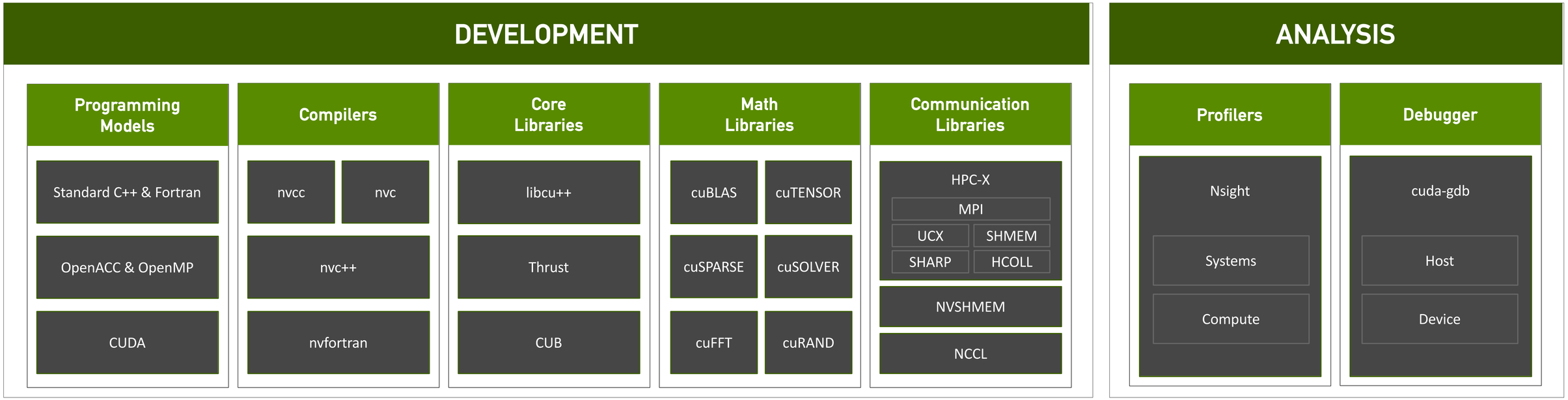
Programming the NVIDIA Platform

NVIDIA HPC SDK

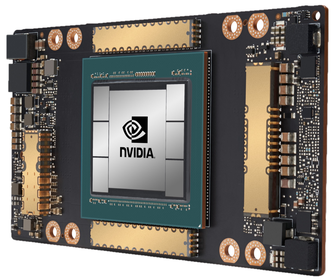


Develop for the NVIDIA Platform: GPU, CPU and Interconnect
Libraries | Accelerated C++ and Fortran | Directives | CUDA
x86_64 | Arm | OpenPOWER
7-8 Releases Per Year | Freely Available

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

HPC Compilers

NVC | NVC++ | NVFORTRAN



Accelerated

H100
Automatic

Fortran

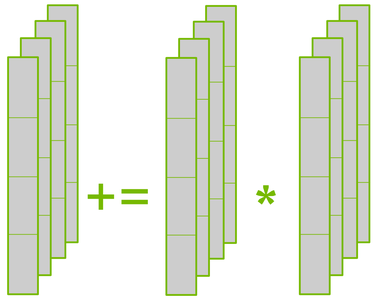
NVIDIA
CUDA

OpenACC
More Science, Less Programming

OpenMP

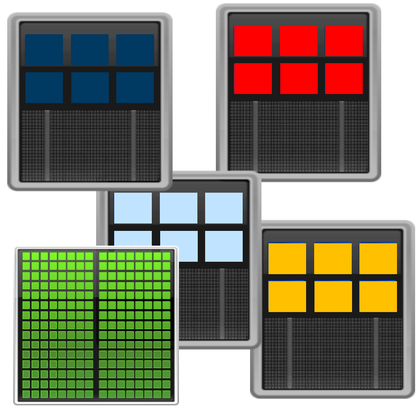
Programmable

Standard Languages
Directives
CUDA



CPU Optimized

Directives
Vectorization



Multi-Platform

x86_64
AArch64
OpenPOWER

Programming the NVIDIA platform

CPU, GPU, and Network

ACCELERATED STANDARD LANGUAGES

ISO C++, ISO Fortran

```
std::transform(par, x, x+n, y, y,
    [=](float x, float y){ return y +
a*x; }
);
```

```
do concurrent (i = 1:n)
   y(i) = y(i) + a*x(i)
enddo
```

```
import cunumeric as np
…
def saxpy(a, x, y):
    y[:] += a*x
```

INCREMENTAL PORTABLE OPTIMIZATION

OpenACC, OpenMP

```
#pragma acc data copy(x,y) {
...
std::transform(par, x, x+n, y, y,
    [=](float x, float y){
        return y + a*x;
});
...
}
```

```
#pragma omp target data map(x,y) {
...
std::transform(par, x, x+n, y, y,
    [=](float x, float y){
        return y + a*x;
});
...
}
```

PLATFORM SPECIALIZATION

CUDA

```
__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];
}

int main(void) {
  ...
  cudaMemcpy(d_x, x, ...);
  cudaMemcpy(d_y, y, ...);

  saxpy<<<(N+255)/256,256>>>(...);

  cudaMemcpy(y, d_y, ...);
```

ACCELERATION LIBRARIES

Core	Math	Communication	Data Analytics	AI	Quantum

Accelerated Standard Languages

Parallel performance for wherever your code runs

ISO C++

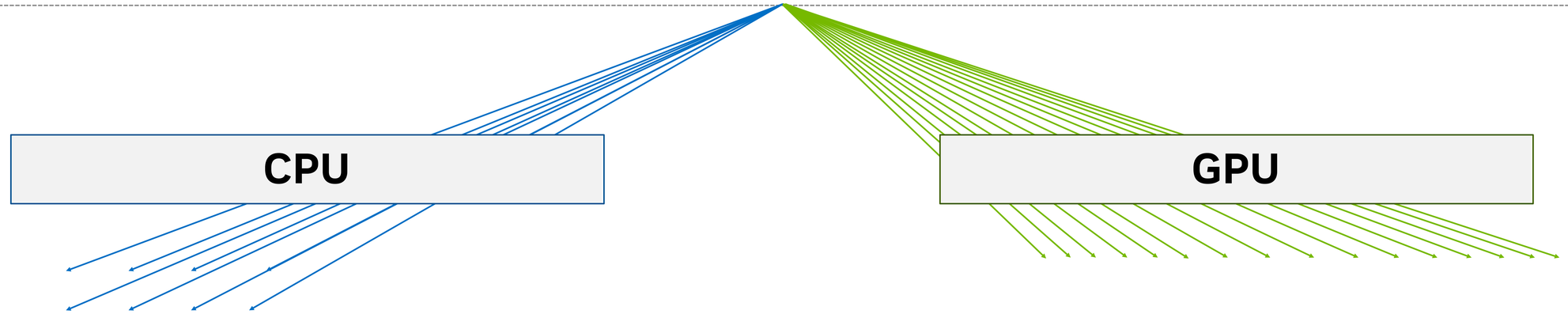
```
std::transform(par, x, x+n, y,
   y,[=](float x, float y){
      return y + a*x;
    }
);
```

ISO Fortran

```
do concurrent (i = 1:n)
   y(i) = y(i) + a*x(i)
enddo
```

Python

```
import cunumeric as np
…
def saxpy(a, x, y):
    y[:] += a*x
```

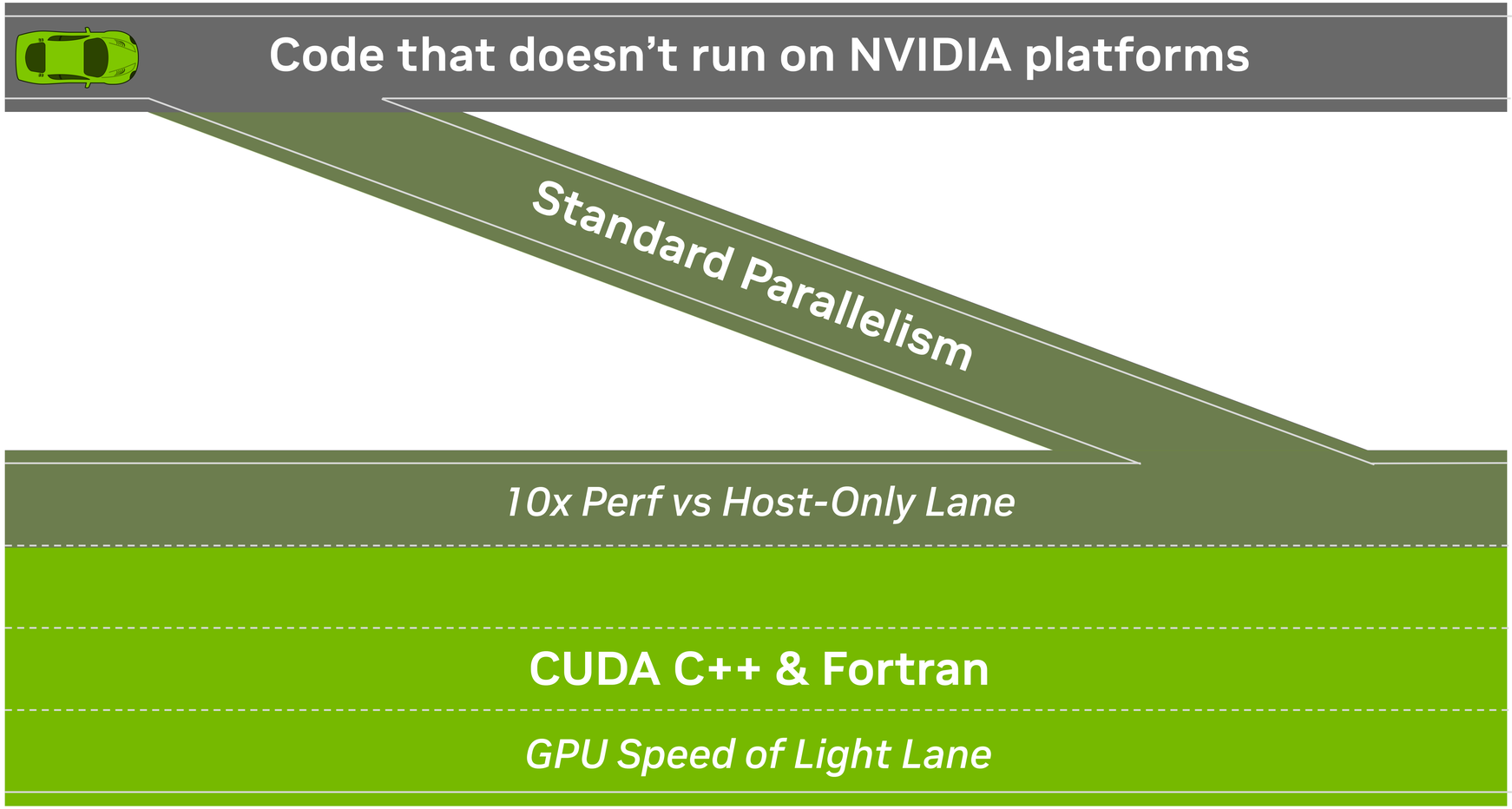



```
nvc++ -stdpar=multicore
nvfortran –stdpar=multicore
legate –cpus 16 saxpy.py
```

```
nvc++ -stdpar=gpu
nvfortran –stdpar=gpu
legate –gpus 1 saxpy.py
```

Scientists Need On-Ramps

Promote Parallelism, not Heterogeneity



Benefits of Adopting Standard Language Parallelism

- For initial code development, standard parallelism is a great way to add CPU and GPU parallelism without having to learn directives
- For codes already using directives like OpenMP or OpenACC, refactoring to use standard parallelism can help:
  - Cleaning up the code for those who do not know directives, or removing the large numbers of directives that make the source code distracting
- **Increases portability** of the code in terms of vendor support and longevity of support
- **Future-proofs** the code, as ISO-standard languages have a proven track record for stability and portability
- **Unified codebase** for both CPUs and GPUs
- **Interoperability** with other parallel programming models
  - E.g., OpenMP, OpenACC, CUDA

https://commons.wikimedia.org/wiki/File:Ferrari_F355_Spider_-_Flickr_-_The_Car_Spy_(16).jpg

What is the GPU Gearbox?

The GPU gearbox is a mental model for thinking about programming models, to deliver the best performance at different levels of developer effort and specialization.

Think about torque, not speed…

First Gear

ISO standard parallelism: Easiest to adopt. Maximum portability. Good performance in a subset of use cases.

Second Gear

Performance libraries: Peak performance for supported features, which include a wide range of common patterns in linear algebra, machine learning and data analysis.

Third Gear

Directives and Pragmas: Easy to adopt. Good portability. Great performance in many use cases.

