Developing for NVIDIA Superchips

Dr. John Linford, Principal Technical Product Manager

jlinford@nvidia.com

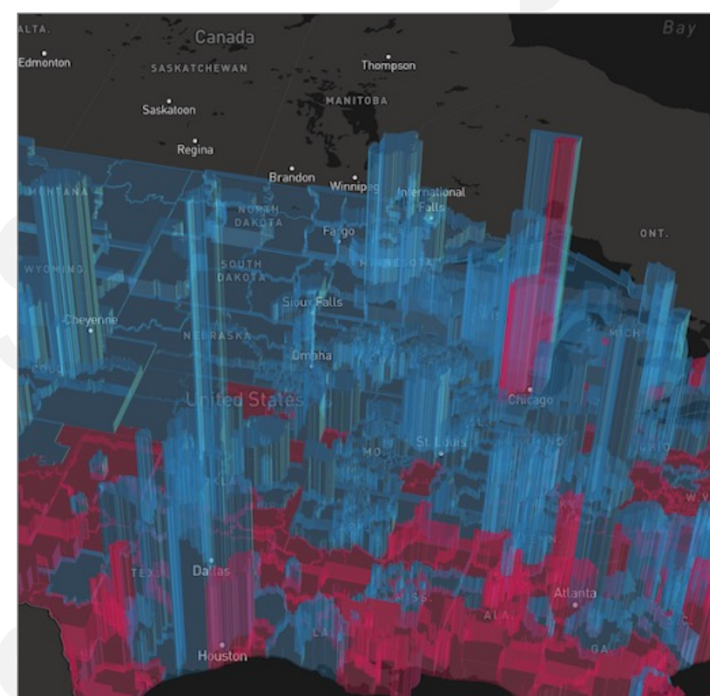
Agenda

- Grace Hopper and Grace CPU Headlines in HPC
- Introduction to NVIDIA Superchips
- Programming Grace Hopper and Grace CPU Superchip
- Porting and Optimizing for Grace CPU
- Optimizing for Grace Hopper / Blackwell Coherent Memory

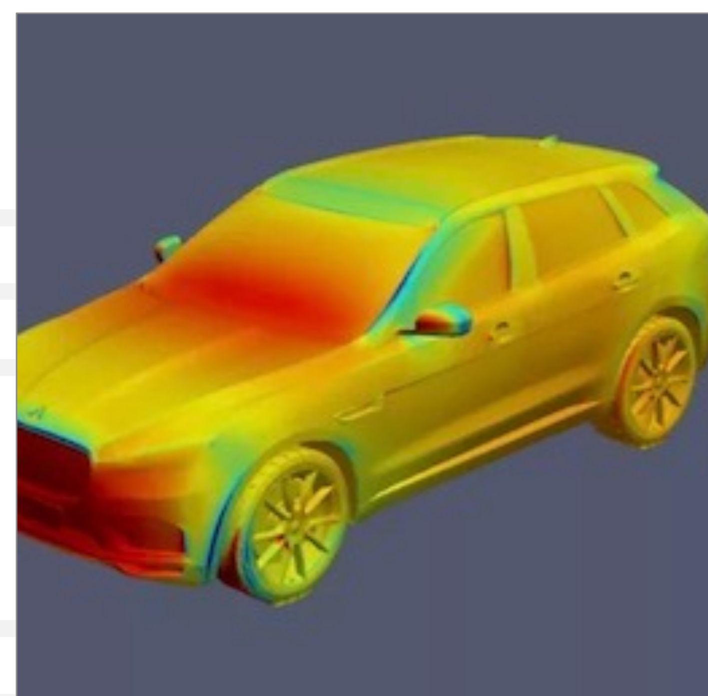
Stony Brook University

NVIDIA AI Accelerated Computing Platform

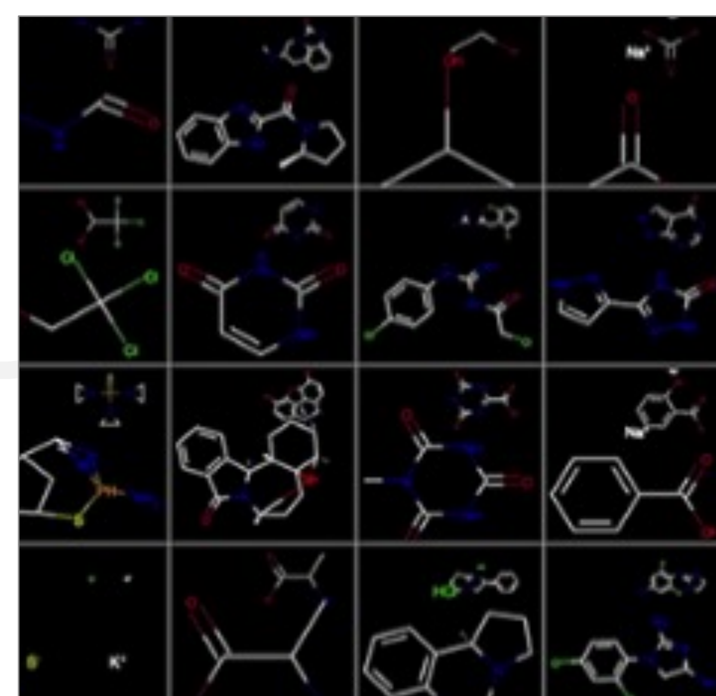
Hardware and Software Acceleration Across Every Workload and Vertical



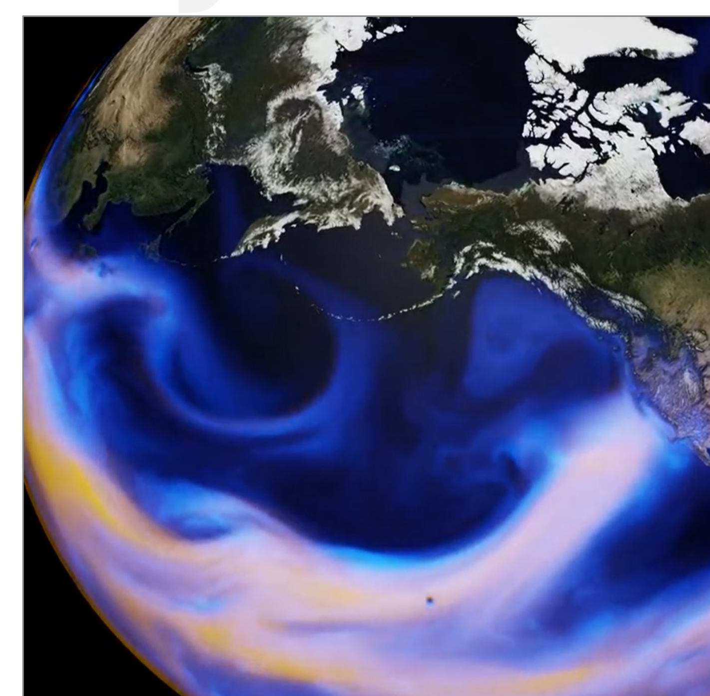
Data Processing



CAD, CAE, SDA

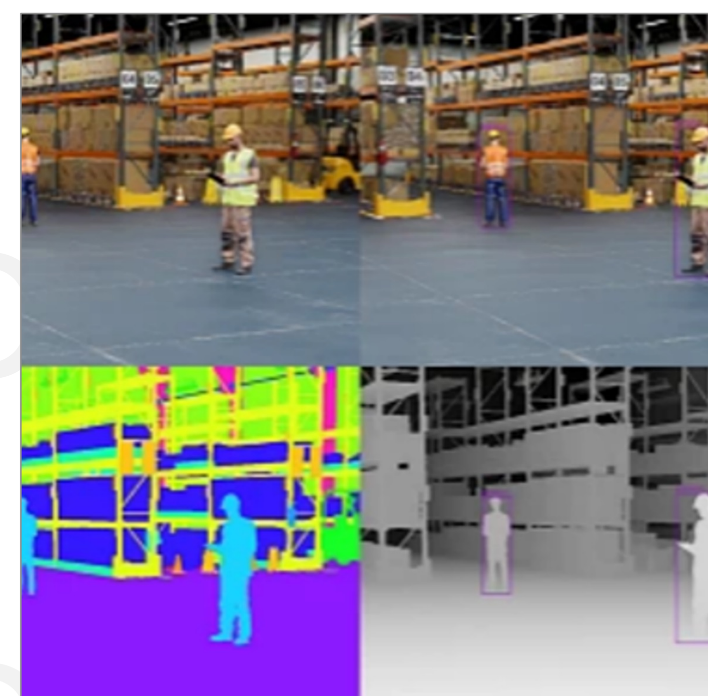


Computer-aided Drug Design

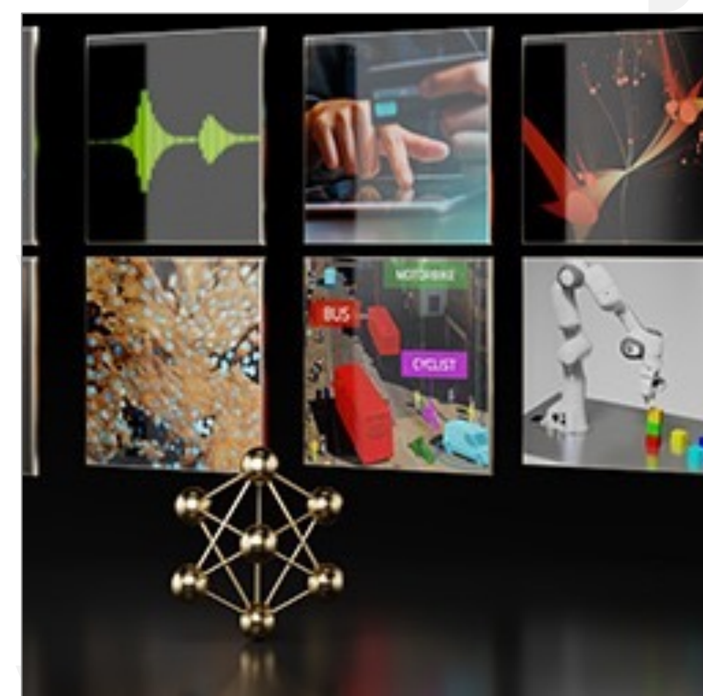


Climate Simulation

Quantum Simulation

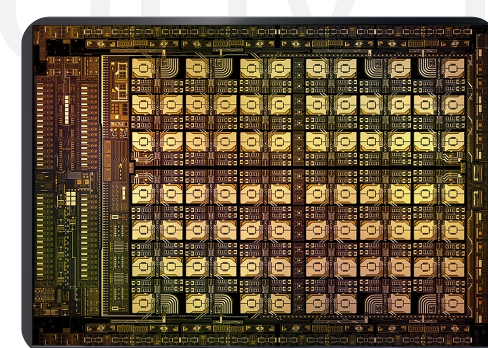


Robotics & Industrial Digital Twins

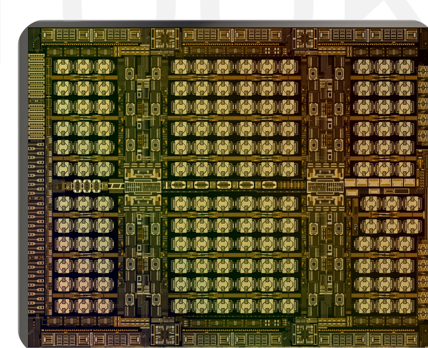


Enterprise AI

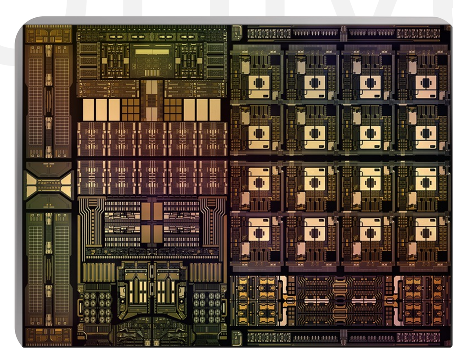
CUDA-X Libraries



CPU

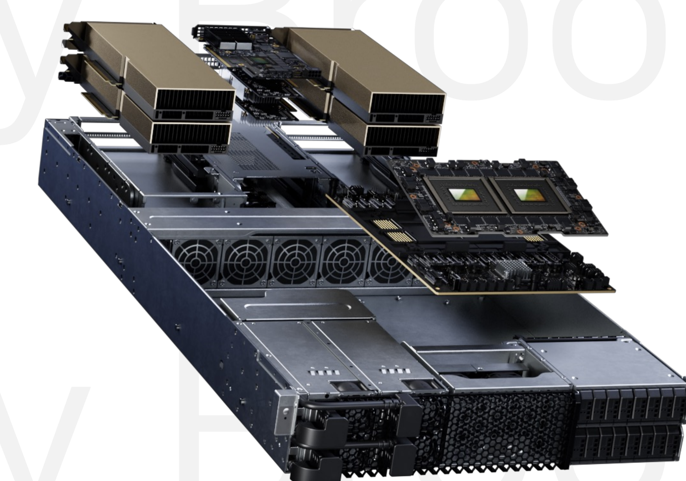


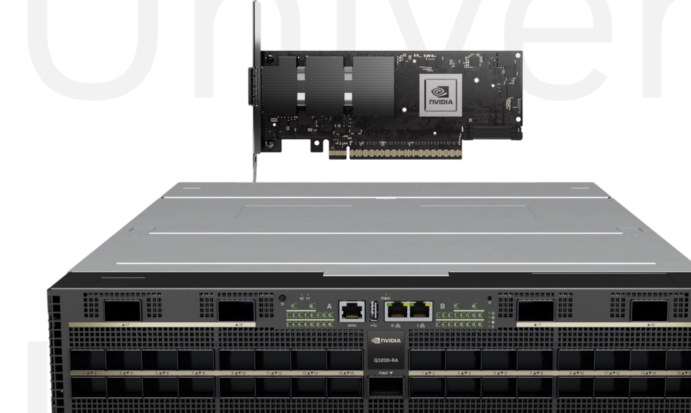
GPU



DPU

Accelerated Computing





Stony Brook University

High Performance, Energy-Efficient Systems in TOP500/GREEN500

Grace Hopper is powering the next wave of new Exaflop AI systems around the globe

