repetitions

NVSHMEM

Implementation of OpenSHMEM, a Partitioned Global Address Space (PGAS) library



Symmetric objects are allocated collectively with the same size on every PE

Symmetric memory: `nvshmem_malloc(…);` Private memory: `cudaMalloc(…);`

CPU (blocking and stream-ordered) and CUDA Kernel interfaces

Read: `nvshmem_get(…);` Write: `nvshmem_put(…);` Atomic: `nvshmem_atomic_add(…);`
Flush writes: `nvshmem_quiet();` Order writes: `nvshmem_fence();`

Synchronize: `nvshmem_barrier();` Poll: `nvshmem_wait_until(…);`

Interoperable with MPI

Multi GPU Jacobi Parallel Efficiency

DGX H100 – 20480 x 20480, 1000 iterations

Different baseline [1]

100,00%
98,00%
96,00%
94,00%
92,00%
90,00%
88,00%
86,00%
84,00%
82,00%
80,00%

Parallel Efficiency [%]

1 2 3 4 5 6 7 8

#GPUs

- mpi
- mpi_overlap
- nccl
- nccl_overlap
- nccl_graphs
- nvshmem
- nvshmem+lto
- nvshmem-neighborhood_sync+lto
- nvshmem-norm_overlap-neighborhood_sync+lto
- nvshmem-norm_overlap-neighborhood_sync+lto[1]

[1] Serial execution baseline uses same kernel as parallel version

Benchmarksetup: DGX H100, CUDA Driver 535.129.03, NVIDIA HPC SDK container: `nvcr.io/nvidia/nvhpc:24.1-devel-cuda12.3-ubuntu22.04`, GPUs@1980Mhz AC, Reported Runtime is the minimum of 5 repetitions

Conclusion

Thank you for your attention

	GPUDirect P2P/RDMA	CUDA stream/graph-aware	Kernel Initiated Communication
MPI	Improves Perf.	No	No
NCCL	Improves Perf.	Yes	No
NVSHMEM	Required	Yes	Yes

Source is on GitHub: https://github.com/NVIDIA/multi-gpu-programming-models

NVIDIA®