Fourth Gear

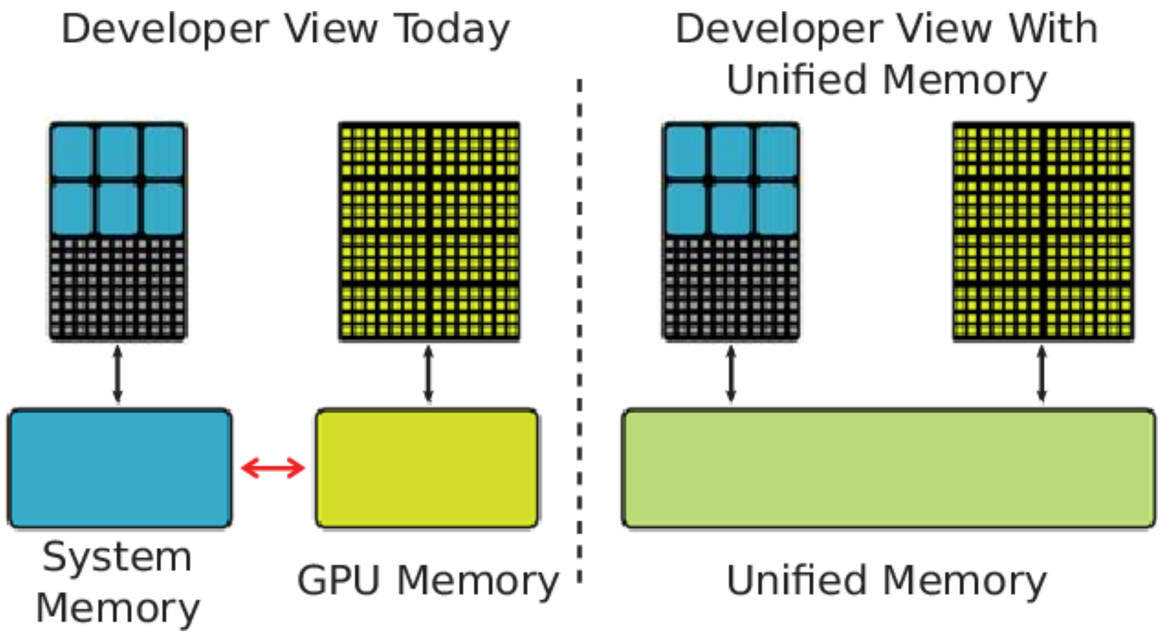
CUDA languages: Exposes full hardware capability and enables maximum performance. Supported on all NVIDIA GPUs.

Unified Memory

Dramatically Lowering Developer Effort

CUDA **Unified Memory** allows a program to dynamically allocate data that can be accessed from the CPU or from the GPU, with the same pointer and address

- Introduced with CUDA 6 / Kepler GK10x
- Leverages UVA (Unified Virtual Addressing) beyond GPU memory pools
- CUDA driver managed page migration CPU <--> GPU on-demand when data is requested (*first touch*)
- Can be controlled / tweaked with prefetching APIs and by define User Policy
  - `ReadMostly`
  - `PreferredLocation`
  - `AccessedBy`

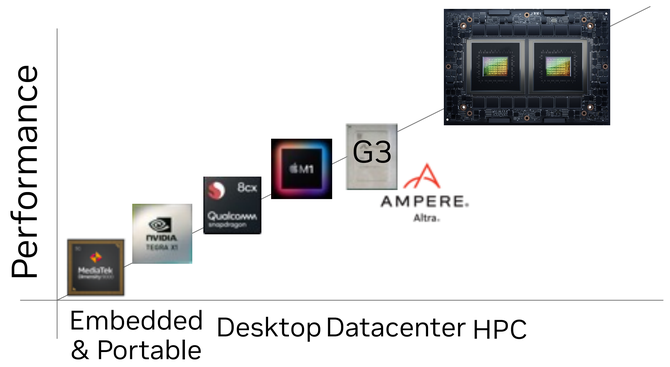


Programming the NVIDIA Grace Superchip

NVIDIA

Performance Backed by Arm Standards

Full performance with full compatibility



docker

KVM

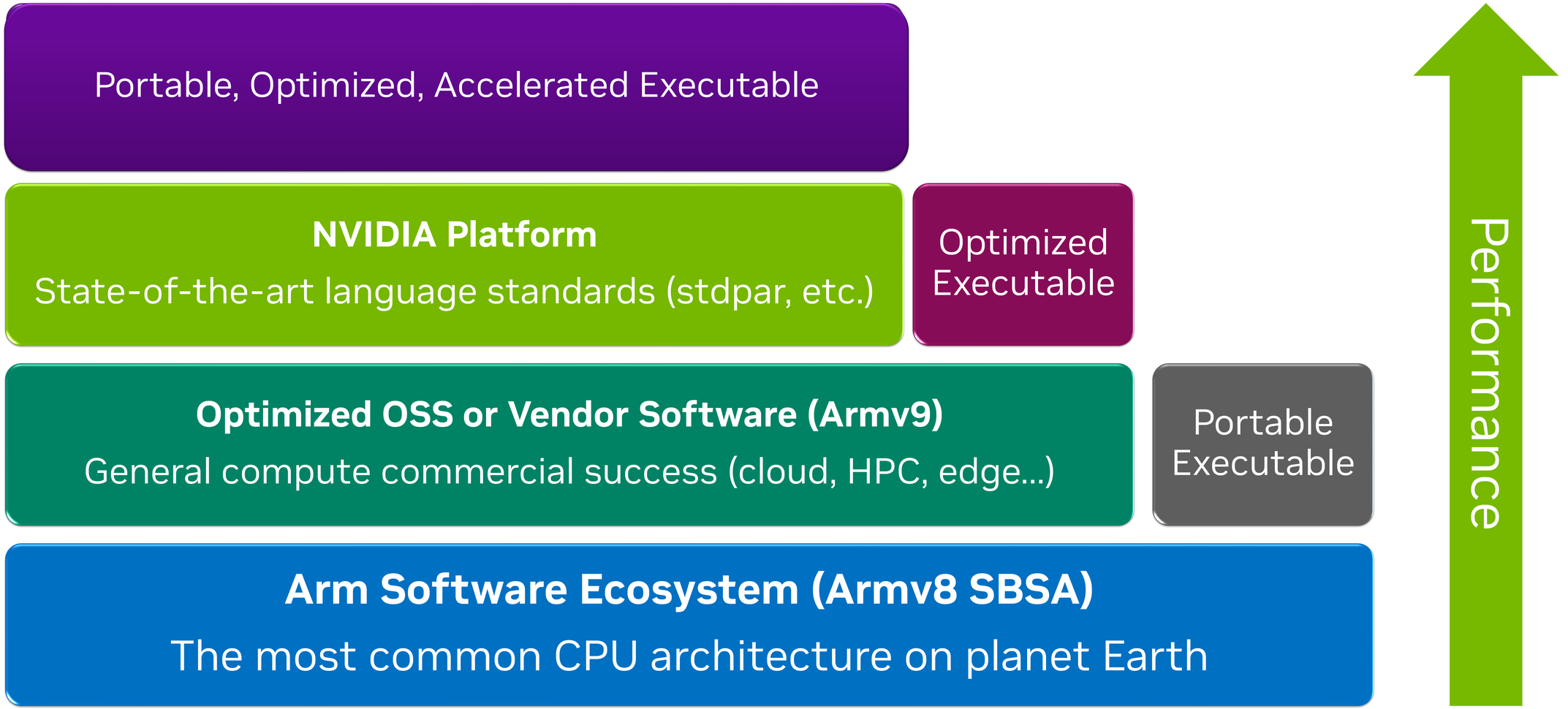
Red Hat

SUSE

Standards-based Arm V9 Core	Performance Gains with Complete Compatibility	One Architecture for Edge, Mobile, Cloud, On-Prem
NVIDIA adds value to a diverse ecosystem that exists anywhere CPUs are found– with all NVIDIA software available on Arm.	Optimizations for Grace are an optimization for the entire Arm ecosystem, maximizing customer and partner ROI. Grace's simple design means less tuning.	The same containers, operating systems, and application binaries that run on Graviton, Altra, and other platforms run on Grace, but faster.

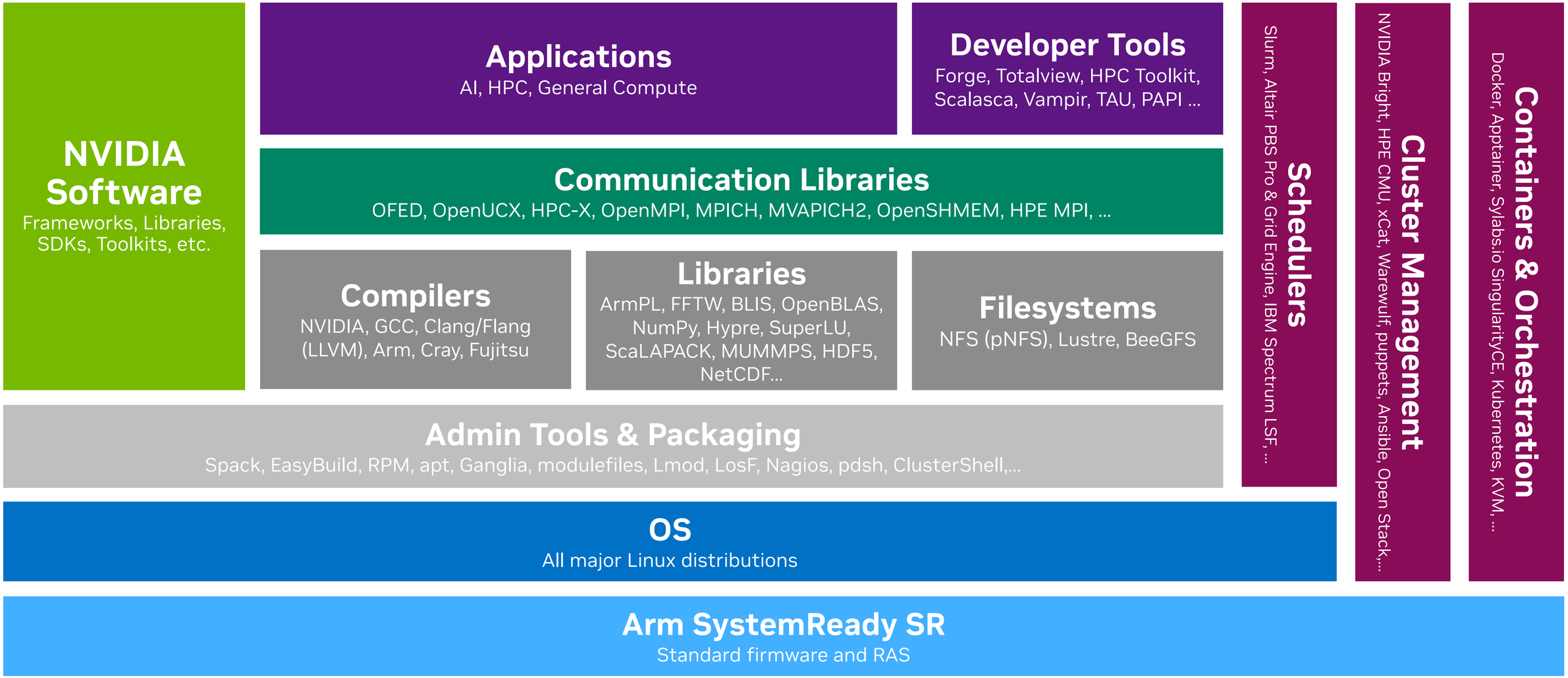
Grace Software Ecosystem is Built on Standards + NVIDIA's Ecosystem

Grace brings the full NVIDIA software stack to Arm.

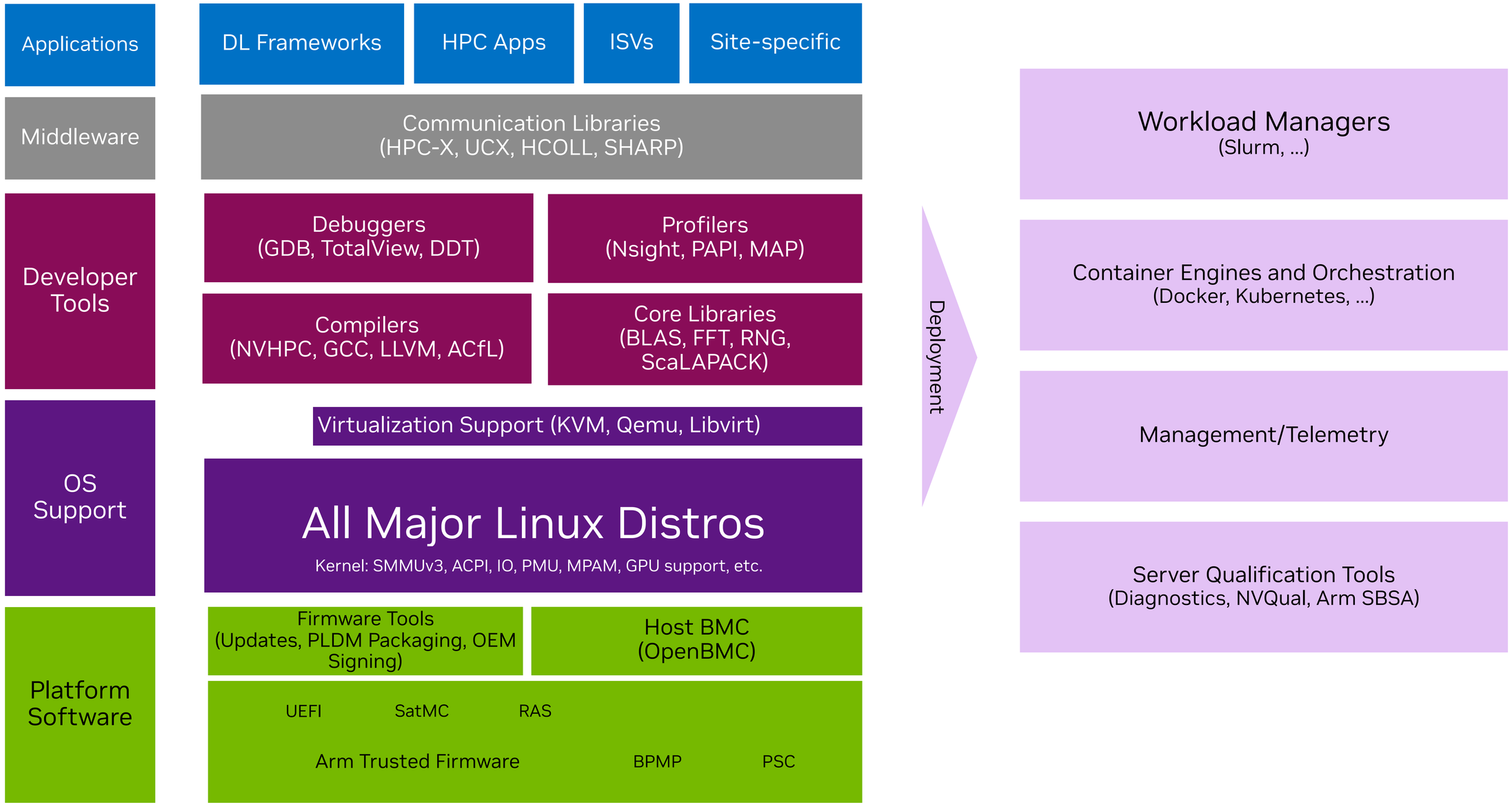


NVIDIA Grace HPC/AI Software Ecosystem

Full support for the broad Arm software ecosystem, both open source and commercial