Powering Worlds Fastest, Greenest Supercomputers

TOP500
The platform of choice

76%
use NVIDIA

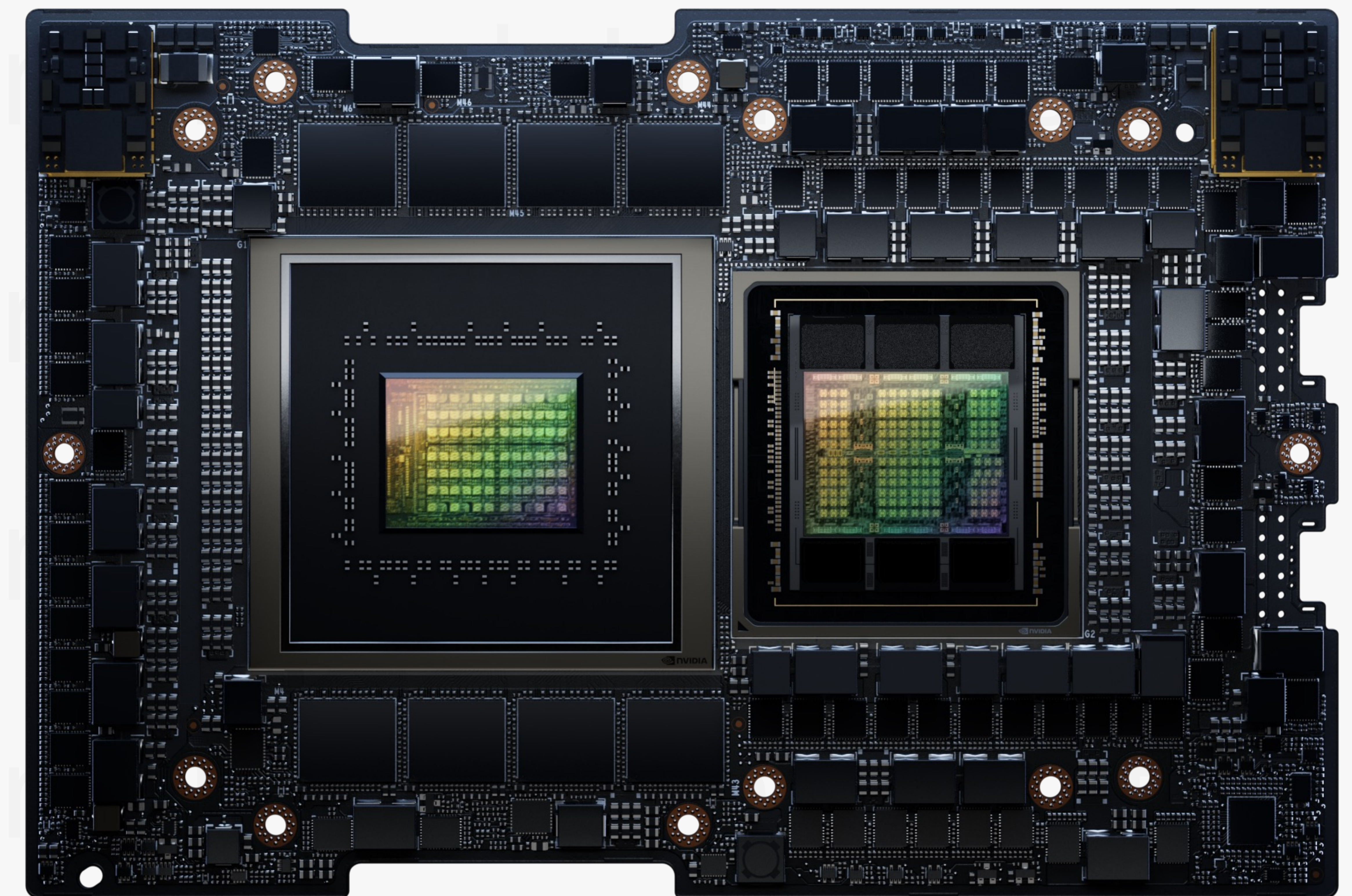
1.5 EF
added to top10

#1
powered by GH200

5 of the top10 powered by GH200

23
NVIDIA of the top 30

Green500
Energy Efficiency



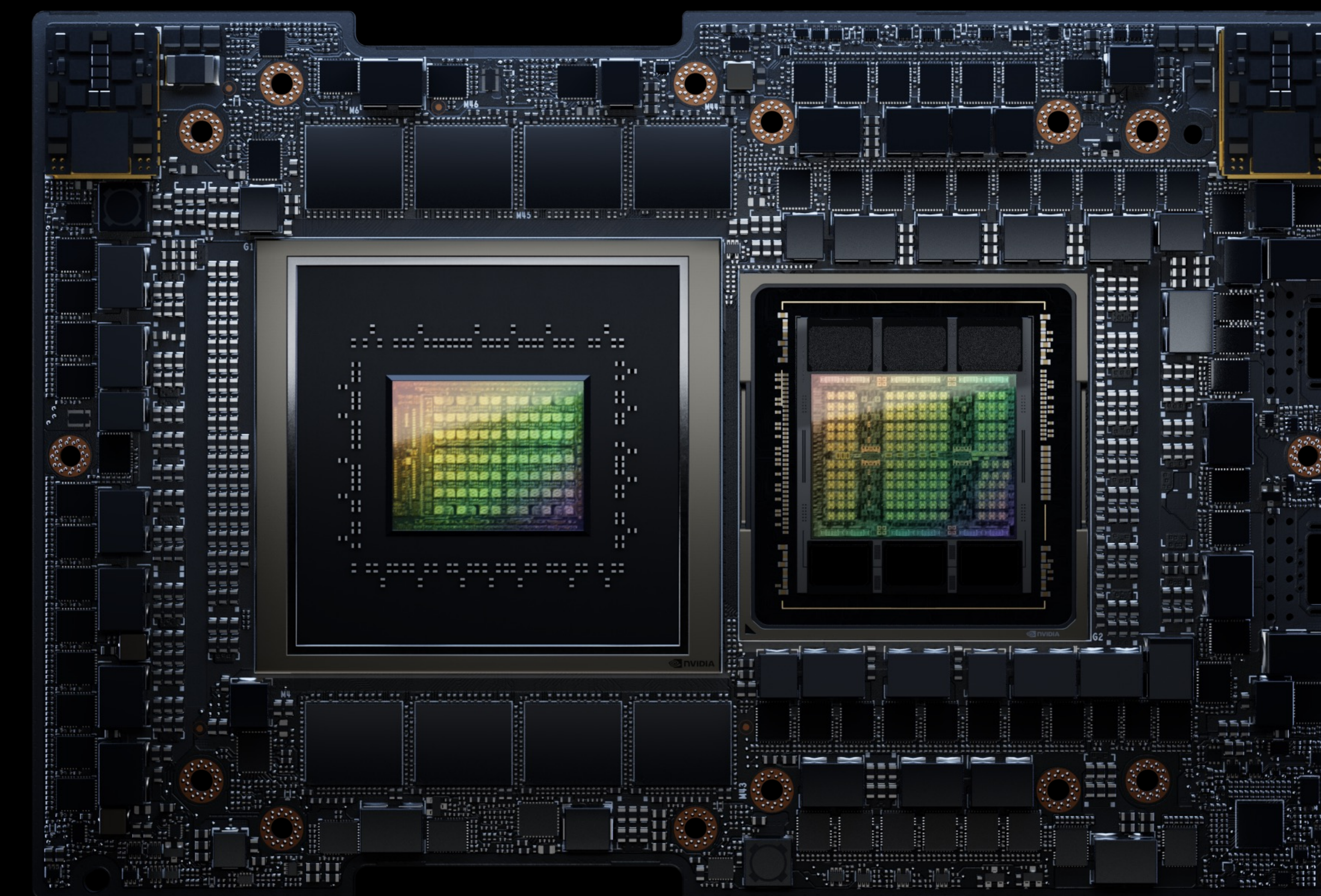
Stony Brook University

NVIDIA GH200 Grace Hopper Superchip

Built for the New Era of AI Supercomputing

CPU to GPU Bandwidth	GPU Memory Bandwidth
900GB/s	**5TB/s**
NVLink-C2C	HBM3e

Energy Efficiency	QFT Quantum Simulation	LLM Inference
50X	**90X**	**200X**
MILC Efficiency vs 2S x86 CPUs	Performance vs 2S x86 CPUs	Performance vs H100 80GB



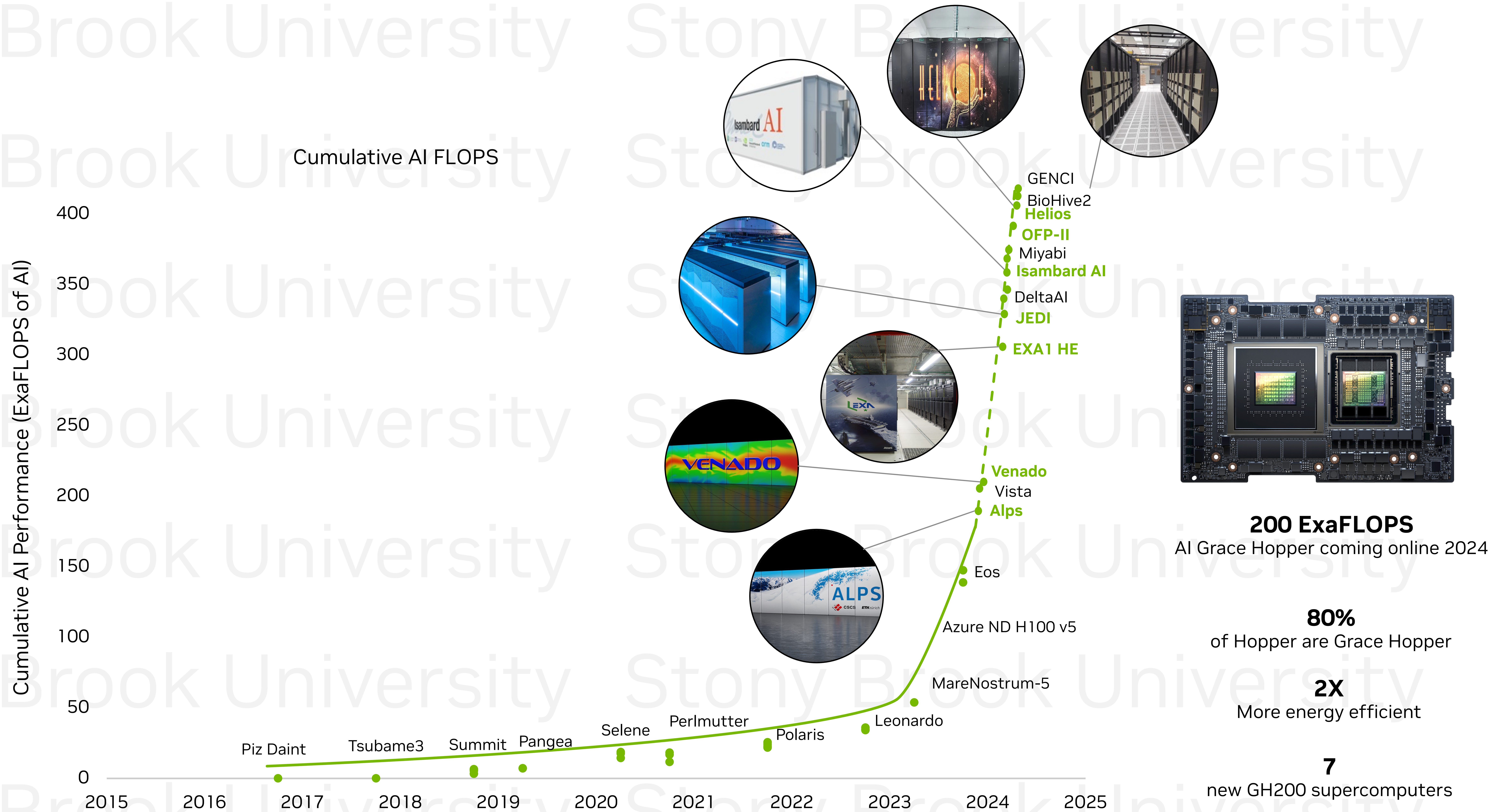
624GB High-Speed Memory | 4 PF AI Perf | 72 Arm Cores

Preliminary measured performance, subject to change
Energy Efficiency: GH200 144GB vs 2S Xeon 8480+ CPU for MILC running dataset: Apex Medium
QFT Quantum Simulation: QFT 2S Xeon 8480+ vs GH200 144GB
LLM Inference: Llama.cpp (2S 8480+) and TensorRT-LLM (GH200, H100) | Max Batch Llama2 70B | Throughput includes time to first token + token generation time

Stony Brook University

Grace Hopper Powers AI Supercomputing Datacenters

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



200 ExaFLOPS
AI Grace Hopper coming online 2024

80%
of Hopper are Grace Hopper

2X
More energy efficient

7
new GH200 supercomputers

NVIDIA

Stony Brook University

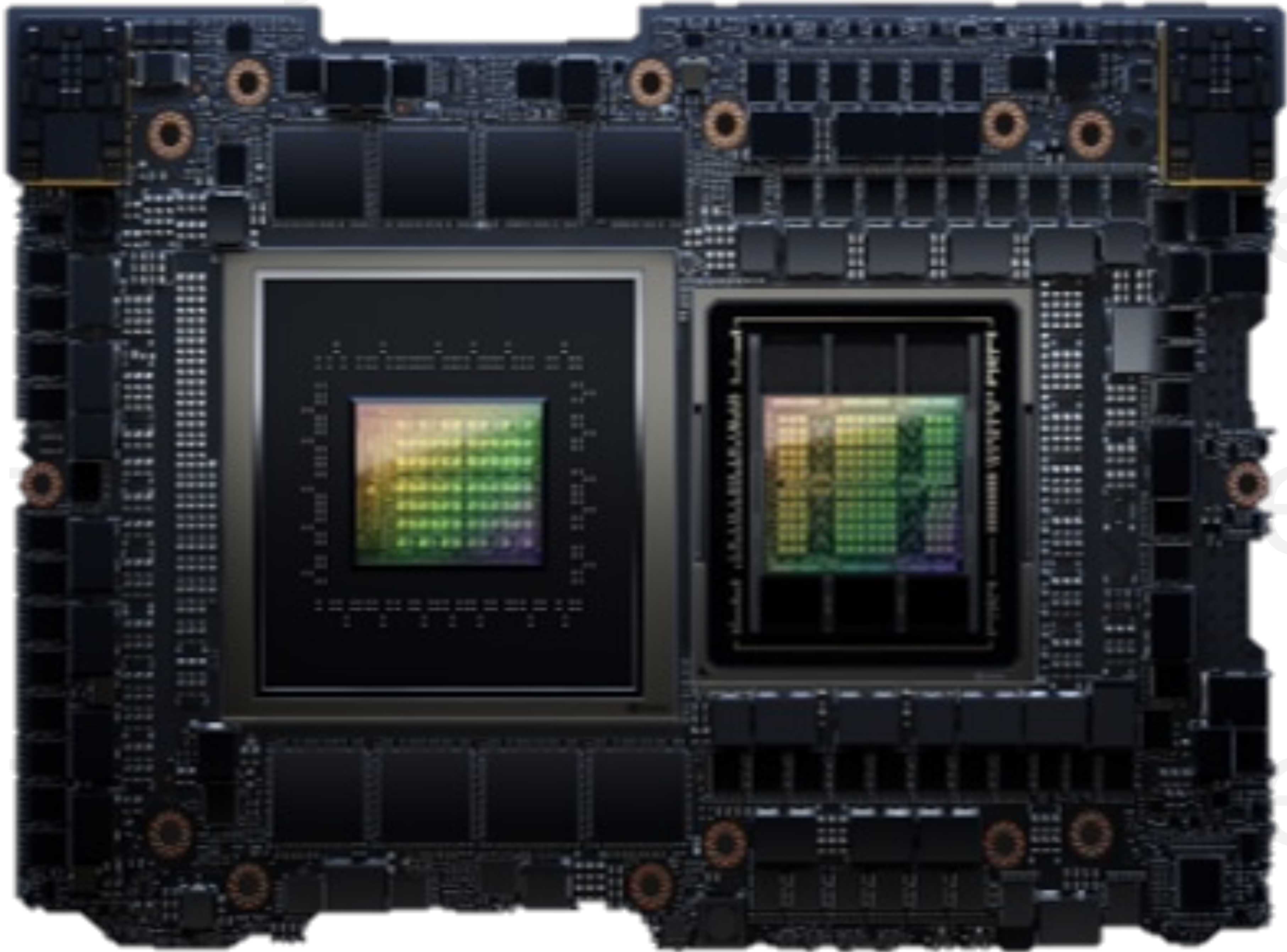
First European Grace Hopper Supercomputer Online

- Fastest AI Supercomputer in Europe
- 20 Exaflops of AI
- 10X more energy efficient than Piz Daint
- Powered by 10,000 Grace Hopper Superchips
- HPC and AI to Advance Weather, Climate (1km global models), and Material Science

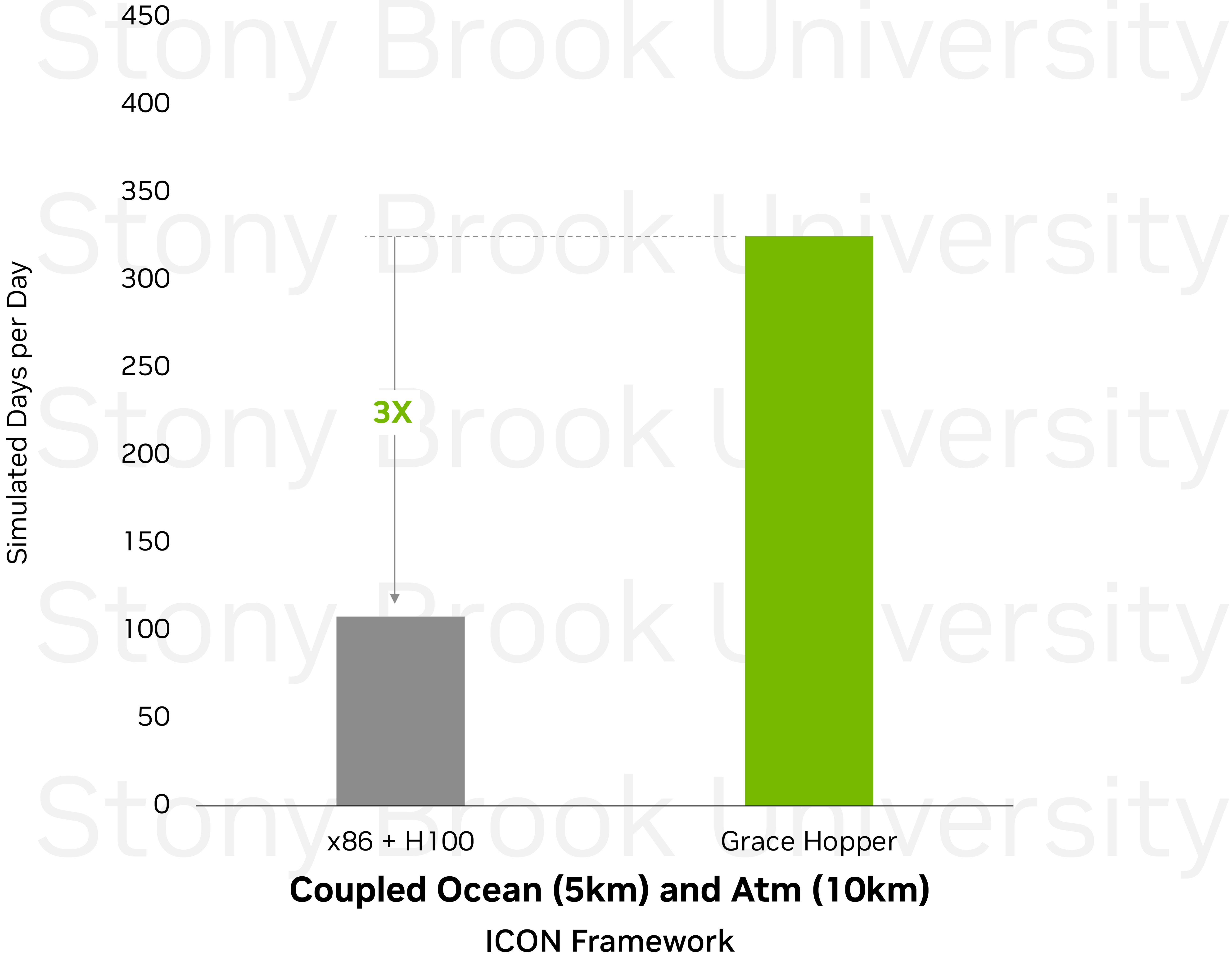
Stony Brook University

CSCS Alps Climate Science Results on Grace Hopper

Up to 4.5X more performance for climate science



Alps GH200 has 4X more, faster CPUs accelerating Ocean simulations. Coupled model waits less and runs faster.