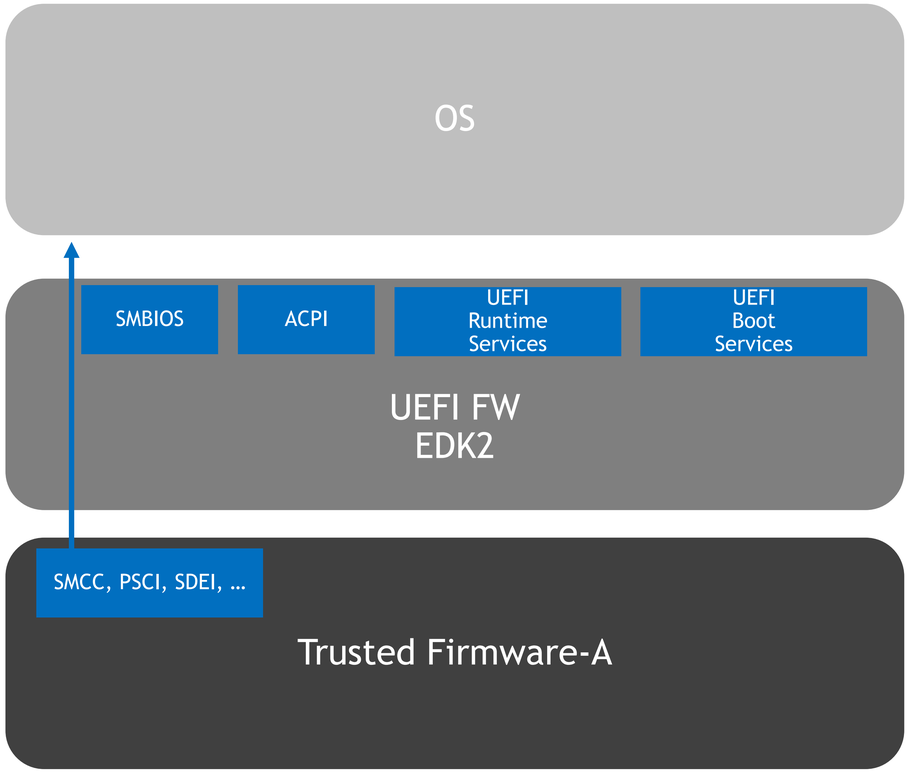
An Example Software Stack for Grace



Grace

Server Base System Architecture (SBSA)
Base Boot Requirements (BBR)

- Standard set of platform requirements and recommendations to enable off-the-shelf OS support
- SBBR recipe support from BBR
- Allows OS and system SW to expect consistency across different SOCs
  - Standard Private Peripheral Interrupt (PPI) assignments
  - Standard UART
  - PCIe – ECAM, ITS for MSI(-X)

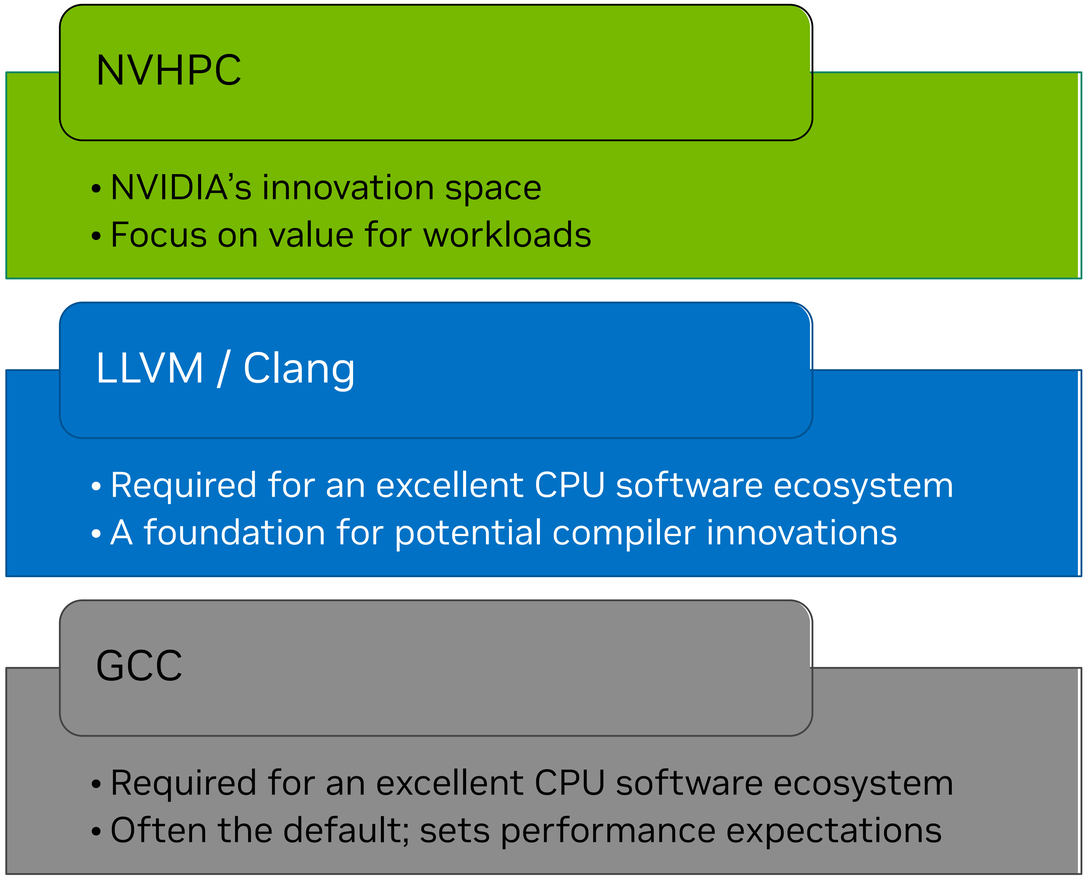


arm SystemReady

NVIDIA’s Approach to Grace+/-Hopper Compilers

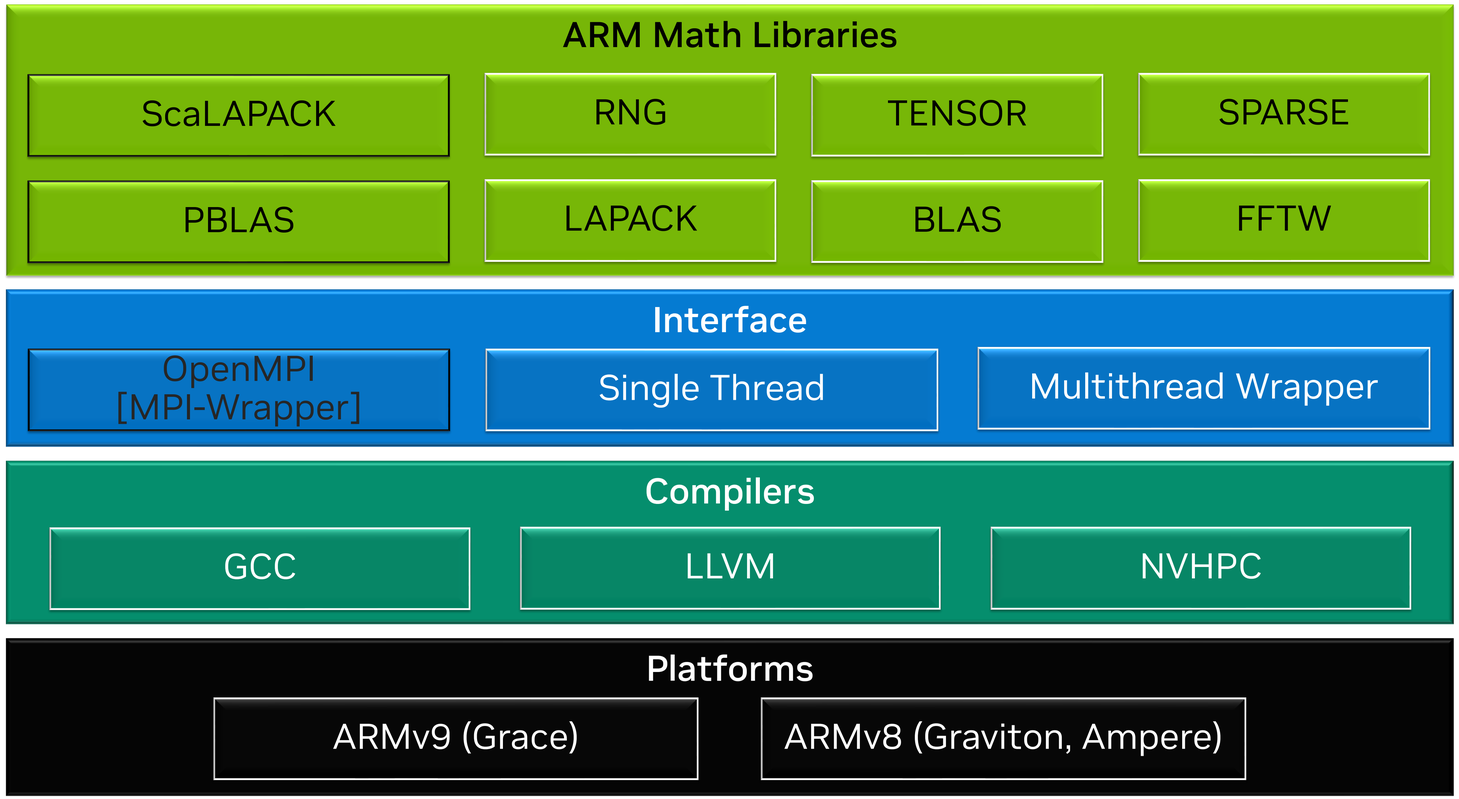
Use the compilers you know and love

- **NVHPC and NVCC**
  - 23.5+ supports Grace and Neoverse V2
  - Continuously improving performance
- **LLVM Clang**
  - LLVM16+ supports -mcpu=neoverse-v2
  - NVIDIA will provide supported builds of Clang as drop-in replacements for mainline Clang
  - NVIDIA contributing to LLVM in open source to maximize Grace application performance
  - Engaging with Arm (Inc.) to improve performance for key CPU benchmarks and applications
- **GCC**
  - GCC12.2+ supports -mcpu=neoverse-v2
  - Engaging with Arm (Inc.)
  - Already finding and fixing bugs in mainline GCC



NVPL: Math Libraries for Arm CPUs

Early Access
In Q3’23

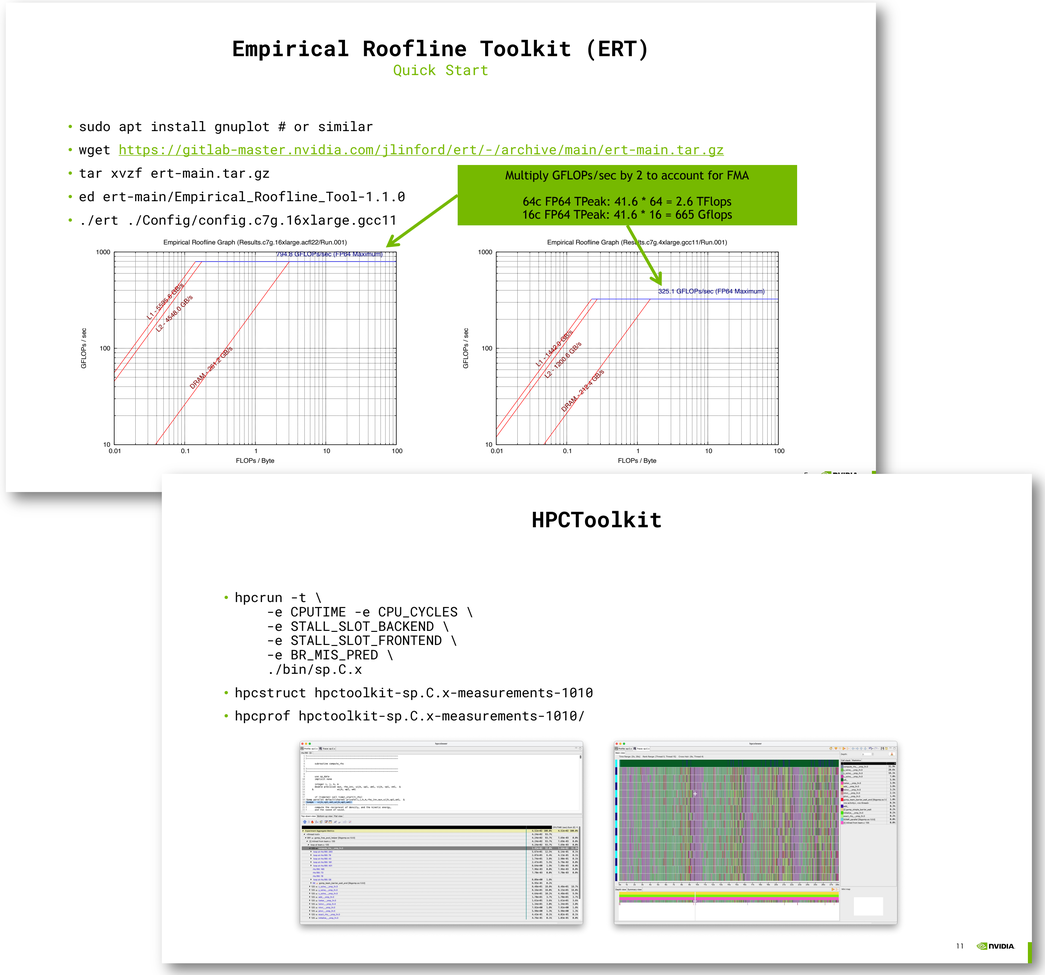


* ARMv8.2 minimum compatibility

Performance Engineering Tools

A mix of NVIDIA tools and community tools – developers want to use the same tools on Grace

- **Objective**: Enable Grace developers to the same extent that they are enabled on x86
- The NVIDIA NSIGHT team are defining CPU profiling workflows that enable Grace developers
- NVIDIA is working with community tools to better support Grace
  - E.g. PAPI and libpfm4 now support Grace in-core counters



PAPI & libpfm4 with Arm Neoverse Support

Upstreamed and maintained by the community

- `cd src`
- `./configure --prefix=$PWD/..`
- `make -j && make install`

```
[jlinford@c7g-4xlarge-dy-c7g4xlarge-2 papi-jlinford-arm-neoverse]$ ./bin/papi_native_avail | grep SVE
|           Data memory read accesses (includes SVE)                          |
|           Data memory write accesses (includes SVE)                         |
|           Unaligned accesses (includes speculatively executed SVE load and s|
| SVE_INST_SPEC                                                               |
|           SVE operations sepculatively executed                             |
| SVE_PRED_SPEC                                                               |
|           SVE predicated operations speculatively executed                  |
| SVE_PRED_EMPTY_SPEC                                                         |
|           SVE predicated operations with no active predicates speculatively |
| SVE_PRED_FULL_SPEC                                                          |
|           SVE predicated operations with all active predicates speculatively|
| SVE_PRED_PARTIAL_SPEC                                                       |
|           SVE predicated operations with partially active predicates specula|
| SVE_LDFF_SPEC                                                               |
|           SVE first-fault load operations speculatively executed            |
| SVE_LDFF_FAULT_SPEC                                                         |
|           SVE first-fault load operations speculatively executed which set F|
```

```
[jlinford@c7g-4xlarge-dy-c7g4xlarge-2 papi-jlinford-arm-neoverse]$ ./bin/papi_avail | grep Yes
PAPI_L1_DCM  0x80000000  Yes   No   Level 1 data cache misses
PAPI_L1_ICM  0x80000001  Yes   No   Level 1 instruction cache misses
PAPI_L2_DCM  0x80000002  Yes   No   Level 2 data cache misses
PAPI_TLB_DM  0x80000014  Yes   No   Data translation lookaside buffer misses
PAPI_L2_LDM  0x80000019  Yes   No   Level 2 load misses
PAPI_STL_ICY 0x80000025  Yes   Yes  Cycles with no instruction issue
PAPI_HW_INT  0x80000029  Yes   Yes  Hardware interrupts
PAPI_BR_MSP  0x8000002e  Yes   No   Conditional branch instructions mispredicted
PAPI_BR_PRC  0x8000002f  Yes   Yes  Conditional branch instructions correctly predicted
PAPI_TOT_INS 0x80000032  Yes   No   Instructions completed
PAPI_FP_INS  0x80000034  Yes   No   Floating point instructions
PAPI_LD_INS  0x80000035  Yes   No   Load instructions
PAPI_SR_INS  0x80000036  Yes   No   Store instructions
PAPI_BR_INS  0x80000037  Yes   No   Branch instructions
PAPI_VEC_INS 0x80000038  Yes   Yes  Vector/SIMD instructions (could include integer)
PAPI_RES_STL 0x80000039  Yes   No   Cycles stalled on any resource
PAPI_TOT_CYC 0x8000003b  Yes   No   Total cycles
PAPI_LST_INS 0x8000003c  Yes   Yes  Load/store instructions completed
PAPI_SYC_INS 0x8000003d  Yes   Yes  Synchronization instructions completed
PAPI_L1_DCA  0x80000040  Yes   No   Level 1 data cache accesses
PAPI_L2_DCA  0x80000041  Yes   Yes  Level 2 data cache accesses
PAPI_L1_DCR  0x80000043  Yes   No   Level 1 data cache reads
PAPI_L2_DCR  0x80000044  Yes   No   Level 2 data cache reads
PAPI_L1_DCW  0x80000046  Yes   No   Level 1 data cache writes
PAPI_L2_DCW  0x80000047  Yes   No   Level 2 data cache writes
PAPI_L1_ICH  0x80000049  Yes   Yes  Level 1 instruction cache hits
PAPI_L1_ICA  0x8000004c  Yes   No   Level 1 instruction cache accesses
PAPI_L2_TCA  0x80000059  Yes   No   Level 2 total cache accesses
```

APPLICATION PORTING: MANY NON-TRIVIAL CASES REALLY ARE TRIVIAL

Vector intrinsics, dependencies, and nonstandard features are easily ported

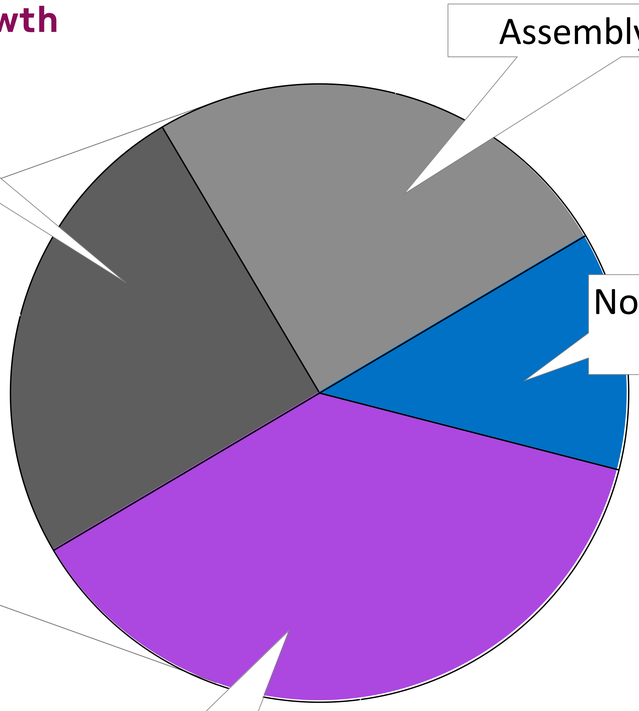
Straightforward, easy work < 1 day

Recompile and reconfigure runtime parameters

Job done!

Found on Arm at another HPC center

Cloud momentum
Arm ecosystem growth



Intrinsic conversion tools – SIMDe, SSE2NEON, etc.

**Indicative workload mix inspired by an US DoE lab usage*

PORTING ASSEMBLY AND VECTOR INTRINSICS

Translate intrinsics to port functionality, then focus on performance tuning

- For a quick fix, use a drop-in header-based intrinsics translator
  - SIMD Everywhere (SIMDe): https://github.com/simd-everywhere/simde
  - SSE2NEON: https://github.com/DLTcollab/sse2neon
  - Tutorial using BWA-MEM2: https://www.nvidia.com/en-us/on-demand/session/gtcspring22-s41702/

- Follow Arm's documentation on rewriting x86 vector intrinsics
  - Porting and Optimizing HPC Applications for Arm SVE [https://developer.arm.com/documentation/101726/latest]
  - Coding for NEON [https://developer.arm.com/documentation/101725/0300/Coding-for-Neon]

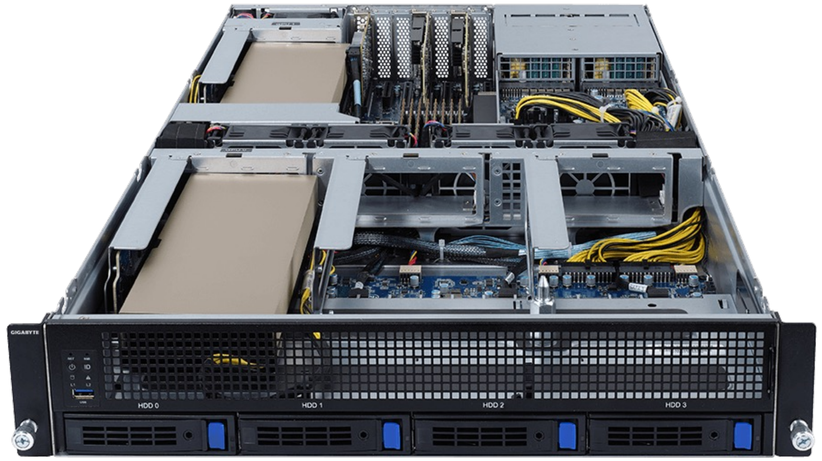
- Arm assembly is simpler than x86
  - Arm processors have a much simpler and general set of registers than x86. Just assign a one-to-one mapping from an x86 register to an Arm register when porting code.
  - Complex x86 instructions will become multiple Arm instructions

NVIDIA ARM HPC DEVELOPER KIT

Ampere Computing ‘Altra’ CPU + NVIDIA A100 GPU (April 2021, July 2021 GA)

Enablement Program Objectives

- Provide HPC and AI developers fully with an Arm-based affordable development platform with NVIDIA Compute (dual A100 PCIe GPU) and Networking (dual BF2 DPU) technology
- Enable developers to testing / porting / verify HPC and AI software on Arm, removing (if any) x86 dependencies hence de-risking Grace adoption