Coupled Ocean (5km) and Atm (10km)

ICON Framework

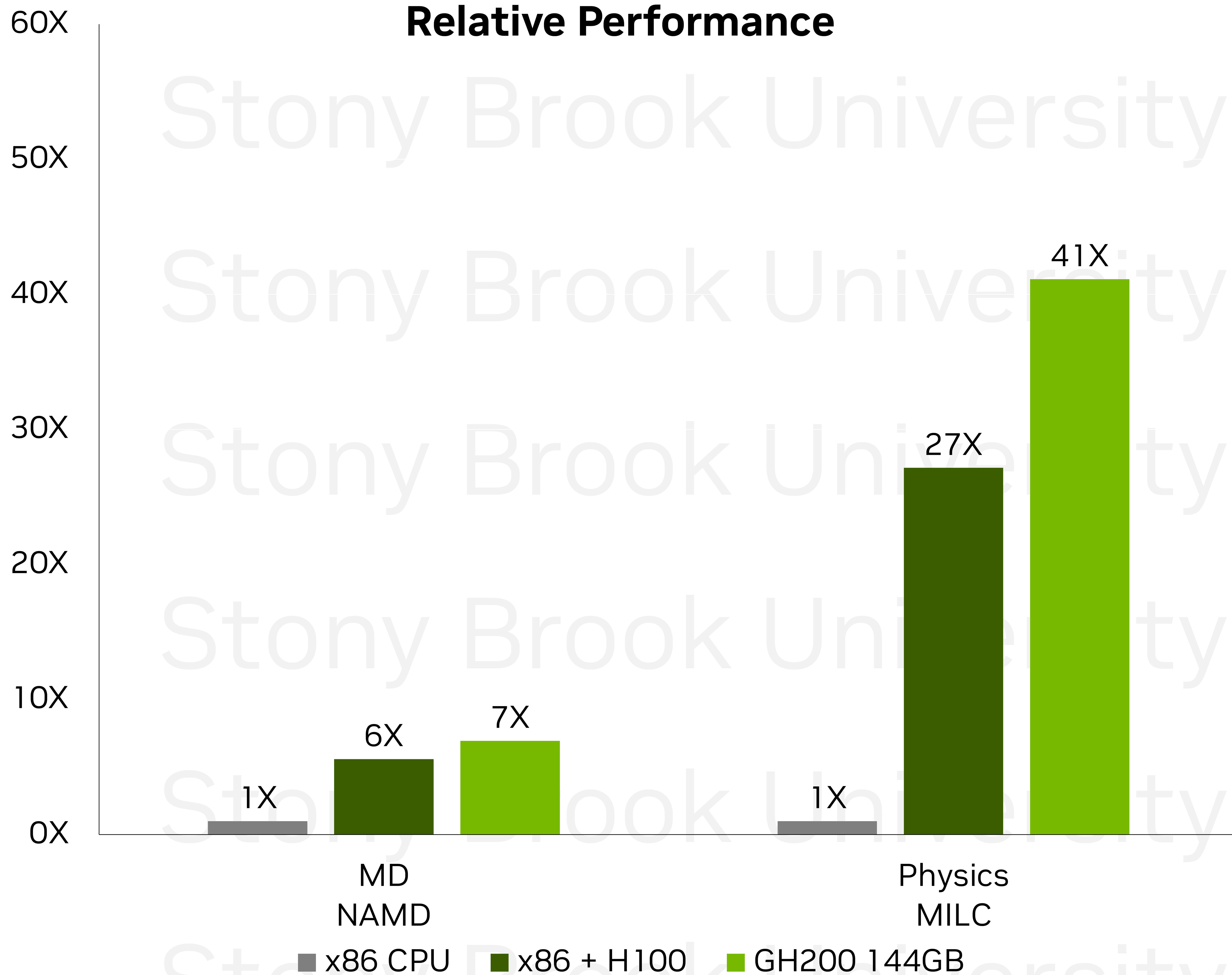
Performance of ICON-openACC (ICON-22 model production at MeteoSwiss) measured on March 18. 2024 | 64 GPUs in each case | DGX-H100 80 GB SXM4 | GH200 CG4 96 GB per GPU with power cap 560-660W per GPU & 128 GB LPDDR coupled atmosphere at R2B8 with ocean at R2B9 resolution (10 km atm + 5km ocean). Atm timestep 90 s, ocean step 5 minutes, coupling step 15 minutes. 90 atm levels, 72 ocean levels.
ICON is a flexible, scalable, high-performance modeling framework for weather, climate and environmental prediction. It provides actionable information for society and advances our understanding of the Earth's climate system.

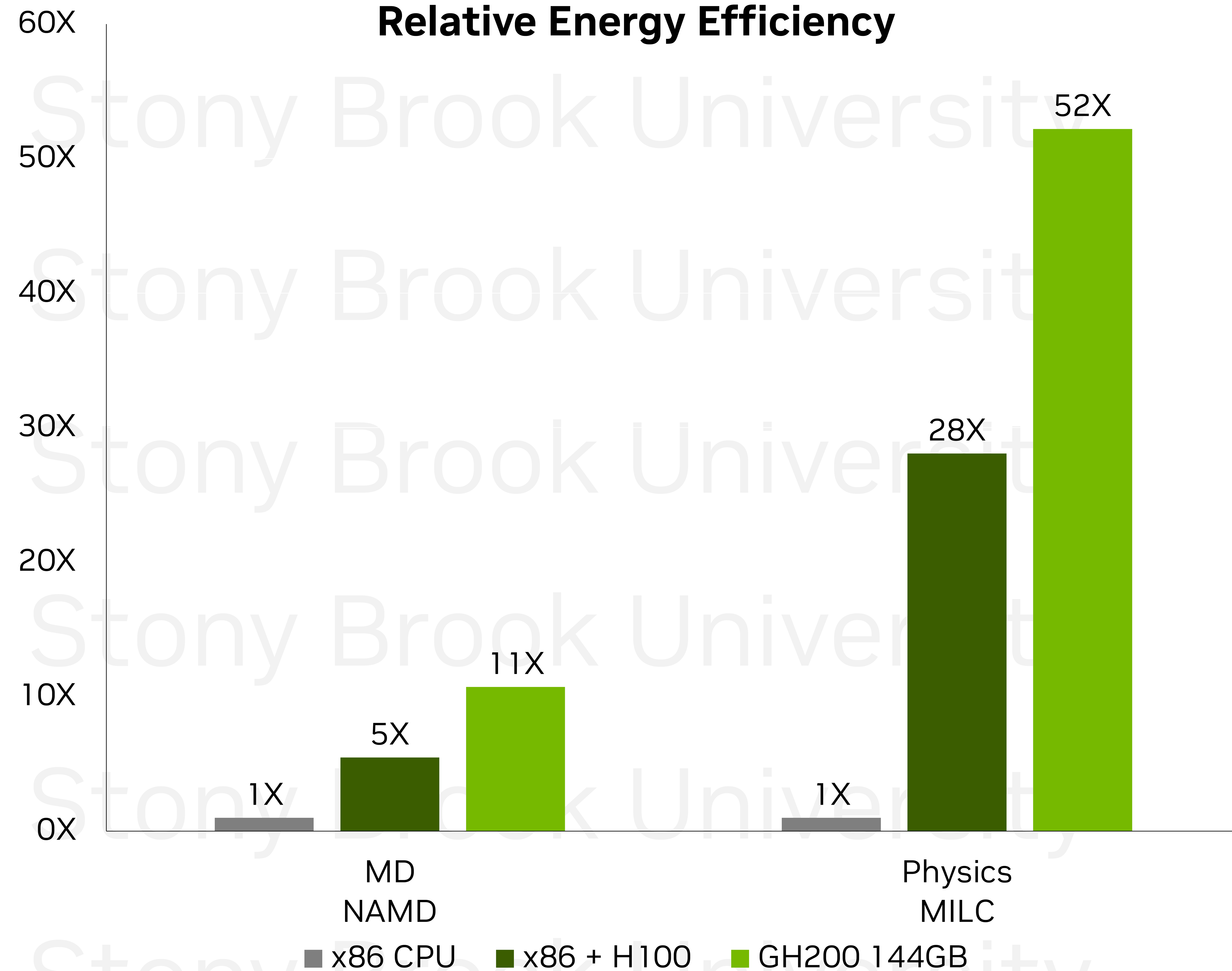
NVIDIA

Stony Brook University

NVIDIA GH200 Delivers Breakthrough HPC Performance

Up to 50X more energy efficient





Single Node GH200 vs 2S Xeon Platinum 8480+ and CPU + Memory + GPU power
Physics: MILC Apex Medium | MD: NAMD LaINDY_ColVars

NVIDIA

Stony Brook University

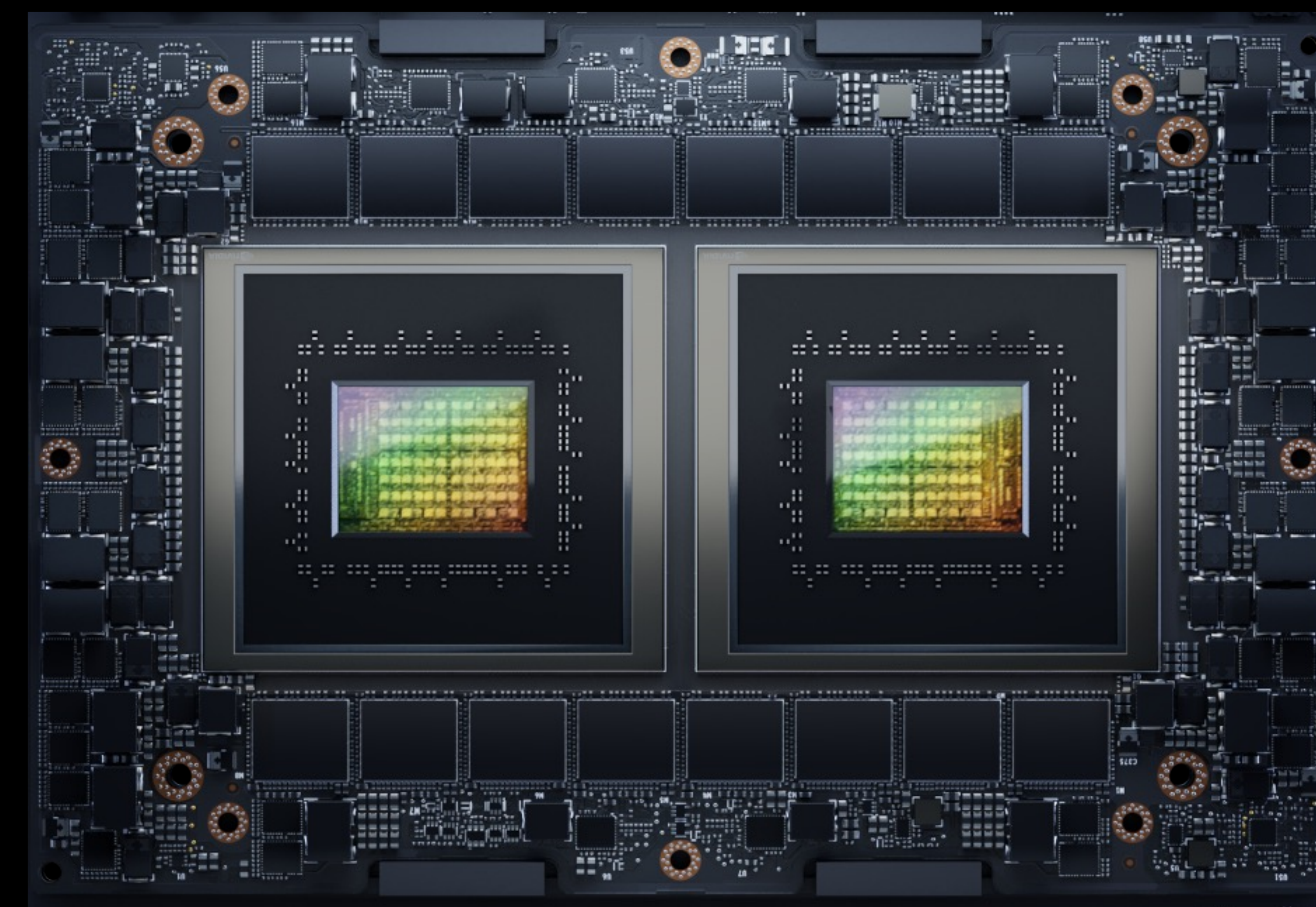
Stony Brook University

NVIDIA Grace CPU Superchip

Breakthrough Performance and Efficiency for the Modern Data Center

CPU + Memory Power	Memory Bandwidth	Green 500
500W	**1 TB/s**	**7** Grace systems
Grace CPU Superchip TDP	LPDDR5X	Performance, Energy Efficiency with GH200

Energy Efficiency	Weather	Graph Analytics
2X	**1.3X**	**2X**
Performance vs x86 CPU	Performance vs x86 CPU	Performance vs x86 CPUs



144 Arm Neoverse V2 Cores | 234MB L3 Cache
3.2 TB/s NVIDIA Scalable Coherency Fabric | 960GB LPDDR5X

Preliminary measured performance, subject to change
Energy Efficiency: Grace CPU Superchip vs 2S AMD EPYC 9654 and Xeon Platinum 8480+. Geomean of OpenFOAM (Motorbike Large), WFR (CONUS12km), ICON (QUBICC 80 km resolution) specfm3d (four_material_simple_model) and Branson (3D_hohlraum_single_node)
Weather: WRF (CONUS12km) Grace CPU Superchip vs 2S AMD EPYC 9654
Graph Analytics: GAP BS Breadth First Search

Isambard 3
Grace CPU Supercomputer

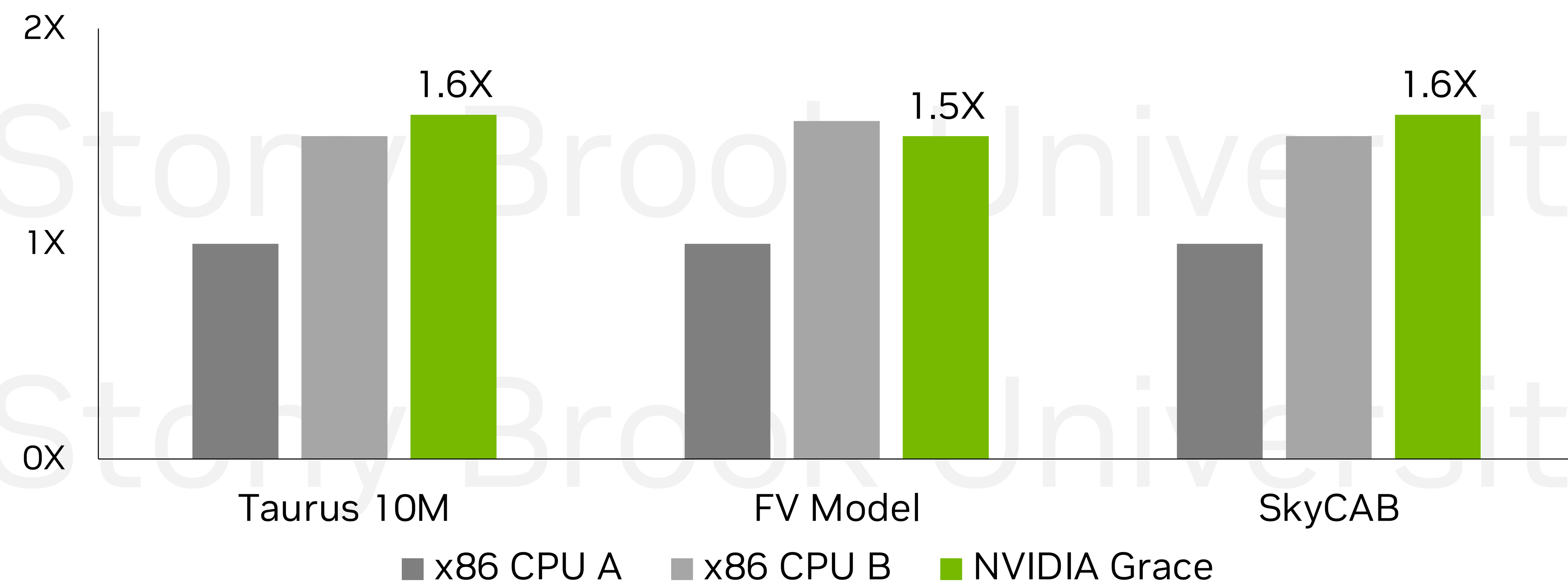
University of Bristol

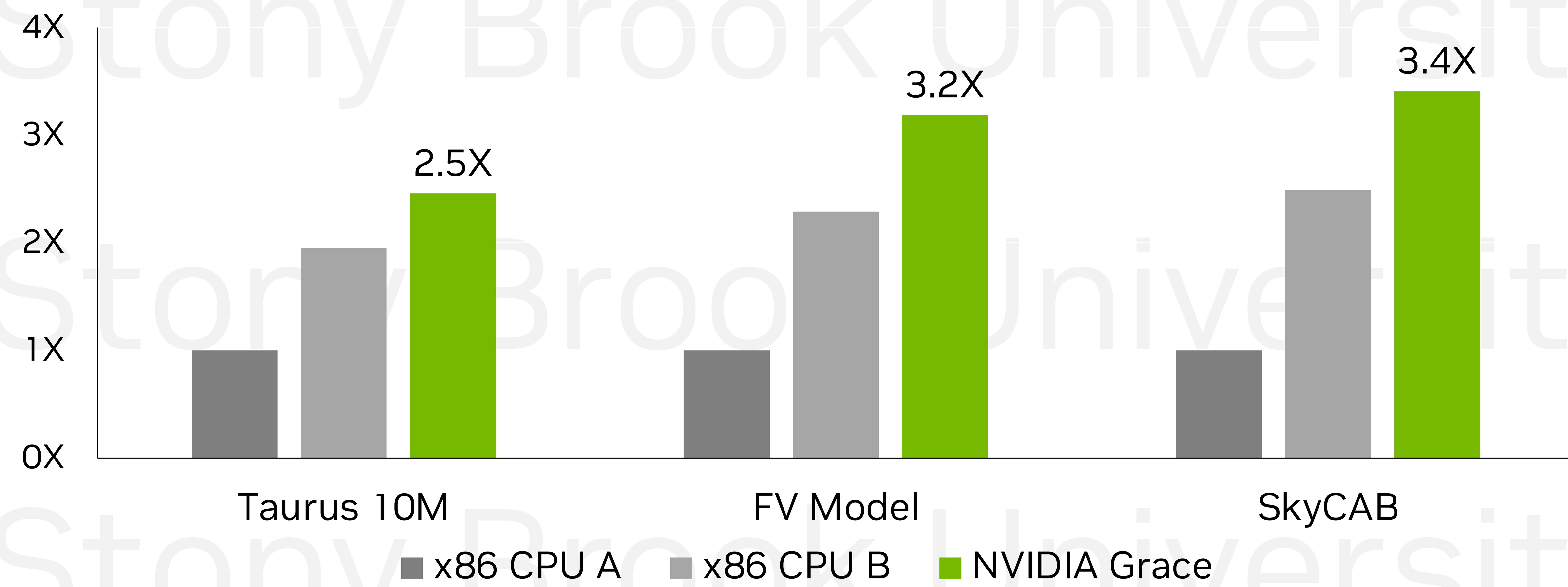
- Over 55,000 Arm Neoverse V2 cores
- 2.7 PF of HPC Performance, 270 kW of power
- 6X more performance and energy efficiency vs. predecessor
- Built by HPE
- Enabling breakthroughs in Climate Science, Drug Discovery, and Industrial HPC.