GIGABYTE™

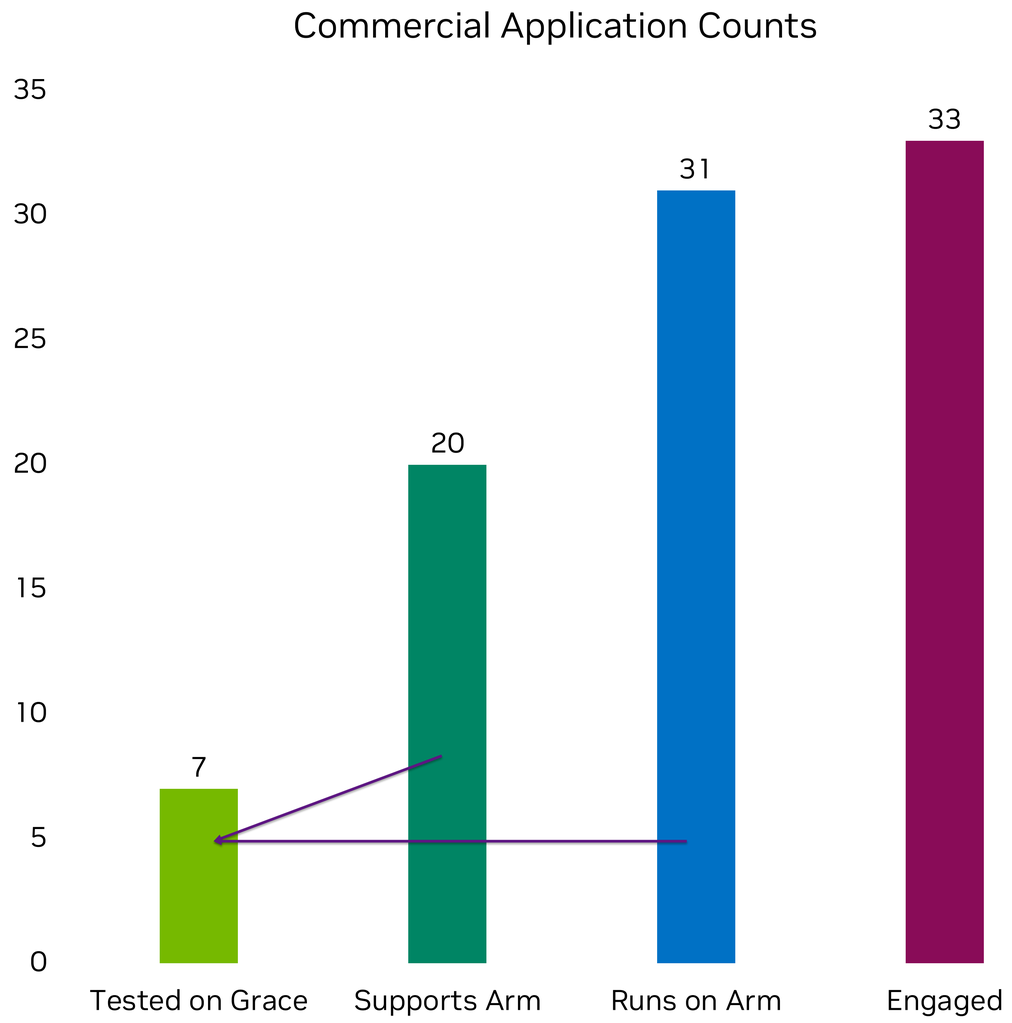
NVIDIA

Getting Started with HPC on Arm

https://github.com/arm-hpc-devkit/nvidia-arm-hpc-devkit-users-guide

- How-to guides, sample code, recommendations, and technical best practices for new Arm HPC users
- Generically useful for anyone running HPC applications on Arm CPUs, with or without GPUs
- Tracker of commercial codes supporting Arm
- Language-specific considerations for:
  - C/C++
  - Fortran
  - Python
  - Rust
  - Go
  - Java
  - .NET
- Optimization guides for Arm hardware features
  - SVE and NEON SIMD instructions
  - LSE atomic instructions
- Worked examples
  - OpenFOAM
  - WRF
  - Gromacs
  - GPU TensorFlow
  - CPU-only TensorFlow
  - Anaconda/Miniconda
  - Arkouda
  - Velox
  - CPU HPL
  - CPU Stream

NVIDIA Grace Commercial Applications Workstream



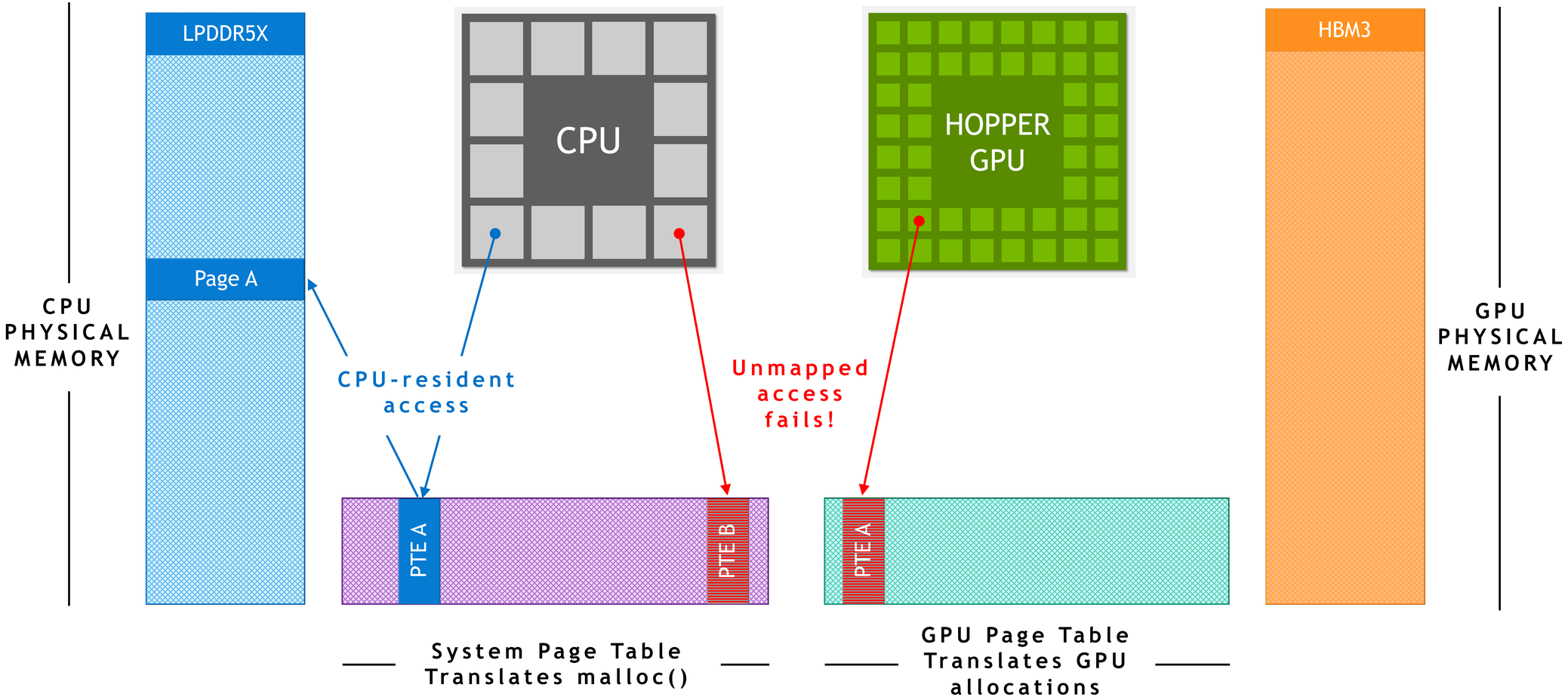
- Many commercial independent software vendors (ISVs) are accelerating their plans for Arm support following Grace announcements
- ISVs currently supporting Arm are pressing forward with Grace CPU optimization and Grace-Hopper performance analysis
- NVIDIA continues to engage and enable

Programming the NVIDIA Grace Hopper Superchip

NVIDIA

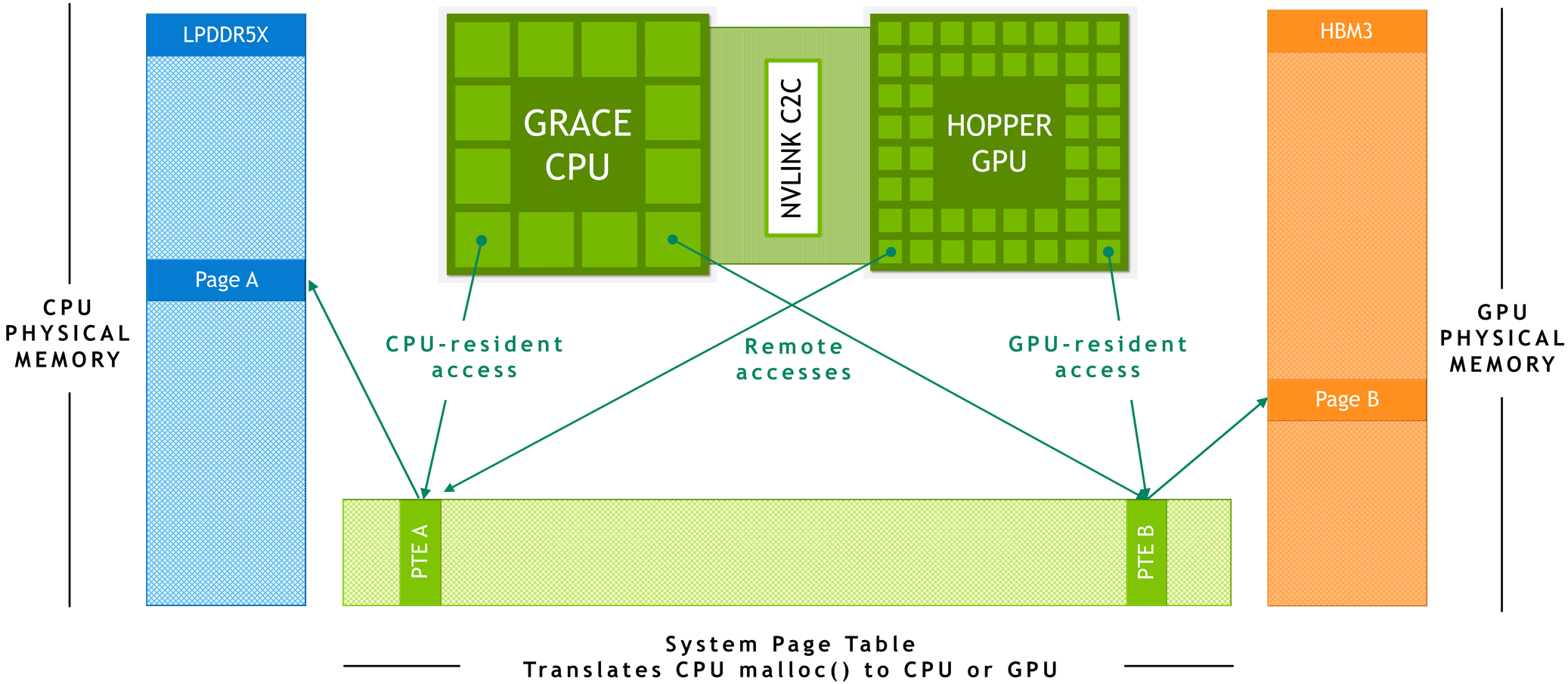
HW/SW memory view on X86 + GPU

Separate page tables



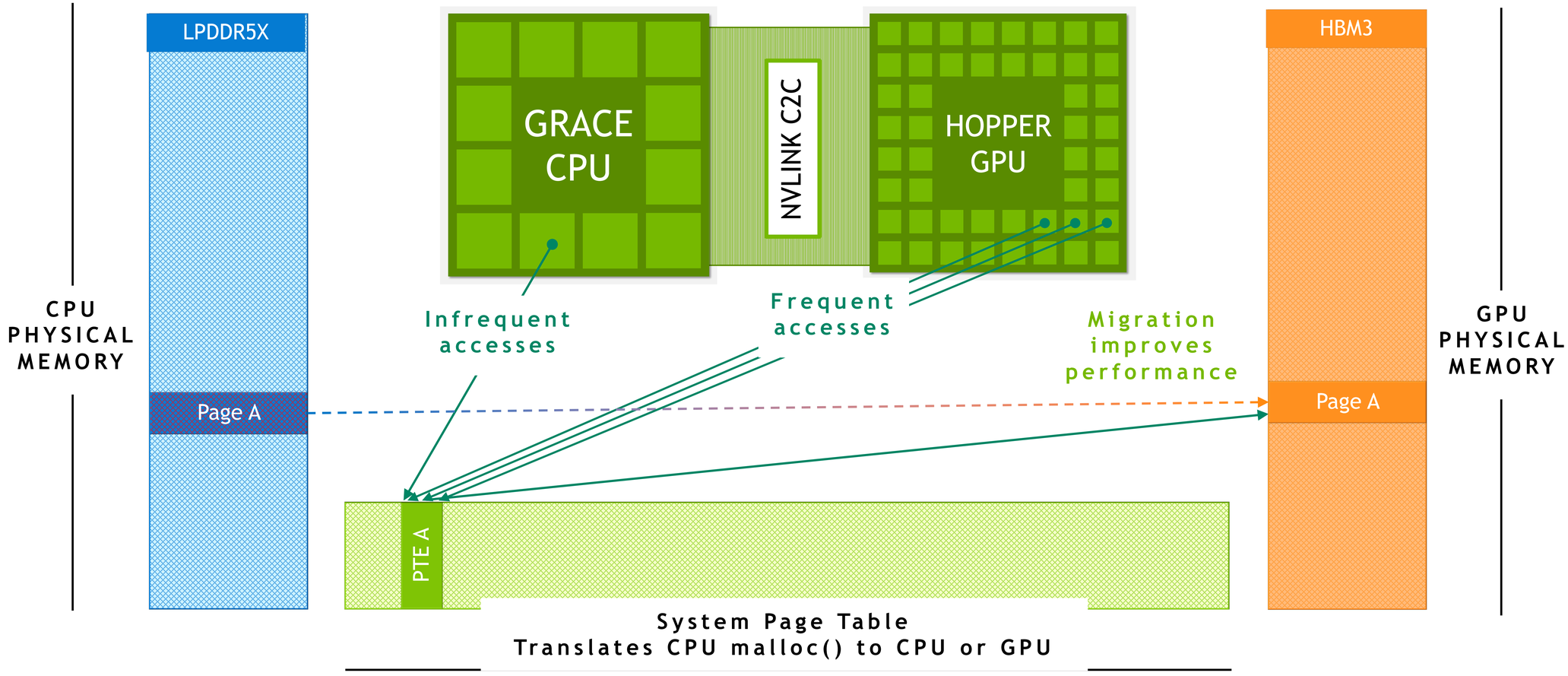
HW/SW memory view on Grace Hopper Superchip

Simplified Unified Memory via **Address Translation Service (ATS)**



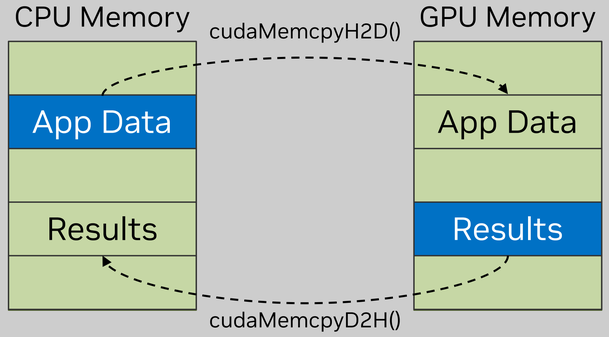
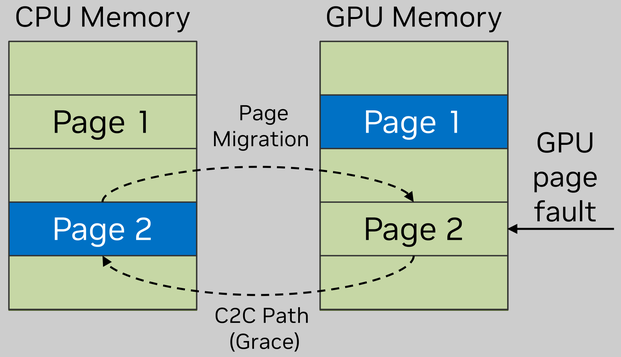
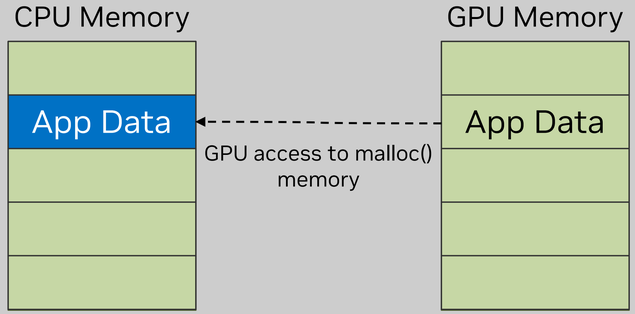
HW/SW memory view on Grace Hopper Superchip

Automatic Migrations



ADVANTAGES OF THE GRACE HOPPER MEMORY MODEL

Full CUDA support with additional Grace memory extensions

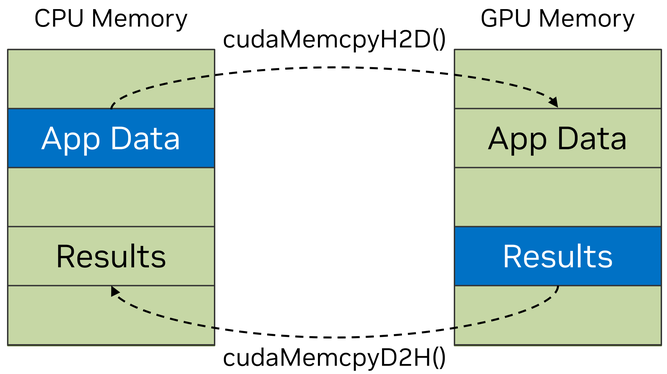
	Explicit Copy	Managed Memory	System Allocated
	Application explicitly moves data between CPU & GPU as needed	CPU and GPU can access memory on-demand and data migrated locally for higher BW access	GPU can access memory allocated from `malloc()`, `mmap()`, etc.
			
HGX	~60 GB/s PCIe Gen5 transfers (H2D/D2H)	Requires migration to GPU	Access possible with explicit call to `cudaHostRegister()` at PCIe speeds Requires HMM patch in Linux Kernel
G+H	7x faster transfers, up to 450 GB/s (NVLink C2C)	Migrations not required and faster migrations when they happen at NVLink C2C speed	`cudaHostRegister()` not needed; access at NVLink C2C speeds

Explicit Data Movement (bulk/blocking)

Works on both x86 and Grace Hopper

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

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

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

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

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

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

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

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

- Allocation done on CPU in the “usual way” (malloc)
- cudaMalloc continues to behave as x86 platform, GPU memory cannot be accessed on CPU.
- Uses GPU MMU for translation



PCIE: ~60 GB/s PCIE transfers (H2D/D2H)

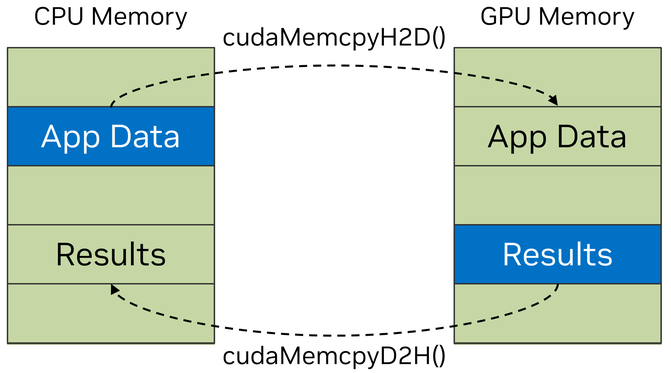
Grace: Faster transfers; up to 450 GB/s C2C transfers (per direction)

Explicit Data Movement (async)

Works on both x86 and Grace Hopper

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

int main()
{
  cudaStream_t stream;
  cudaStreamCreate(&stream);
  int N = 128;
  int *data = (int *)malloc(N * sizeof(int));
  int *d_data;

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

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

  cudaMemcpyAsync(d_data, data, N * sizeof(int),
                               cudaMemcpyHostToDevice, stream);
  kernel<<<1, N, 0, stream>>>(d_data);
  cudaMemcpyAsync(d_data, data, N * sizeof(int),
                               cudaMemcpyDeviceToHost, stream);
  cudaDeviceSynchronize();

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

- **On x86**: allocates page locked memory on Host (*cudaMallocHost*) or pinning existing Host memory (*cudaHostRegister*)
- **On Grace-Hopper**:
  - Host page locking is not needed as ATS can be used to access data to GPU or direct access
  - *cudaMallocHost* is no longer needed
    - But cudaMallocHost == malloc + cudaHostRegister
  - *cudaHostRegister* is no longer needed.
    - Calling it can populate pages as it is considered as first touch
    - Sets the preferred location to host.



PCIE: ~60 GB/s PCIE transfers (H2D/D2H)

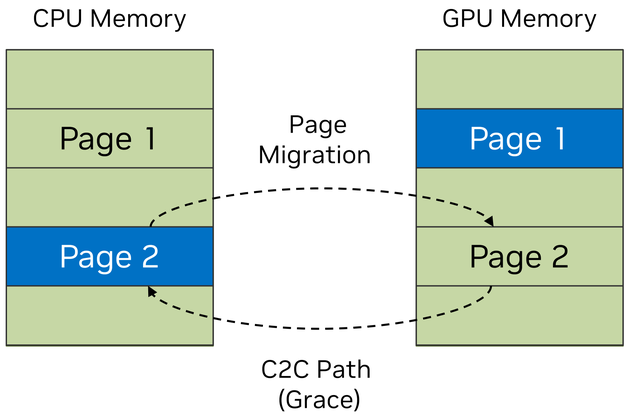
Grace: Faster transfers; up to 450 GB/s C2C transfers (per direction)

Managed Memory

Portable codes across x86 and Grace Hopper

```cpp
__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; }

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

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

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

  cudaFree(data);
}
```

- CPU and GPU can access memory on-demand and data migrated locally for higher BW access
- CUDA UVM has been available since Kepler architecture and performance has been tuned since then.
- **On x86 & Grace-Hopper**:
  - User migration hints using CUDA API (*cudaMemPrefetchAsync*, *cudaMemAdvise*)
  - cudaMemPrefetchAsync can be used to move memory pages under ATS



PCIE: Requires migration to GPU

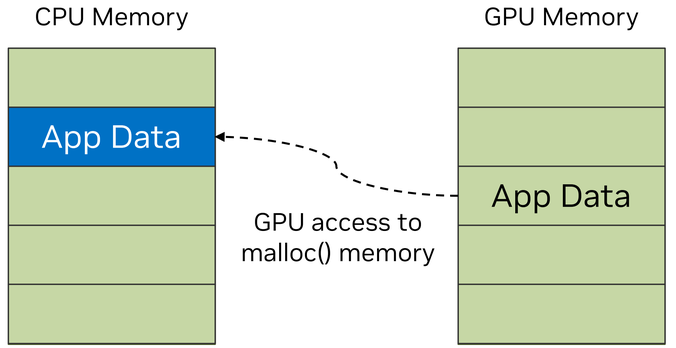
Grace: Migrations not required and faster migrations when they happen