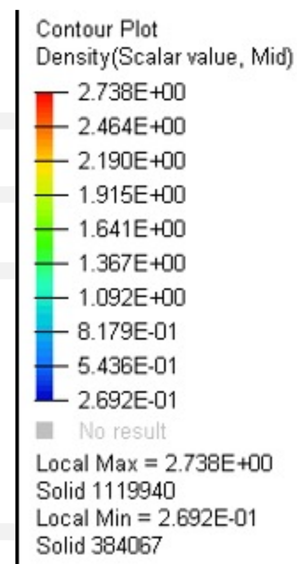
Stony Brook University

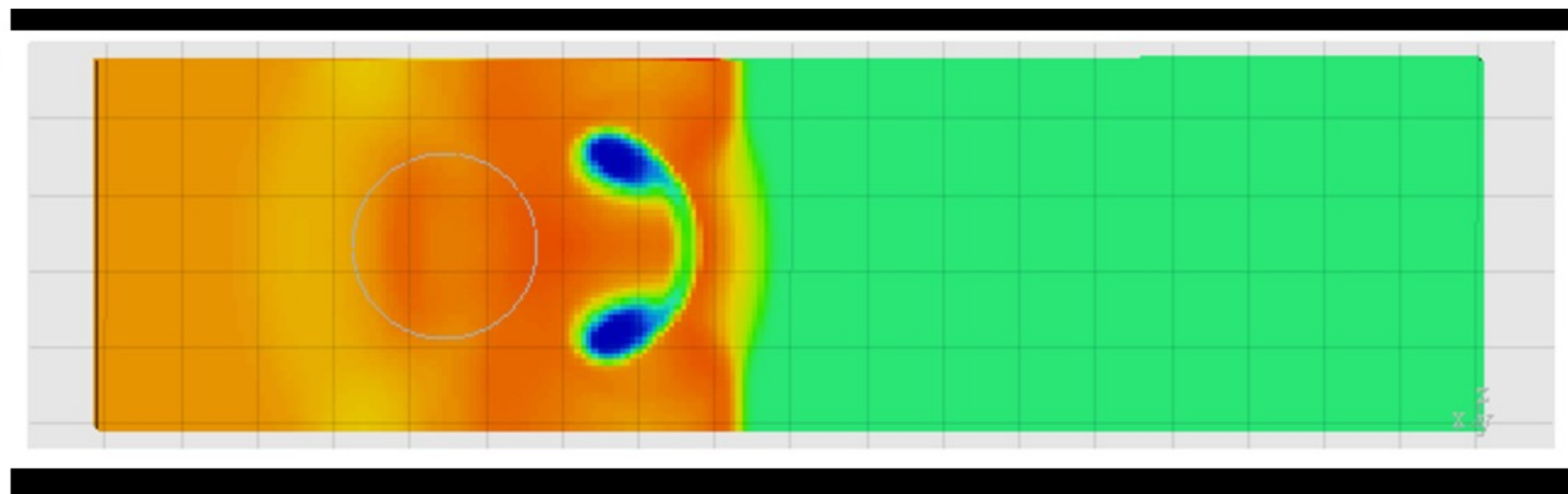
OpenRadioss Crash and Impact

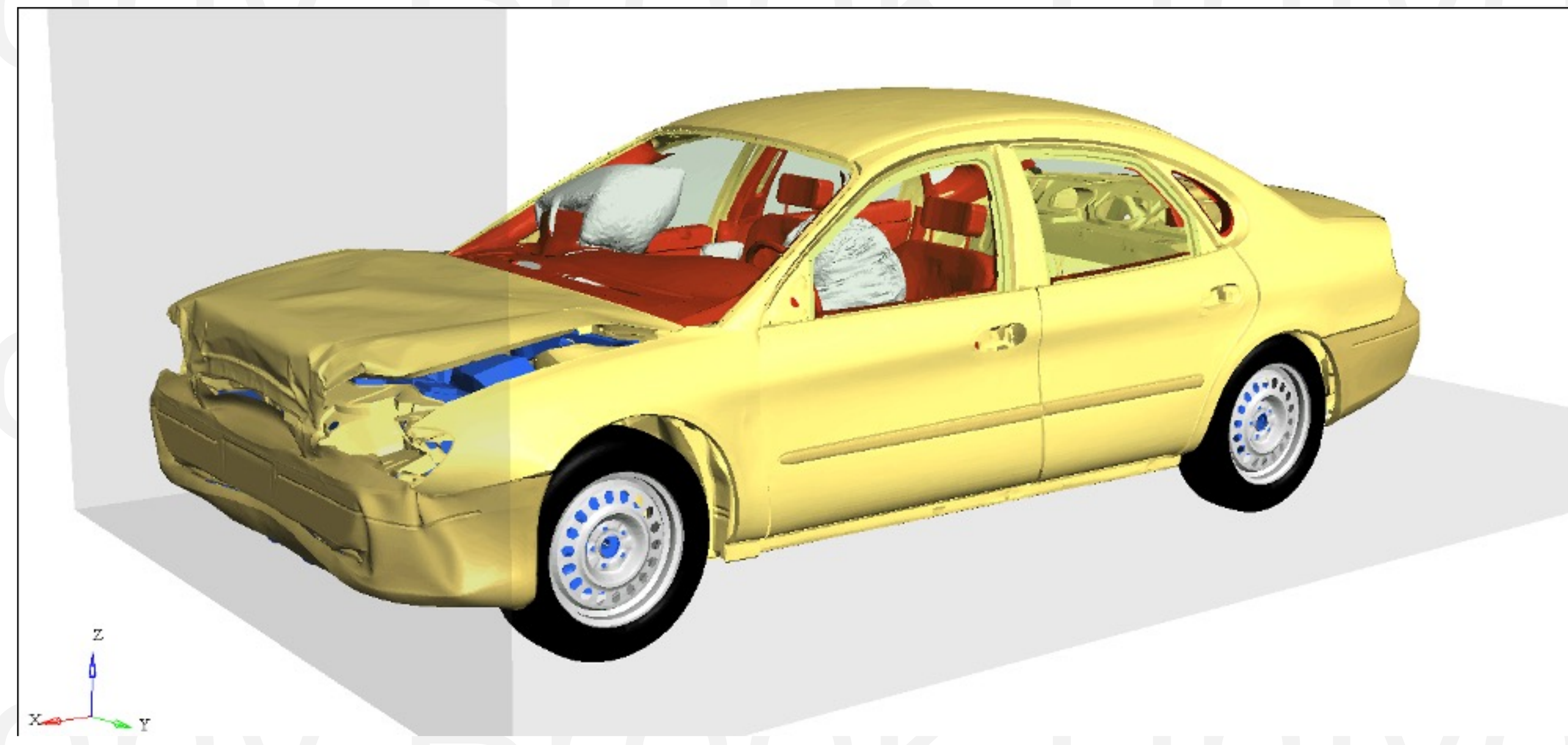
NVIDIA Grace CPU delivers up **3.4X** energy efficiency

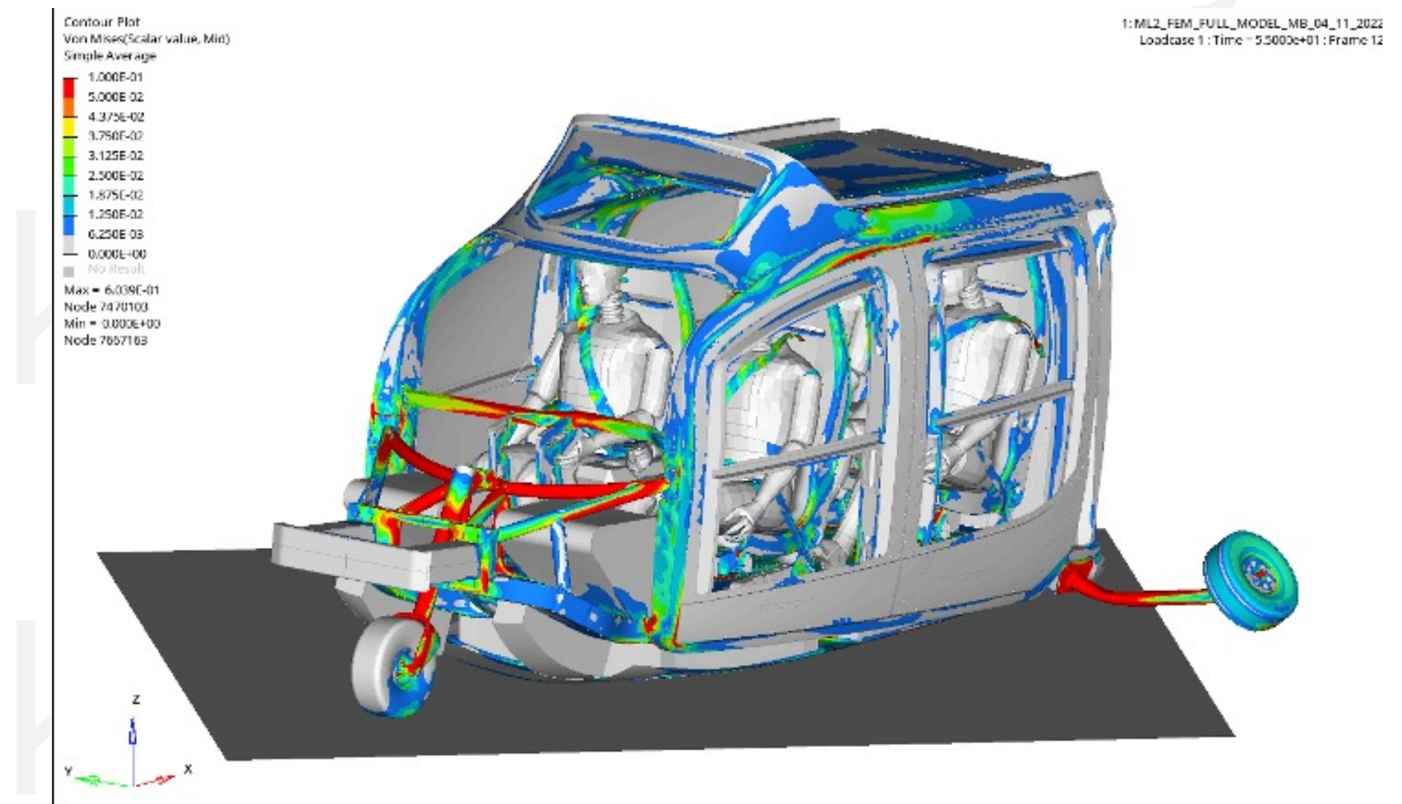


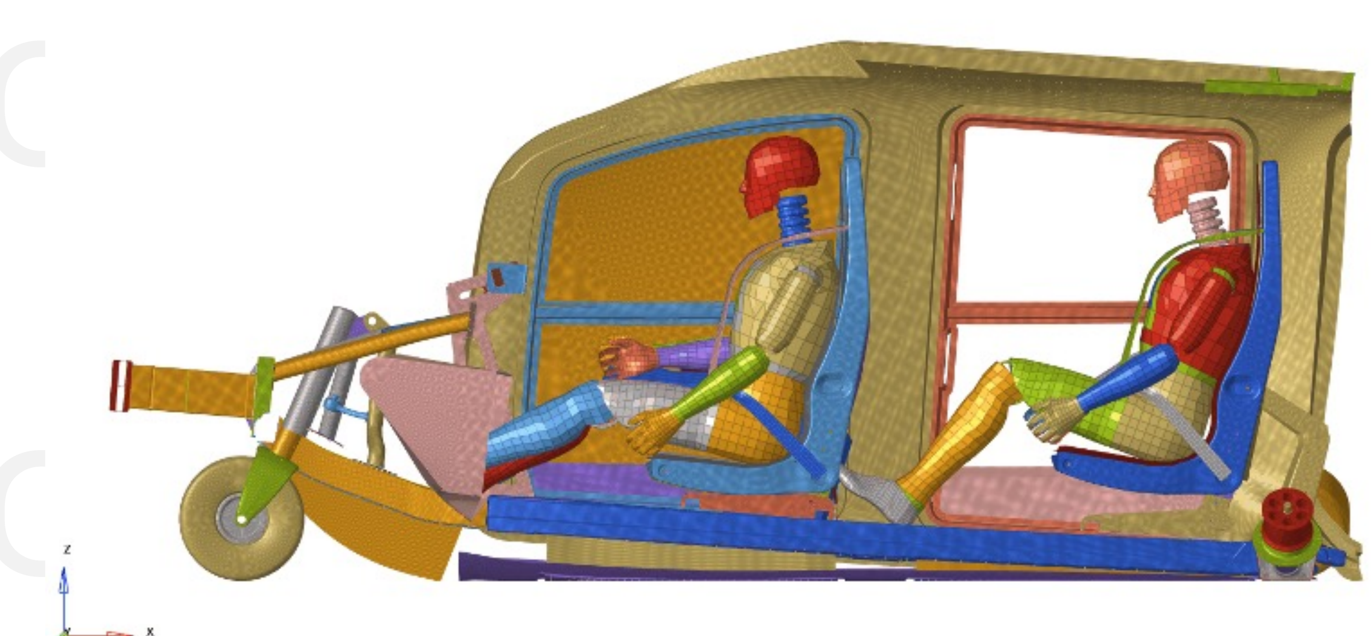












NVIDIA Grace Superchip vs x86 flagship 2-socket data center systems (x86 CPU A: Intel Xeon 8470Q and x86 CPU B: AMD EPYC 9654). Open Radioss Taurus 10M Shells + Solids | SkyCAB FEA Model 5.1M Elements | FV Model – Classic Finite Volume Test Using 15 Million ALE Solids | Results subject to change.

Stony Brook University

NVIDIA Grace Superchips

The NVIDIA Grace CPU

The building block of the superchip

High Performance Power Efficient Cores

72 flagship Arm Neoverse V2 Cores with
SVE2 4x128b SIMD per core
3.5 FP64 TFLOP/s TPeak

Fast On-Chip Fabric

3.2 TB/s of bisection bandwidth connects
CPU cores, NVLink-C2C, memory, and system IO

High-Bandwidth Low-Power Memory

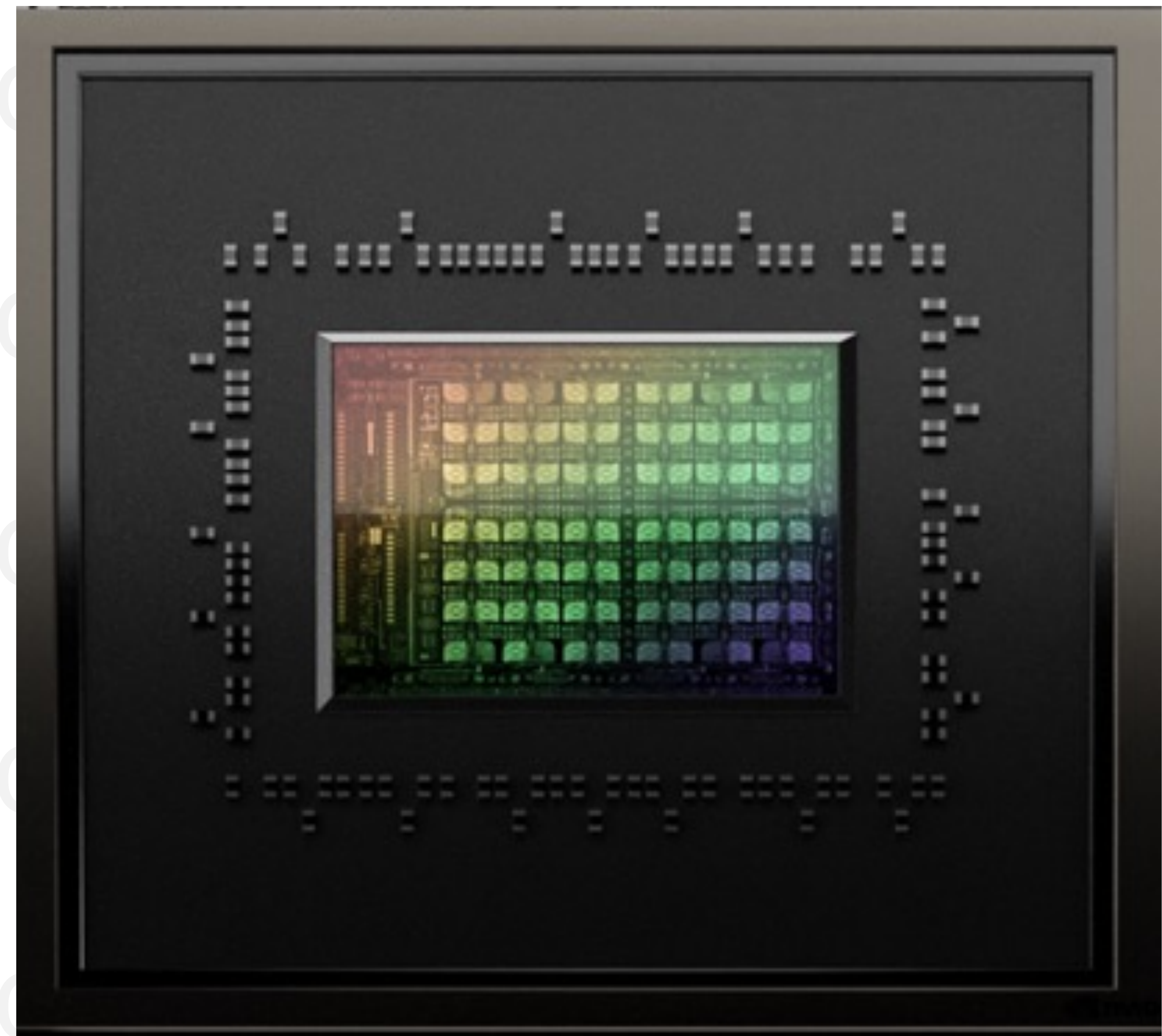
Up to 480 GB of data center enhanced LPDDR5X Memory that
delivers **up to 500 GB/s of memory bandwidth**

Coherent Chip-to-Chip Connections

NVLink-C2C with **900 GB/s bandwidth** for coherent
connection to CPU or GPU

Industry Leading Performance Per Watt

Up to 2X perf / W over today's leading servers

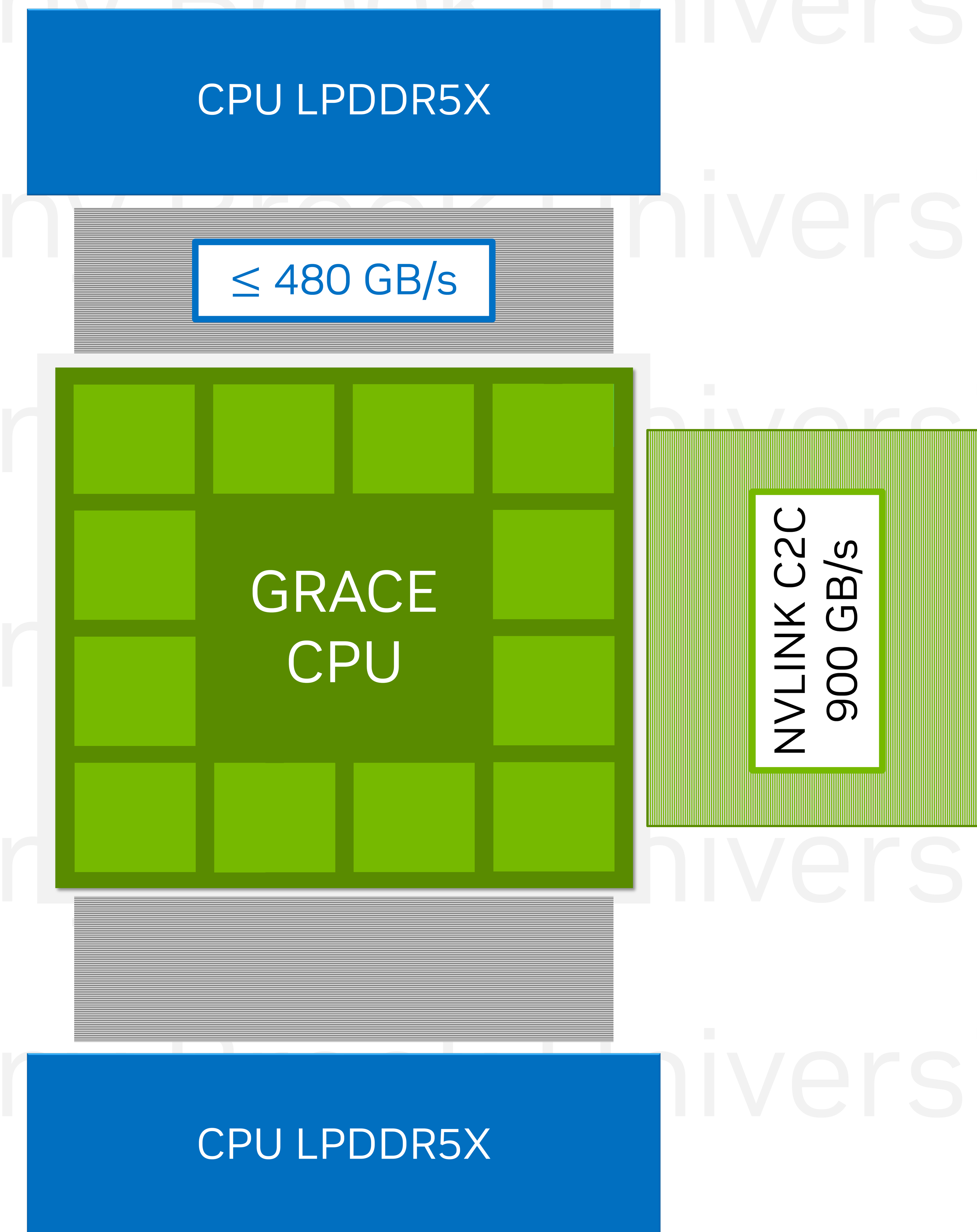


Stony Brook University

NVLINK-C2C

High Speed Chip to Chip Interconnect

- Grace Hopper and Grace Superchips
- Removes the typical cross-socket bottlenecks
- Up to 900GB/s of raw bidirectional BW
  - Same BW as GPU to GPU NVLINK on Hopper
- Low power interface - 1.3 pJ/bit
  - More than 5x more power efficient than PCIe
- Enables coherency for both Grace and Grace Hopper superchips



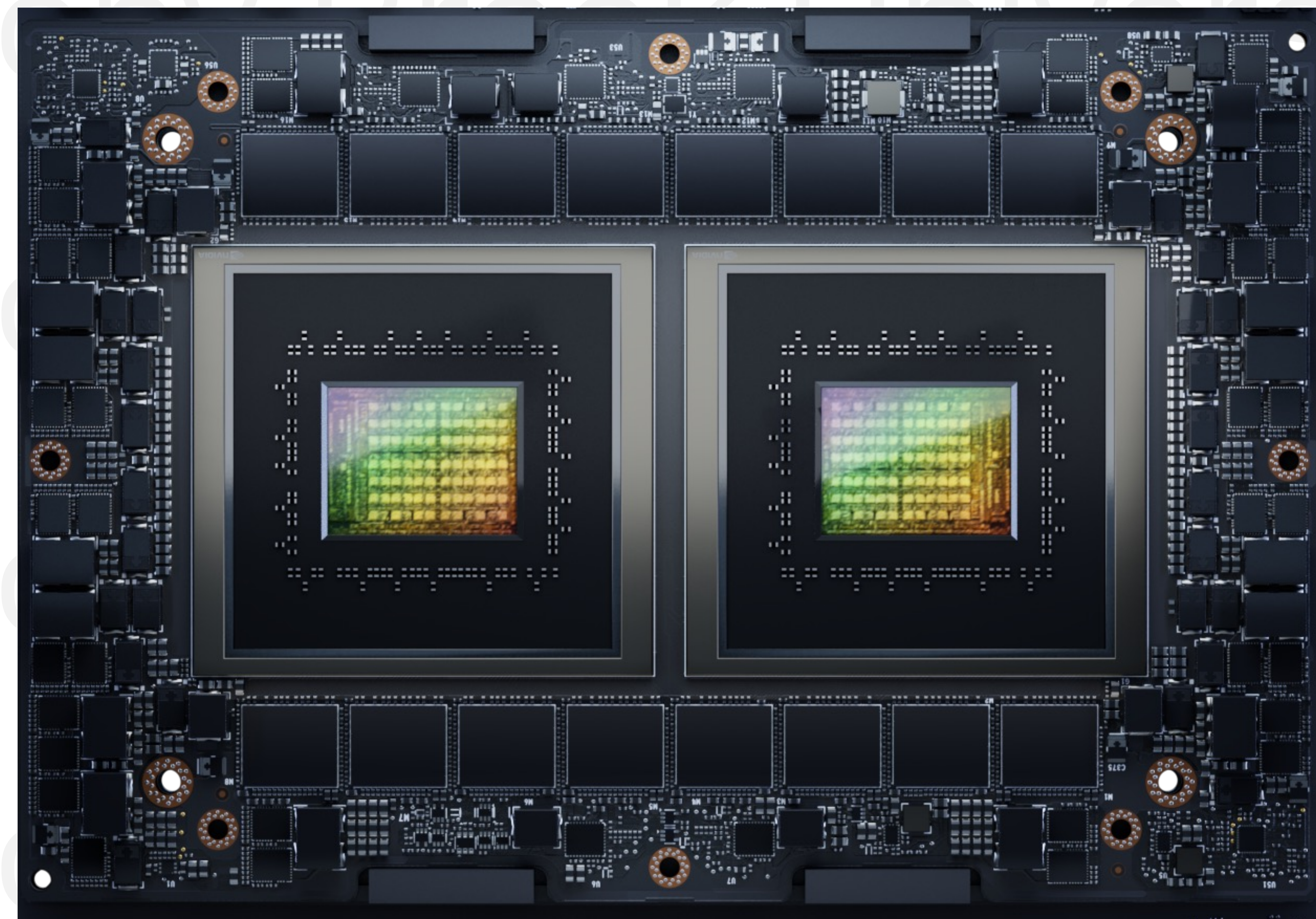
Stony Brook University

NVIDIA

One Powerful CPU – Two Superchip Configurations

Grace CPU Superchip

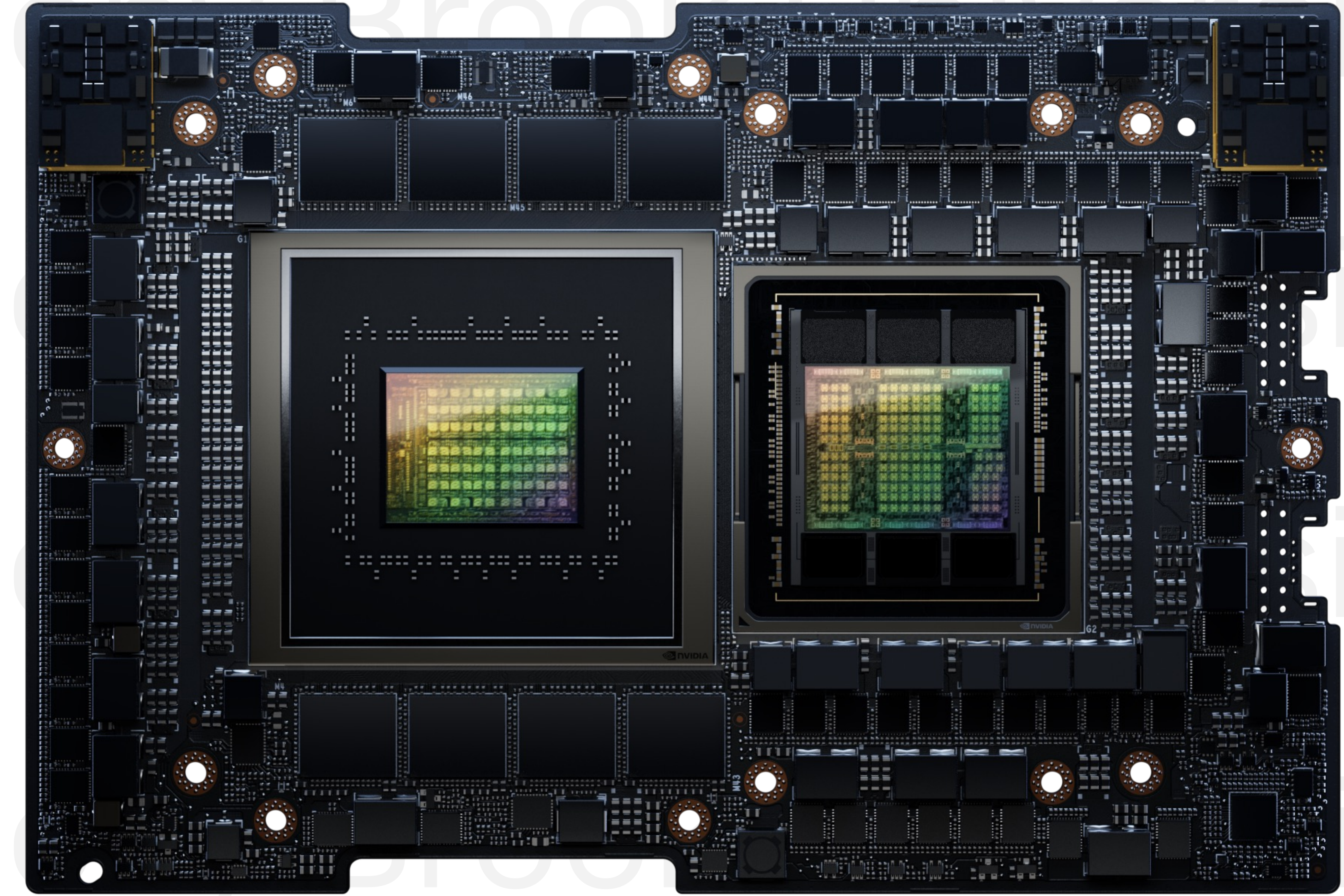
CPU Computing



More than "2x Grace"

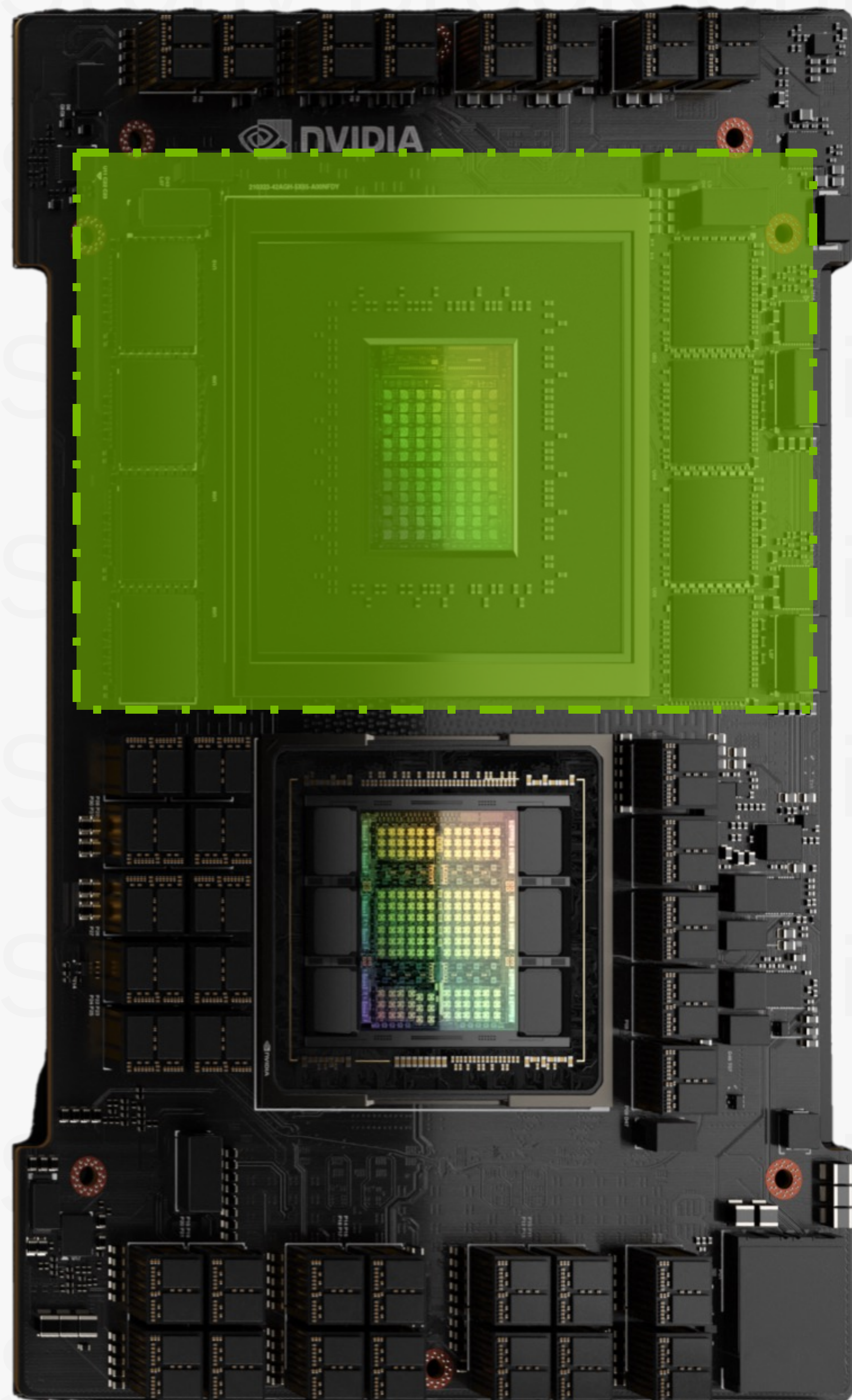
GH200 Grace Hopper Superchip

Large Scale AI & HPC



More than "Grace + Hopper"