Using System Allocator

Super easy to move codes to GPU, portable with HMM on x86

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

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

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

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

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

  free(data);
}
```

- **On x86**: enabled by HMM, data migrates based on page fault mechanism
- **On Grace-Hopper**:
  - GPU can access memory allocated from `malloc()`, `mmap()`, etc.
    - Both stack and heap are accessible
  - Translation done via ATS
  - Direct load/store can be performed by both CPU and GPU. No fault-based migration.
  - Fine grained synchronization can be done using atomics
  - Data can migrate based on user hints or driver heuristics.



PCIE: Access possible with explicit call to `cudaHostRegister()` at PCIe speeds

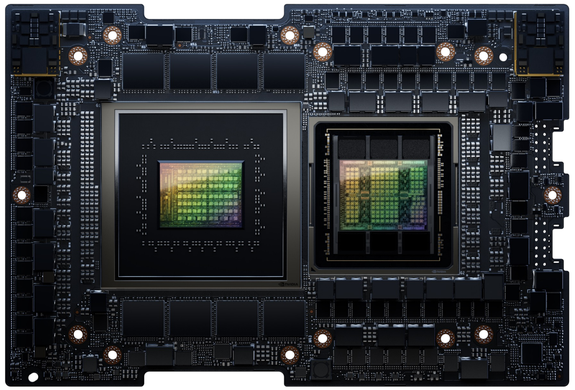
Grace: `cudaHostRegister()` not needed; access at NVLink C2C speeds

GH200 Grace Hopper HPC Platform

Unified Memory and Cache Coherence for Next Gen HPC Performance

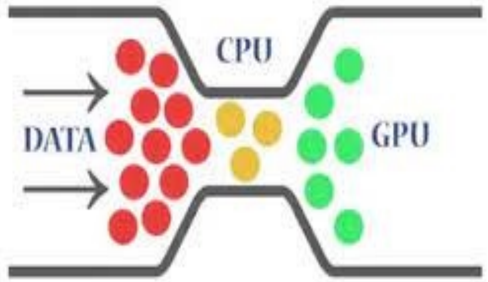
Partially GPU Accelerated Apps

Big performance gains with no code changes



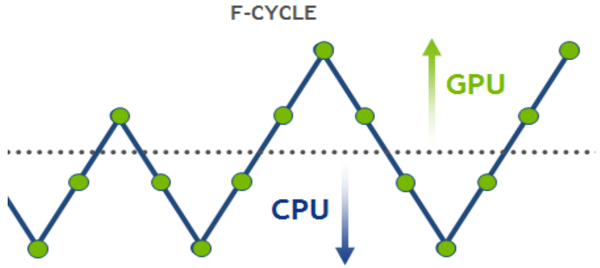
No More PCIe Bottleneck

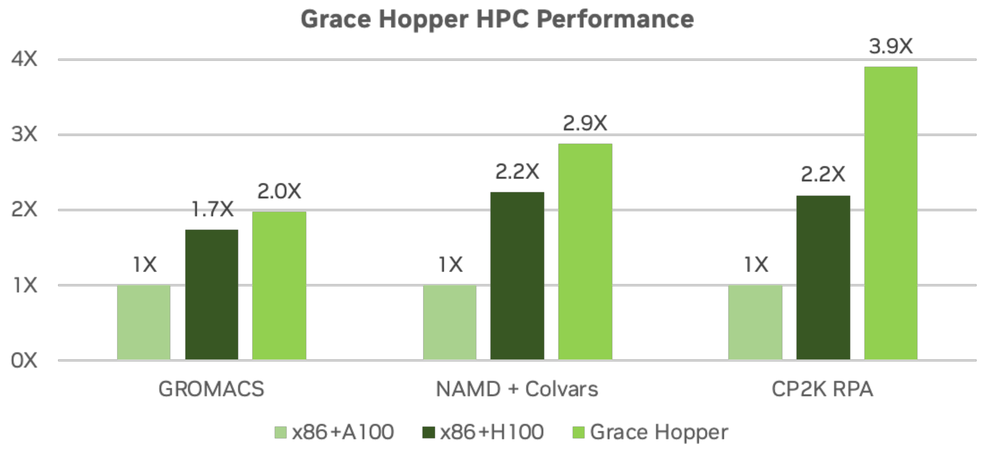
NVLink-C2C is 7X PCIe BW



CPU & GPU Cache Coherence

Incremental code changes yield big gains





Fast Access Memory

576GB

Memory Bandwidth

4TB/s

HPC: Preliminary results comparing DGX A100, DGX H100 and Grace Hopper systems.
AI: Simulation results comparing DGX H100 and Grace Hopper systems. GNN: GraphSAGE model for an augmented ogbn- products dataset of 626M nodes and 31B edges, DRLM: Deep Learning Recommendation Model (DLRM) 27TB model, NLP: fine-tuning a GPT-175B model

Application on Accelerated Systems

Partially GPU Accelerated

As GPUs become faster applications become increasingly limited by non-GPU factors

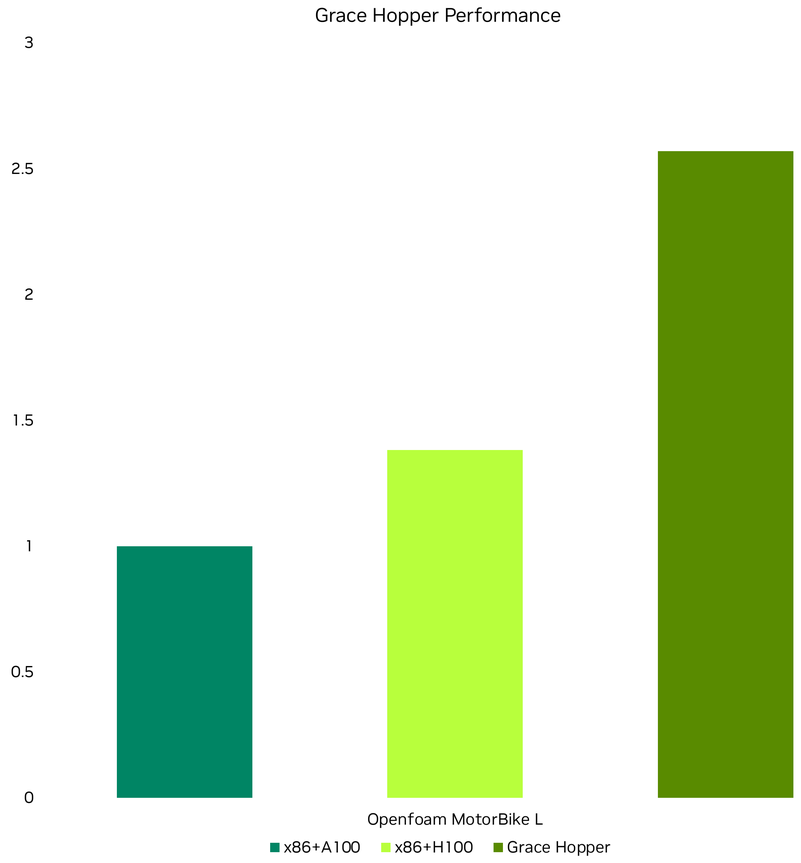
e.g. mostly data transfer (PCIe) limited

- mostly CPU limited

OpenFOAM

Partially GPU Accelerated – mostly CPU limited

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

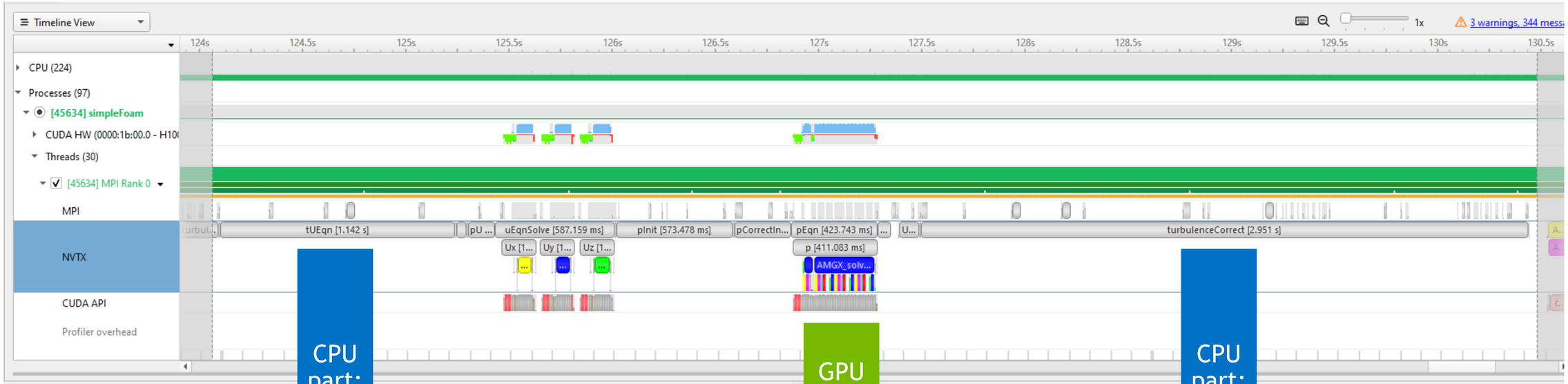


~35M cells benchmark designed by OpenFOAM HPC technical committee

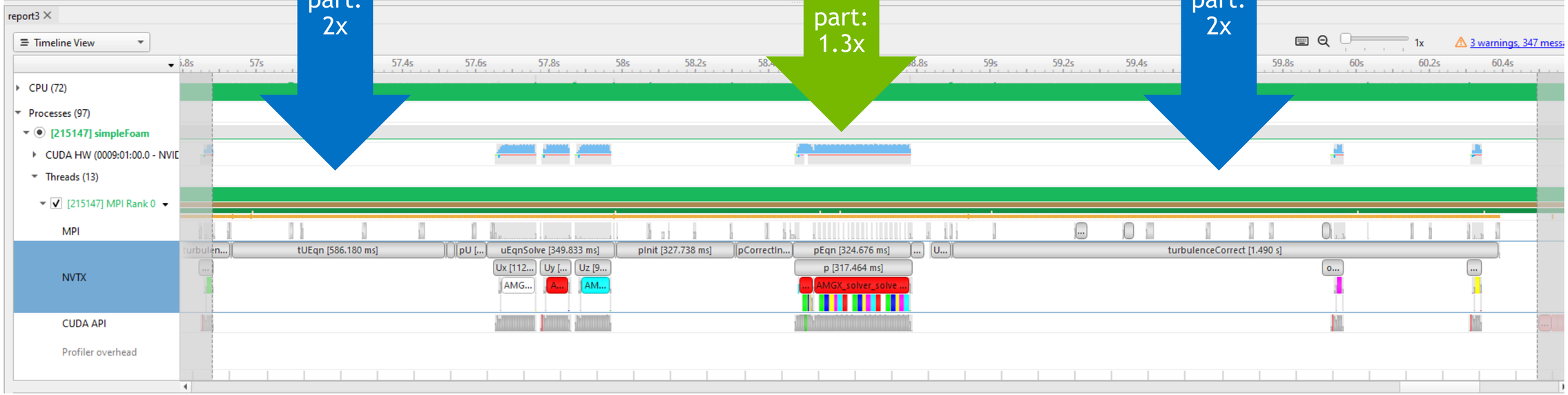
OpenFOAM

Nsight Systems Profile

x86 + H100 (H100 80GB HBM3)



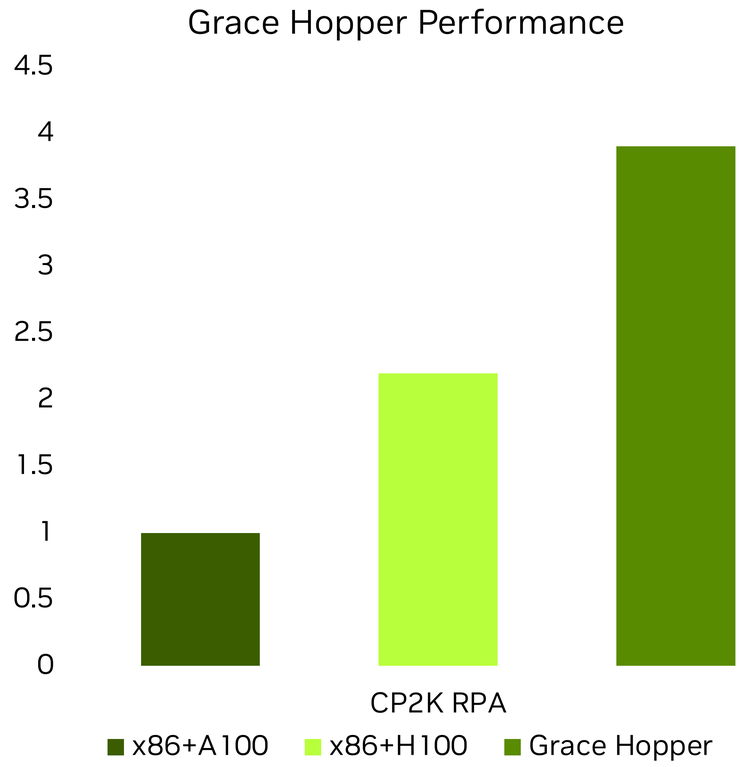
Grace Hopper (H100 96GB HBM3)



CP2K

Partially GPU Accelerated – mostly data transfer limited

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

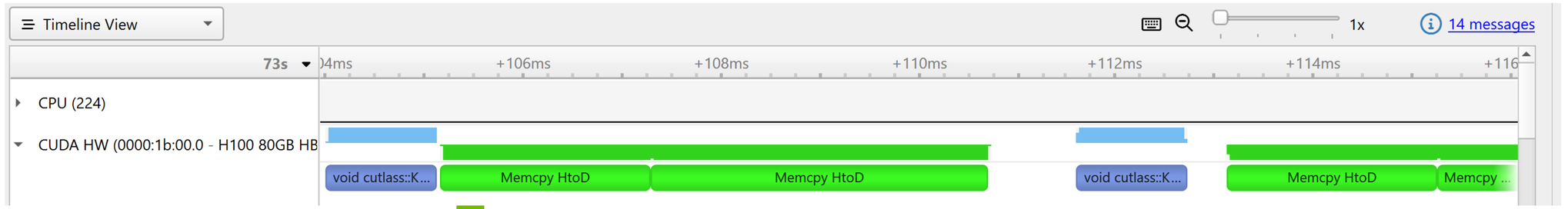


Dataset "128-H2O" with random-phase approximation (RPA) method

CP2K RPA

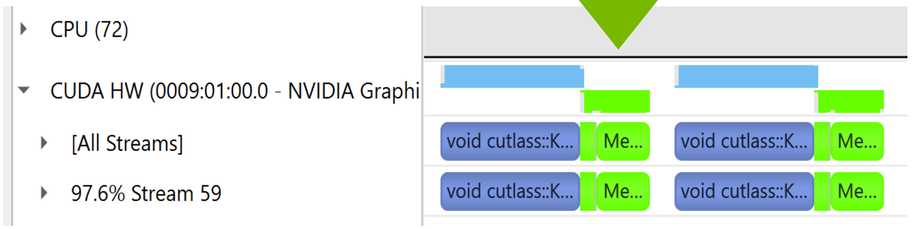
Nsight Systems Profile

x86 + H100 (H100 80GB HBM3)



Host to Device 6.2x

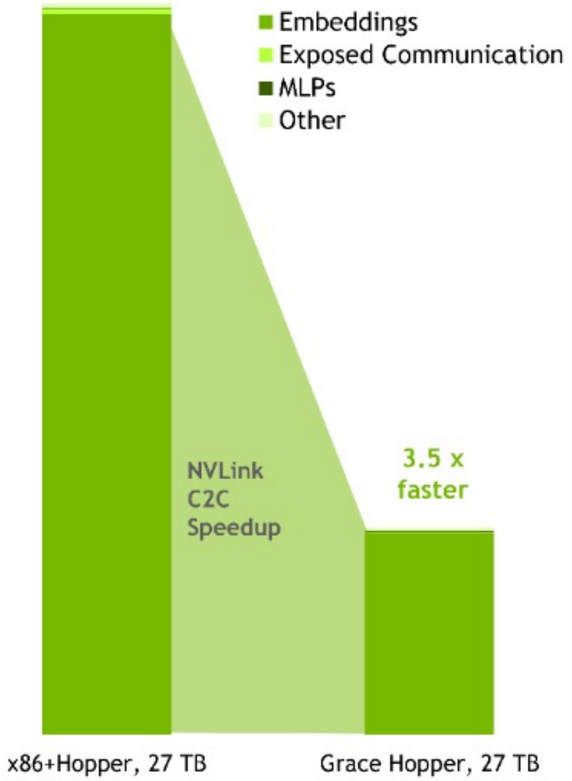
Grace Hopper (H100 96GB HBM3)

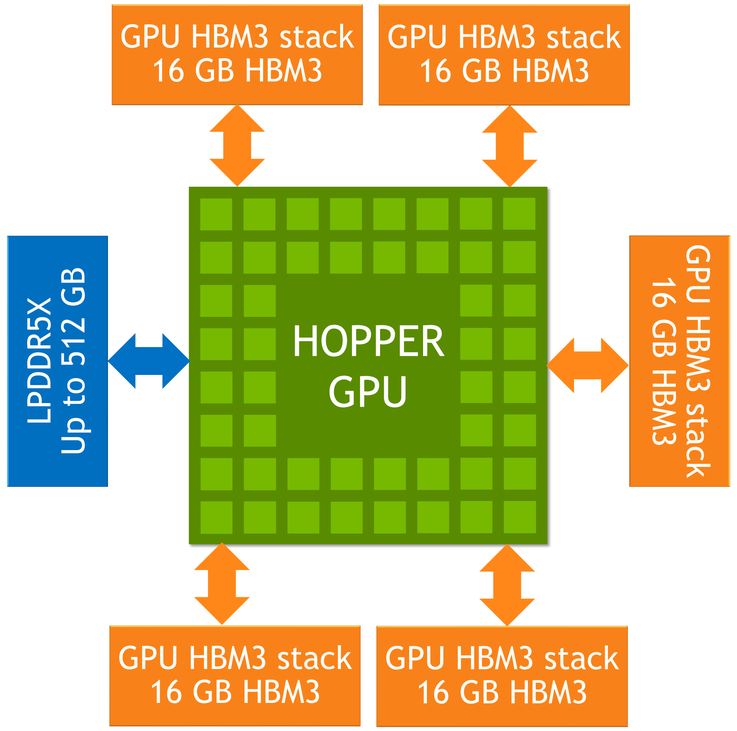


Using CPU as a memory pool

CPU Memory can be thought of large fast access pool

- CPU memory can be thought as “extra stacks” of accessible memory
- Large capacity and slightly further away than HBM3
- Perfect for Deep Recommendation Models (DLRMs)
  - Large embeddings