Stony Brook University



NVIDIA Grace Hopper Superchip

"Super" → more than a "chip"
NVIDIA CPU + NVIDIA GPU w/o compromises

- **NVIDIA Grace CPU + LPDDR5 Memory**
  - 72 Arm-v9 Neoverse V2 CPU cores with SVE2.
    - →Efficiency: 62pJ/DFMA (x86: ~99); 1.6x more efficient
    - →Performance: 3.6 FP64 TFLOP/s
  - Memory:
    - → High capacity: ≤ **480 GB** LPDDR5X (5pJ/bit vs 36 DDR)
    - → High bandwidth: ≤ **500 GB/s**
    - → Low latency: less than competitors at peak bandwidth

Stony Brook University

NVIDIA Grace Hopper Superchip

"Super" → more than a "chip"
NVIDIA CPU + NVIDIA GPU w/o compromises

- **NVIDIA Grace CPU + LPDDR5 Memory**
  - 72 Arm-v9 Neoverse V2 CPU cores with SVE2.
    - →Efficiency: 62pJ/DFMA (x86: ~99); 1.6x more efficient
    - →Performance: 3.6 FP64 TFLOP/s
  - Memory:
    - → High capacity: ≤ **480 GB** LPDDR5X (5pJ/bit vs 36 DDR)
    - → High bandwidth: ≤ **500 GB/s**
    - → Low latency: less than competitors at peak bandwidth
- **NVIDIA Hopper GPU**
  - →High performance: 60 FP64 TC TFLOP/s
  - Memory:
    - → High capacity: 96 GB HBM3
    - → Extreme bandwidth ≤ 4000 GB/s
  - → Threads are threads (not SIMD lanes)

Stony Brook University

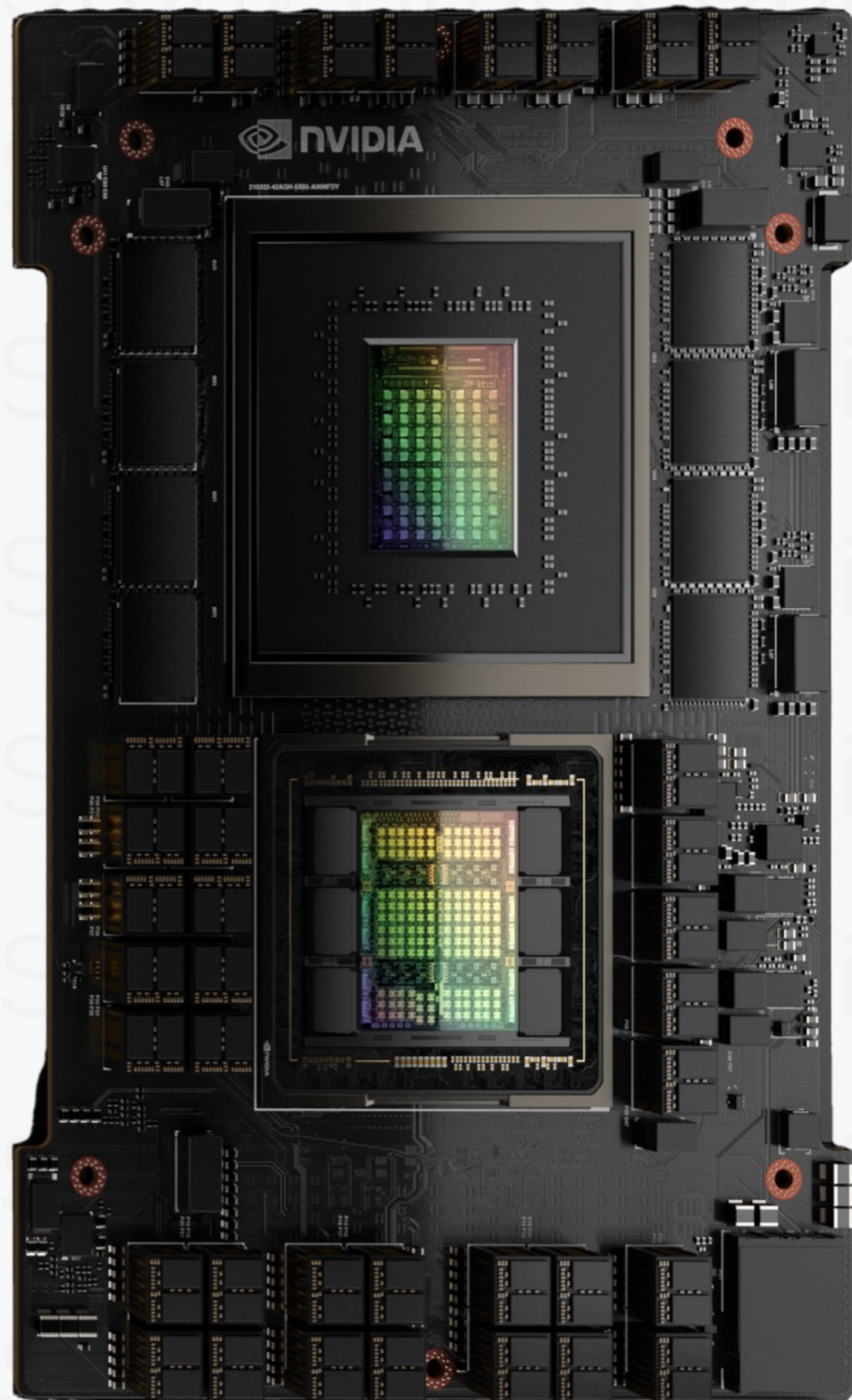
NVIDIA Grace Hopper Superchip

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

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

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

Stony Brook University



NVIDIA Grace Hopper Superchip

Made 💚 for any programming model
Portable ISO C++, ISO Fortran, Python

- **Simplifies parallelization:** less SW changes

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

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

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

Stony Brook University

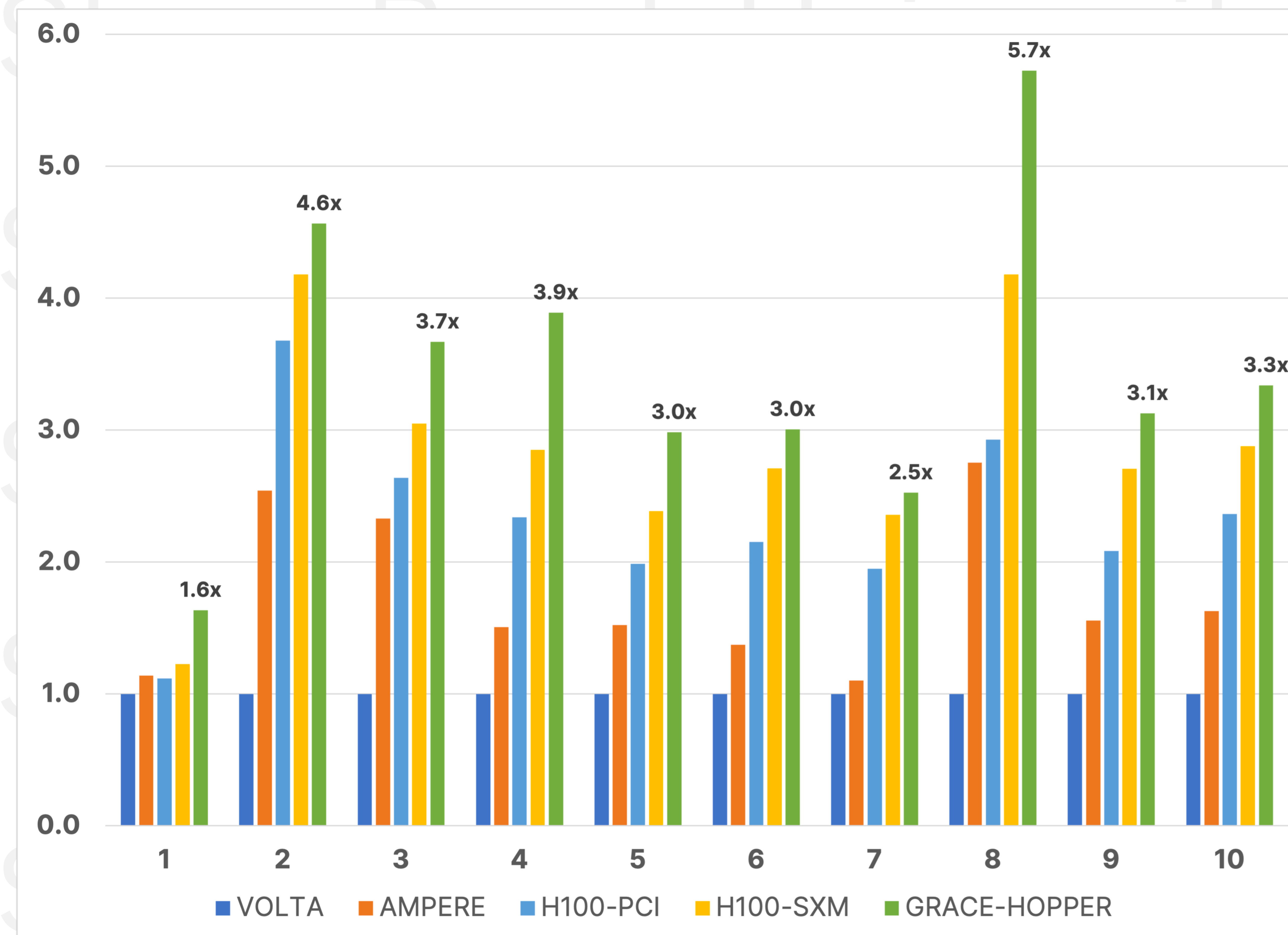
The Grace Hopper Advantage for Developers

- Existing GPU applications require no changes for Grace Hopper
  - No new APIs
  - No restructuring
  - No new programming model
  - Developers who choose to can optimize for the Grace Hopper platform
- Existing GPU applications (fully or partially ported) will run better on Grace Hopper
  - Data migration no longer required, may still be a performance optimization
  - When data migrations happen, they happen faster due to C2C interconnect
  - CPU code will benefit from higher bandwidth memory, high thread performance, coherent accesses
  - Existing, stable Unified Memory APIs may be used for performance optimization
- Non-GPU applications will run unmodified and benefit from Grace architecture
- Porting from CPU to GPU is made simpler by Grace Hopper
  - Coherent Memory Subsystem
  - C2C interconnect
  - Programming model choice
- Some new capabilities may be unlocked
  - Larger data sets
  - Workflows that utilize both halves

Stony Brook University

Stone Ridge Technology "Leap Ahead with Hopper"

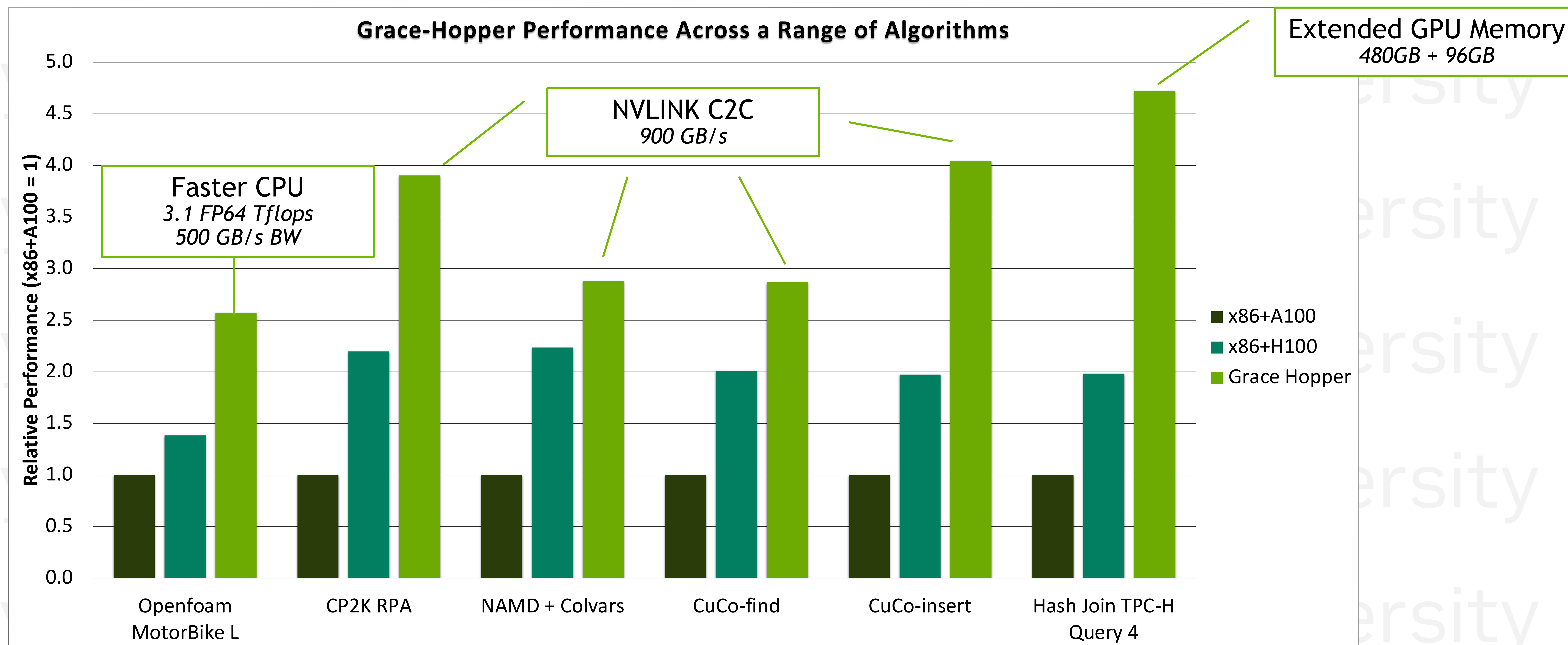
https://stoneridgetechnology.com/company/blog/leap-ahead-with-hopper-2/



- For ECHELON, **rebuilding was a trivial exercise**, and the resulting binary "just worked" on the Grace Hopper Superchip with no further tweaking required
- The performance gains were realized with **no modifications to the code**
- The average performance gain is **3.45x ± 1.07**
- The GPU portion of the code is performing so rapidly that whatever remains on CPU may be starting to illustrate Amdahl's law behavior
- It is also possible that the improved performance on Grace Hopper is due to the **increased bandwidth between GPU and CPU**
- Further optimization for the Grace Hopper system may provide more gains