NVIDIA Superchip Platforms

NVIDIA MGX

Scientific Computing

Data Processing

LLM Training

Gen AI Inference

Cloud Video & Graphics

EDA SDA CADD

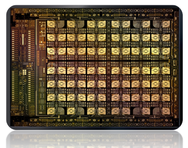
Enterprise Gen AI

Edge AI

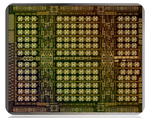
$1T Global Datacenter Infrastructure transitioning to accelerated computing and generative AI

Accelerated Computing

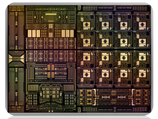
GPU

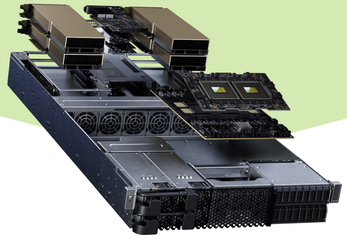


CPU



DPU





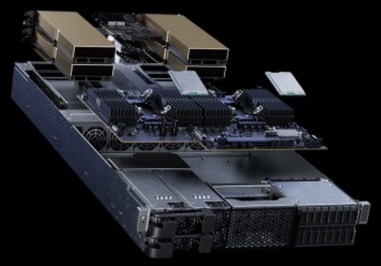
A Modular Reference Architecture for Accelerated Computing

Time-to-Market

Multi-gen Compatibility

Open and Flexible

MGX – Modular Reference Designs - To Enable Large Number of Configurations



2U1N X86 Server

2U | x86 | 4 L40 | BF-3 | 2 CX-7 | 6 PCIE

2U1N Grace Server

2U | Grace | 4 L40 | BF-3 | 2 CX-7 | 6 PCIE

Grace Hopper Server

2U | Grace-Hopper | BF-3 | CX-7 | 6 PCIE

Grace Cloud Gaming Server

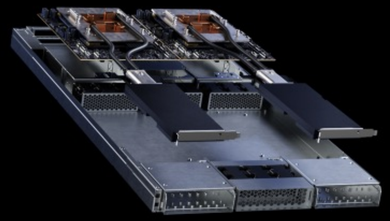
2U | Grace | 10 L4 | BF-3 | 11 PCIE

Grace-Hopper Aerial Server

1U | Grace-Hopper | 2 BF-3 | 4 PCIE

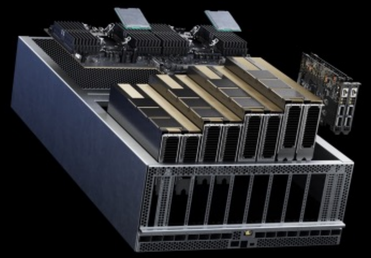
Dense General-Purpose Grace CPU Server

1U | 2 Grace | 2 BF-3 | 4 PCIE



Grace-Hopper Liquid-Cooled Server for HPC

1U | 2 Grace-Hopper | 2 BF-3 | 4 PCIE



Hopper NVL Inference Server

4U | x86 | 8 H100 NVL | 2 BF-3 | 10 PCIE

Grace-Hopper Aerial Server Short Depth

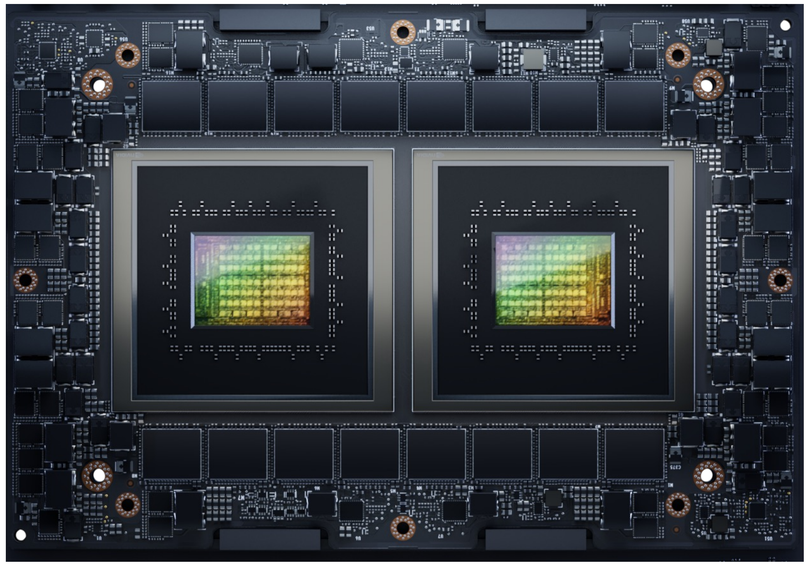
2U 450mm | Grace-Hopper | BF-3 | CX-7 | 3 PCIE


NVIDIA

Further Resources for Grace CPU and Grace Hopper

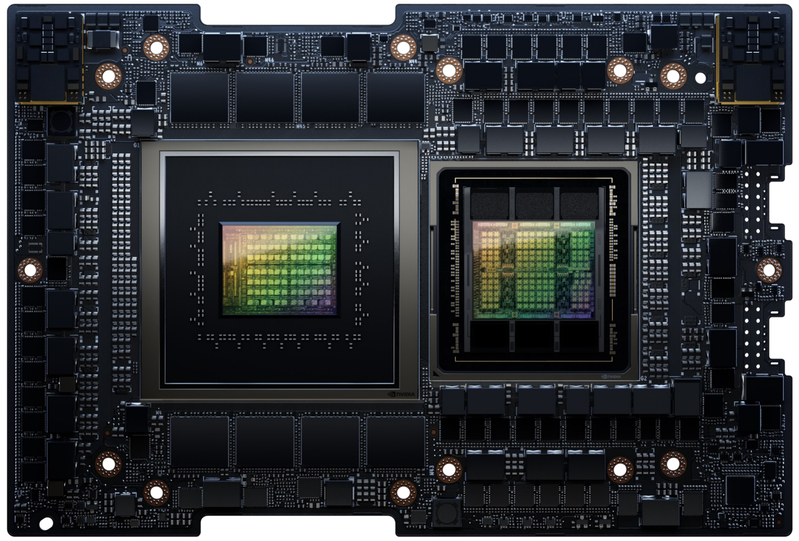
Grace CPU Superchip

- Grace CPU Customer Deck
- Grace CPU Superchip Architecture Whitepaper
- Grace CPU Architecture In-Depth Blog
- Grace CPU Superchip Data Sheet
- Grace CPU Energy Efficiency Blog
- A Demonstration of AI and HPC Applications for NVIDIA Grace CPU [S51880]



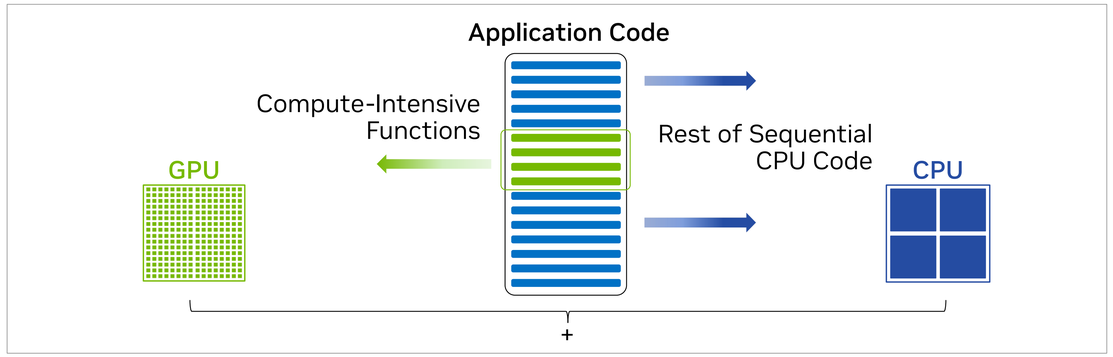
GH200 Grace Hopper Superchip

- GH200 Grace Hopper Customer Deck
- Grace Hopper Superchip Architecture Whitepaper
- Grace Hopper Architecture In-Depth Blog
- Grace Hopper Superchip Architecture Data Sheet
- Grace Hopper Recommender System Blog
- DGX GH200 Tech Blog
- DGX GH200 Data Sheet
- DGX GH200 Whitepaper
- Programming Model and Applications for the Grace Hopper Superchip [S51120]
- Accelerating HPC applications with ISO C++ on Grace Hopper [S51054]



Why Accelerated Computing is Energy Efficient

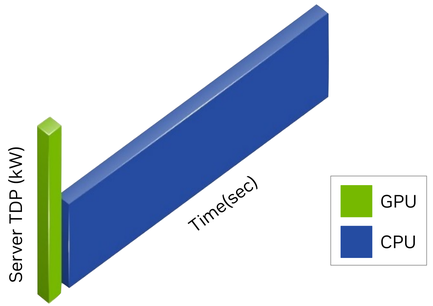
Hardware, Software, & Networking to Optimize Performance & Efficiency



Moving Compute Intense Sections to the GPU

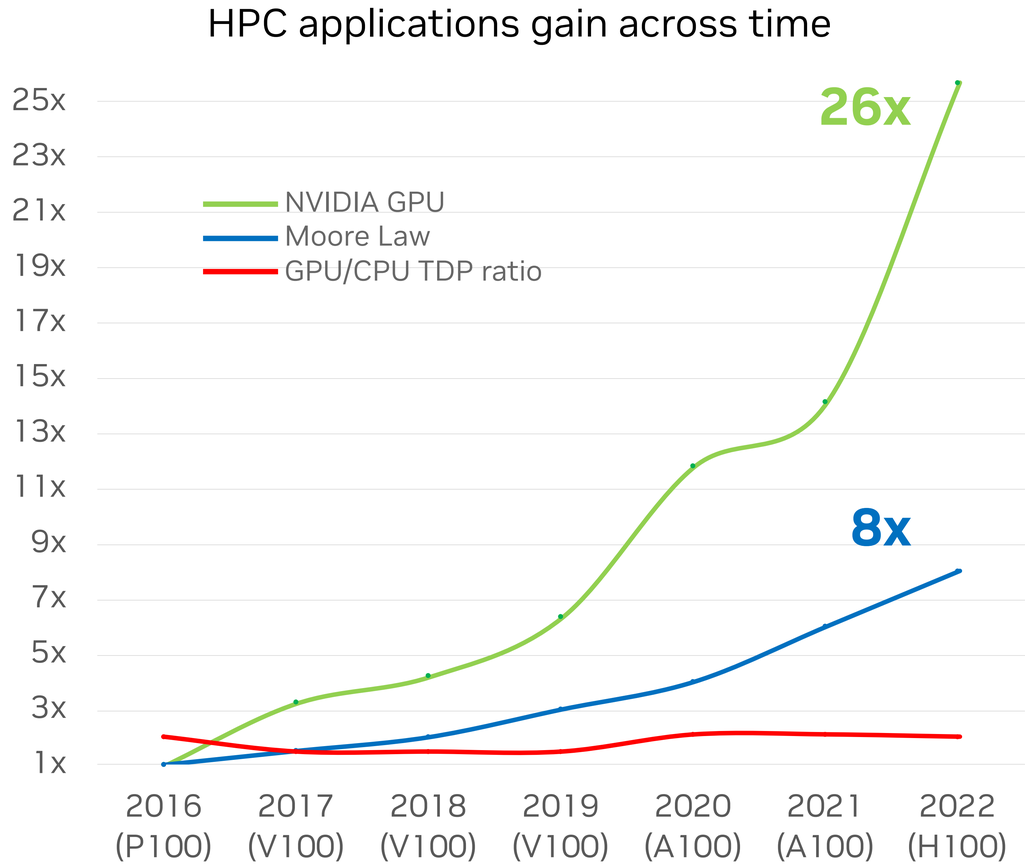
GPUs typically have **higher peak active power consumption** than CPUs

However, they complete workloads **significantly faster**



Energy Used in a Typical AI Inference Job

Consume Less Overall Energy



Center Panel: Geometric mean of application speedups vs. P100 | benchmark applications | Amber [PME-Cellulose_NVE], Chroma [HMC], GROMACS [ADH Dodec], MILC [Apex Medium], NAMD [stmv_nve_cuda], PyTorch (BERT Large Fine Tuner), Quantum Espresso [AUSURF112-jR]; TensorFlow [ResNet-50], VASP 6 [Si Huge], |GPU node: with dual-socket CPUs with 4x P100, V100, or A100 GPUs. H100 values shown for 2022 projected performance subject to change