Stony Brook University

Grace-Hopper Superchip Workload Performance



Stony Brook University

NVIDIA

Grace CPU Superchip

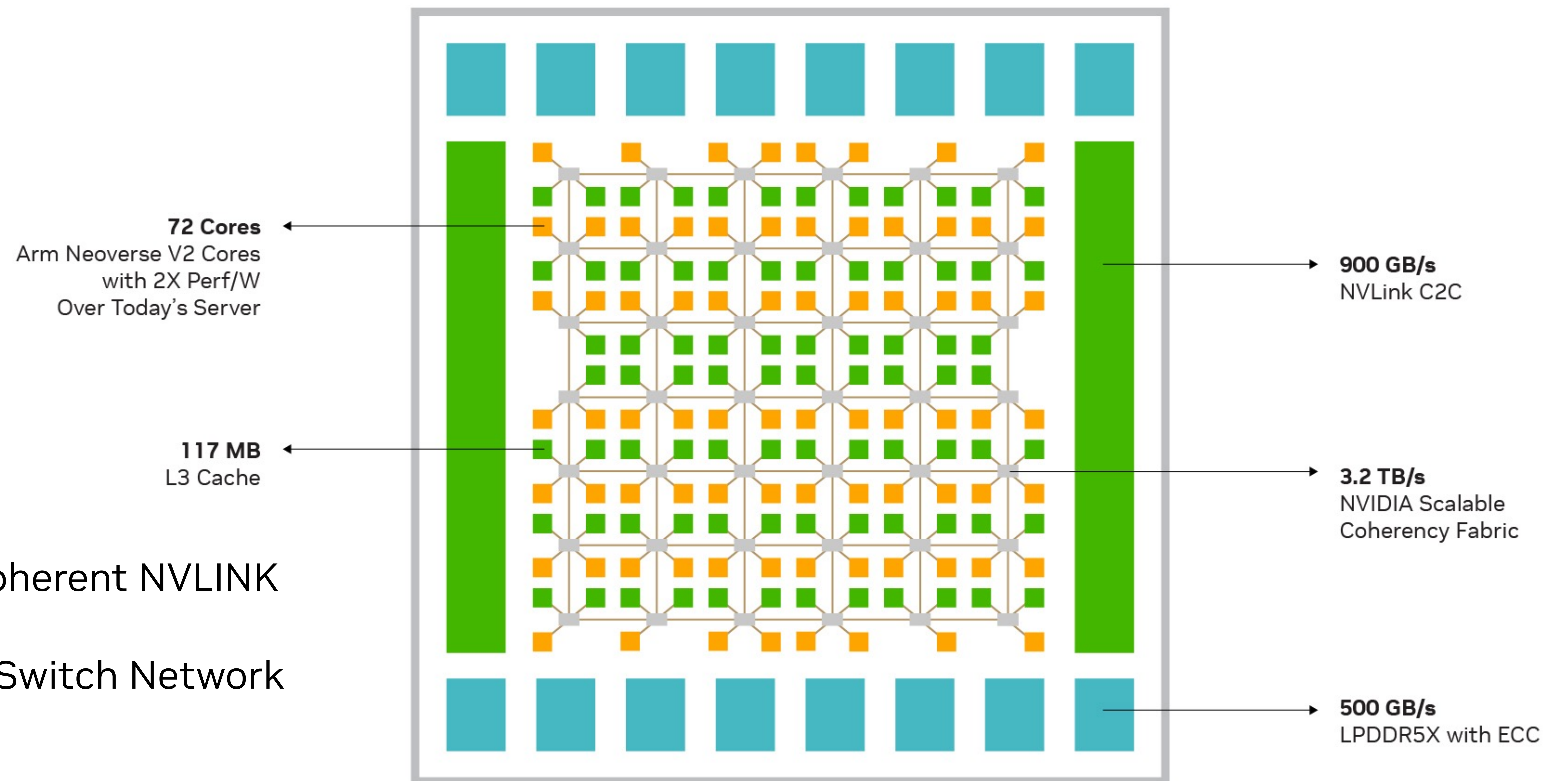
Grace CPU Superchip	
Architecture	Arm Neoverse V2 Armv9.0-A, **SVE2 4x128b SIMD**
Cores / Speed	144 cores, up to 3.2GHz
Memory	LPDDR5x soldered down, 1TB/s BW Up to 480GB per superchip
Cache	L1: 64KB i$ + 64KB d$ per core L2: 1MB per core L3: 234MB per superchip
Power	**500W including LPDDR5x memory**
Interfaces	Up to 8x PCIe Gen5 x16 HS interface
Process Node	TSMC 4N
Availability	Q3 2023

Stony Brook University

Grace is a Compute and Data Movement Architecture

NVIDIA Scalable Coherency Fabric (SCF) and distributed cache design

- **Single Die:** More efficient use of power
- 3,225.6 GB/s Bi-section BW
- 117MB of L3 cache
- Scalable to 72+ cores per die
- Local caching of remote die memory
- Supports up to 4-die coherency over Coherent NVLINK
- Background data movement via Cache Switch Network



Example possible fabric topology for illustrative purposes

Stony Brook University

Grace Simplifies System Design and Workload Optimization

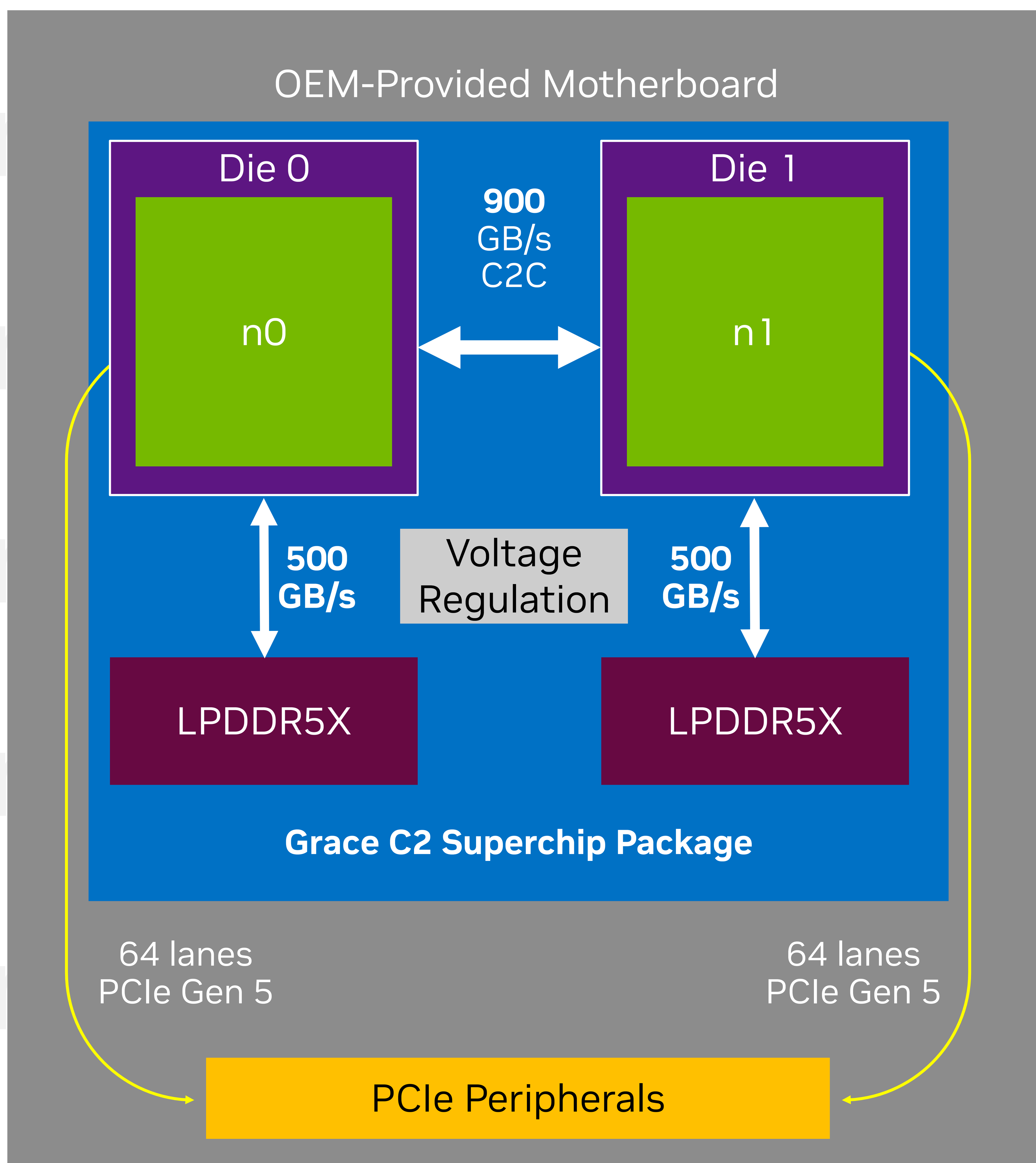
A high-performance server on a single superchip package

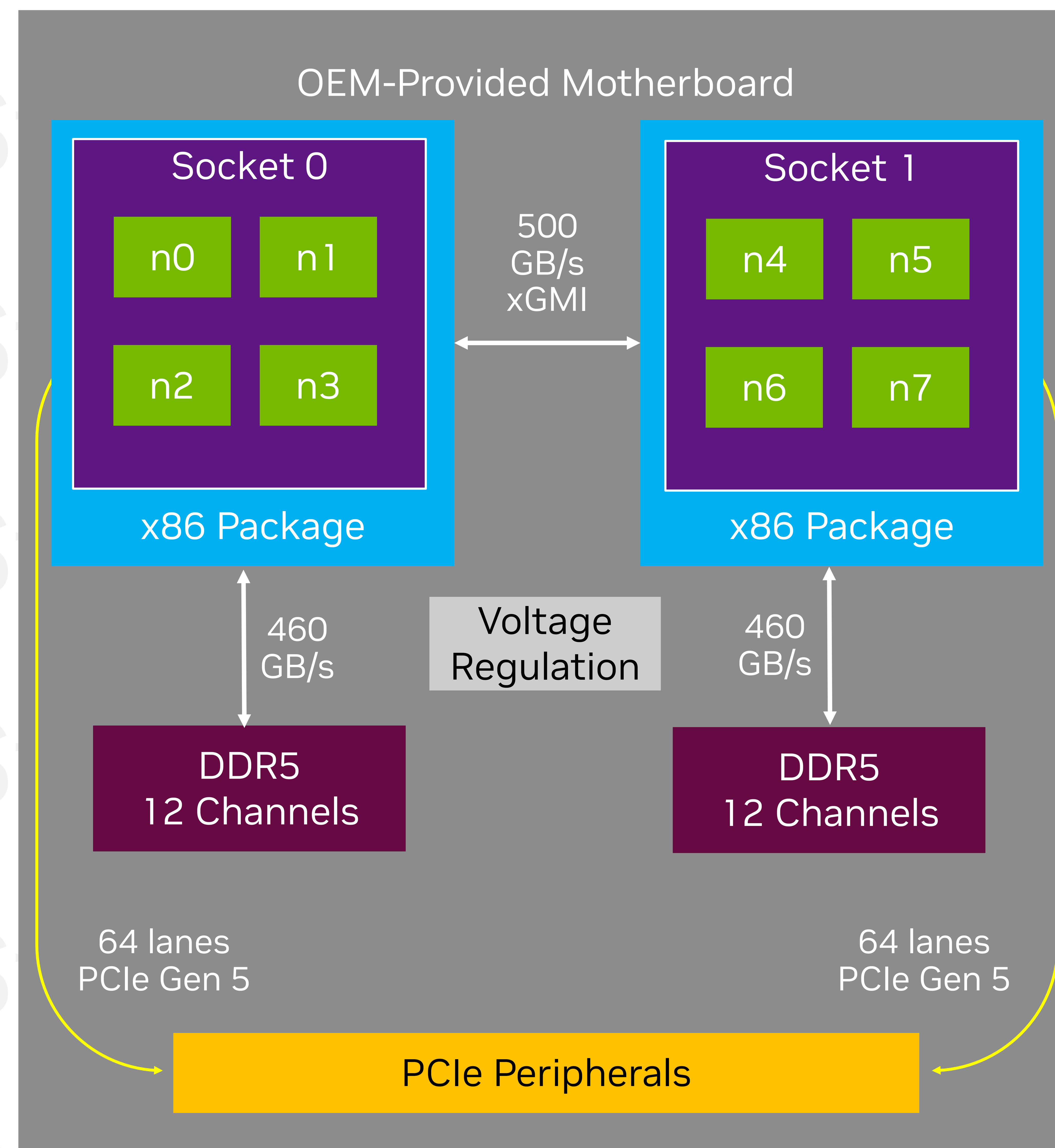
2 NUMA Nodes

2 Compute Dies

500 Watts (CPU + MEM)

900 GB/s worst-case n to n





8 NUMA Nodes

24 Compute Chiplets

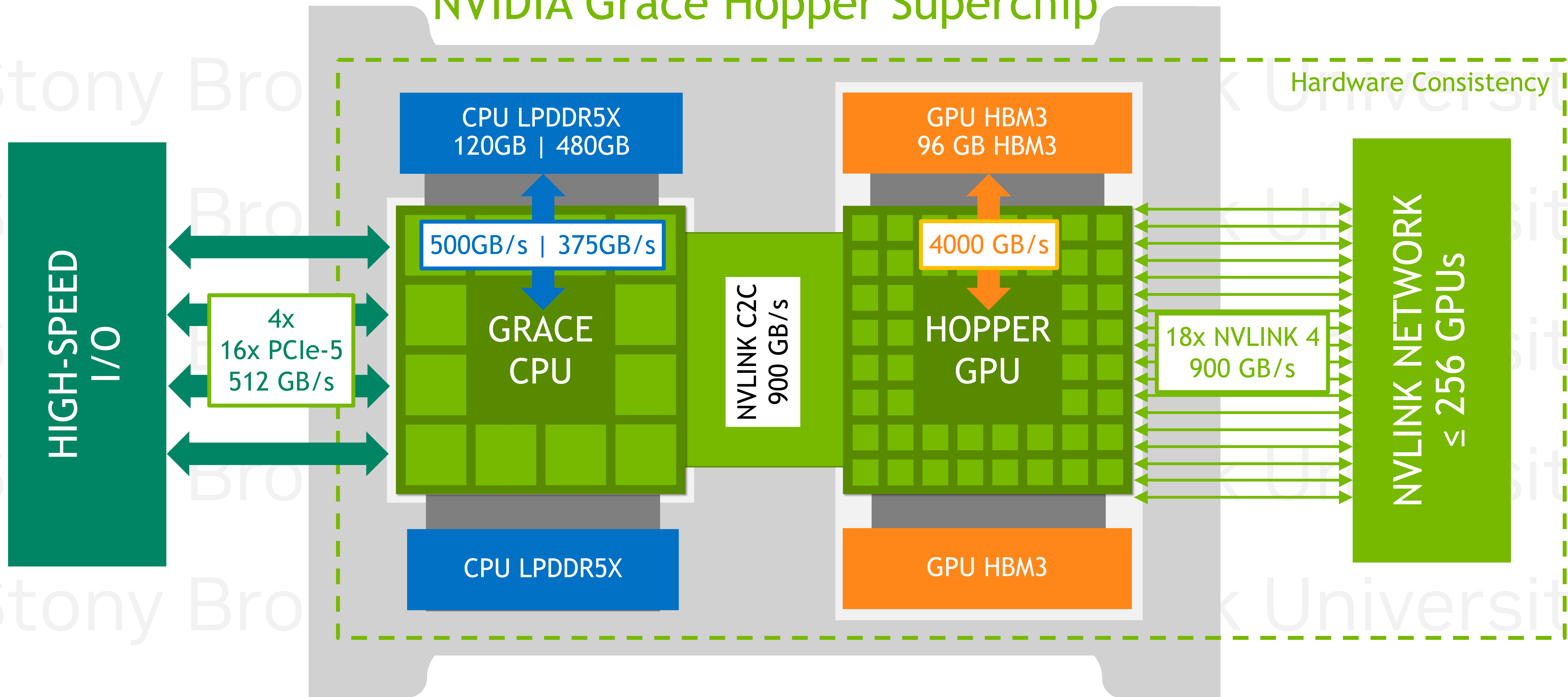
900+ Watts (CPU + MEM)

500 GB/s worst-case n to n

Stony Brook University

Grace Hopper Superchip

GPU can access CPU memory at CPU memory speeds



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

Stony Brook University

Programming the NVIDIA Platform

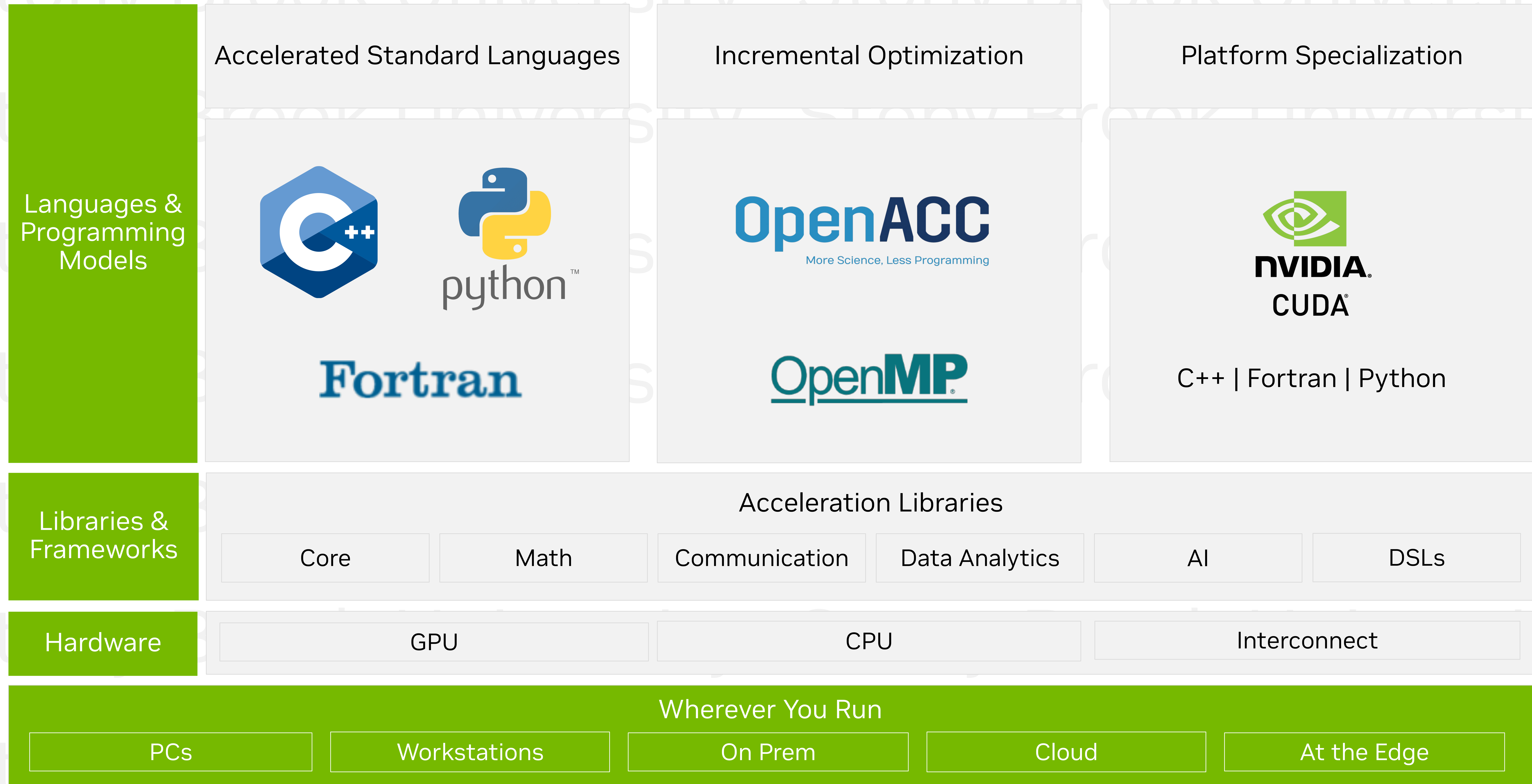
The Grace Software Ecosystem is Built on Standards

The NVIDIA platform builds on optimized software from the broad Arm software ecosystem

Stony Brook University

Programming the NVIDIA Platform

Unmatched Developer Flexibility

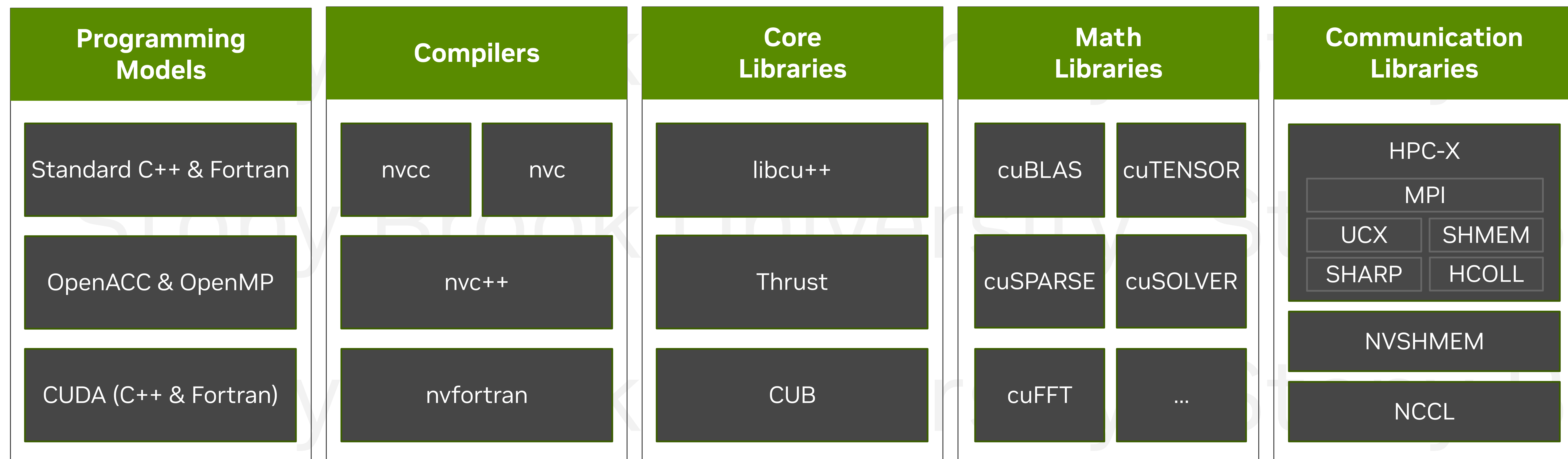


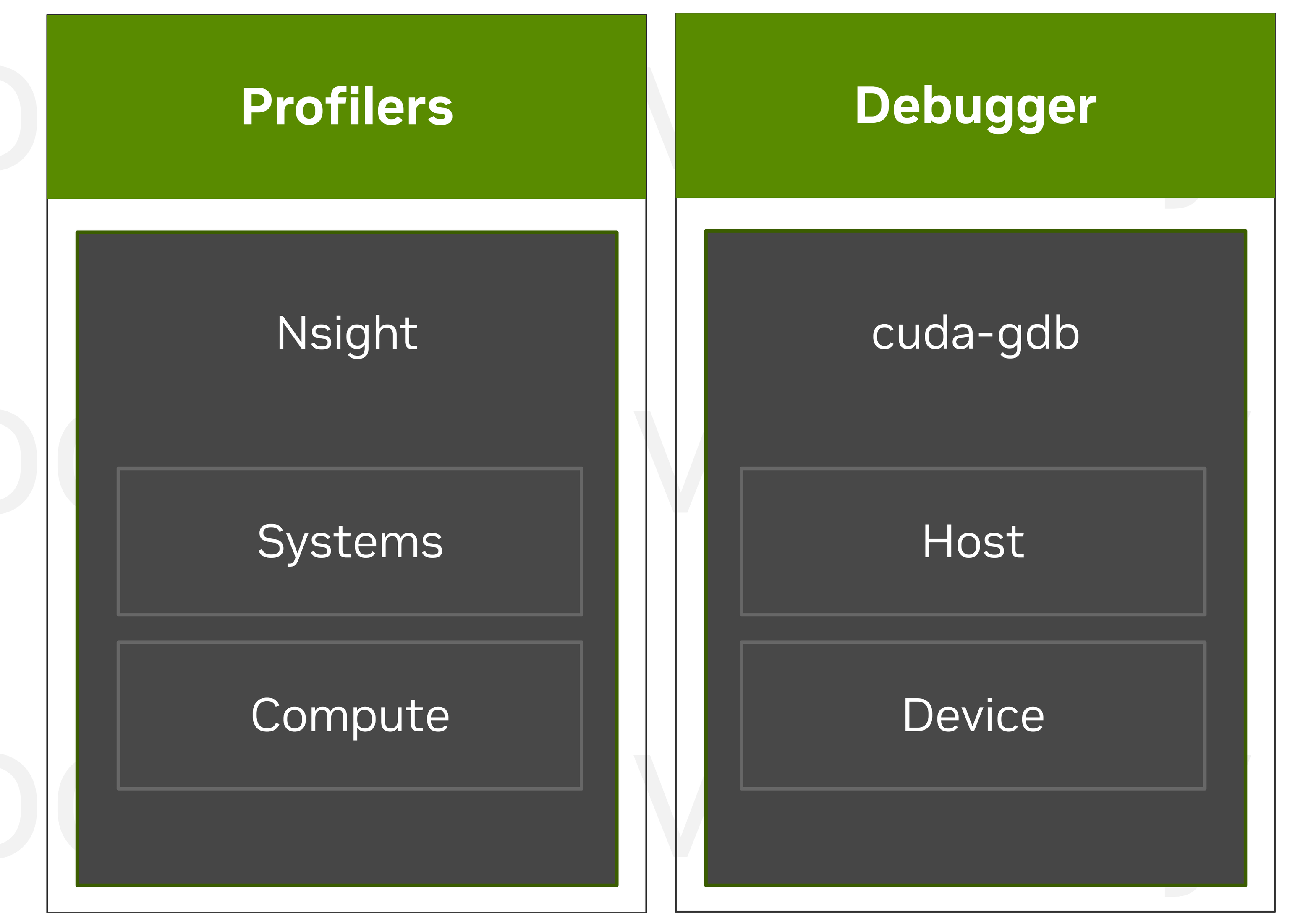
Stony Brook University

NVIDIA

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

6 Releases Per Year | Freely Available

Stony Brook University

HPC SDK Updates

Grace Hopper, unified memory, and more

- **HPC SDK 23.11:**
  - Unified memory support for stdpar, OpenACC, and CUDA C++/Fortran
  - NVTX improvements for stdpar codes
    - Now you can see your stdpar in NSight: improved tools support, developer experience, performance optimizations
  - C-Fortran Interface
    - Better multi-paradigm interoperability for mixed C, C++, and Fortran codes
    - F2008 MPI bindings for nvfortran
  - C++20 Coroutines for CPU
    - Future GPU support will enable alternative async models for stdpar
  - Support for Grace Hopper in all bundled components
    - Compilers, Math Libraries, Networking, Tools.
  - HPC-X is the default MPI implementation optimized for NV platform
  - Grace(/Arm) performance (-tp=neoverse-v2)
    - Re-engineered vectorizer, intrinsics, system math library functions

- **HPC SDK 24.3:**
  - Improved compile speed for nvc++
    - Up to 1.15x - 2x faster for some workloads
  - Unified memory support for OpenMP Target Offload
  - Integrated NVIDIA Performance Library (NVPL) for Grace CPUs
  - CUDA Fortran `unified` attribute
- **HPC SDK 24.5:**
  - New NVPL integrations
  - Ubuntu 24.04 support
  - Improved memory model CLI for HPC Compilers

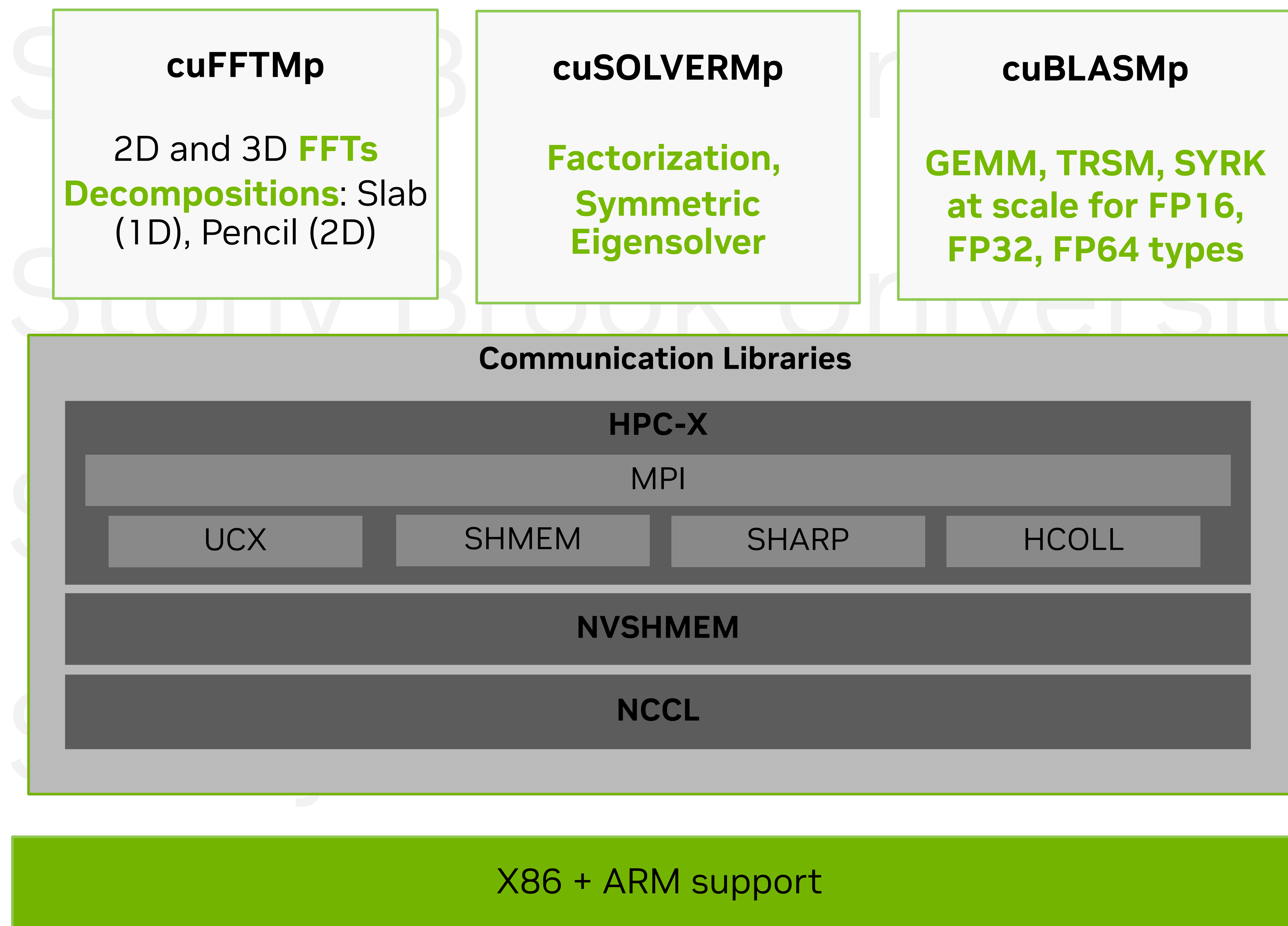
Unified Memory

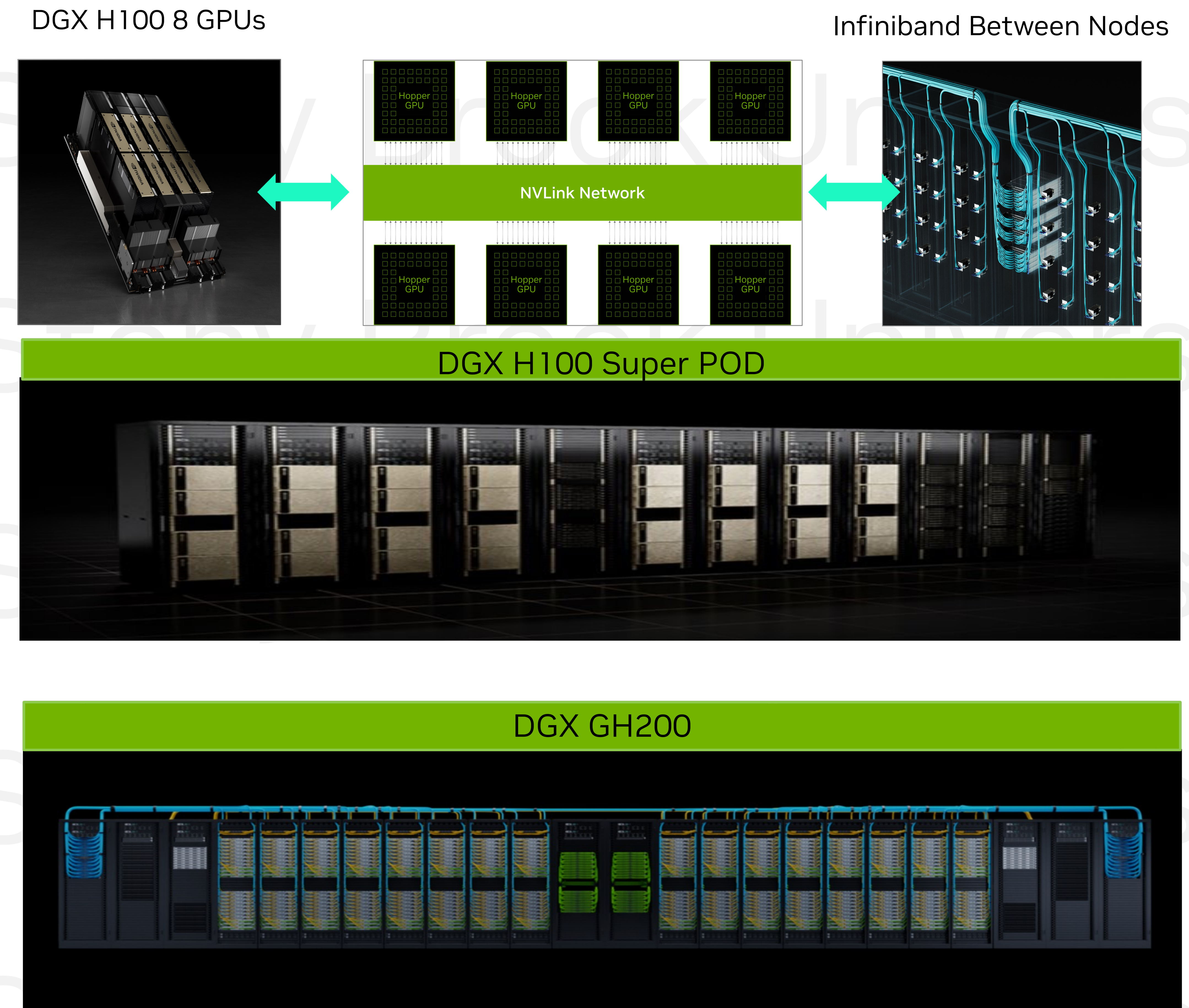
- C++ stdpar improvements
- Fortran stdpar improvements
- OpenACC improvements
- CUDA Fortran
- OpenMP Target Offload
- Unified Functions

Stony Brook University

Multi GPU Multi Node APIs

Scalable and Grace Hopper Support





256 Grace Hopper Superchips | **1EFLOPS** AI Performance | **144TB** unified fast memory | **900 GB/s** GPU-to-GPU bandwidth | **128 TB/s** bisection bandwidth

Stony Brook University

NVIDIA

Stony Brook University

NVIDIA Performance Libraries (NVPL)

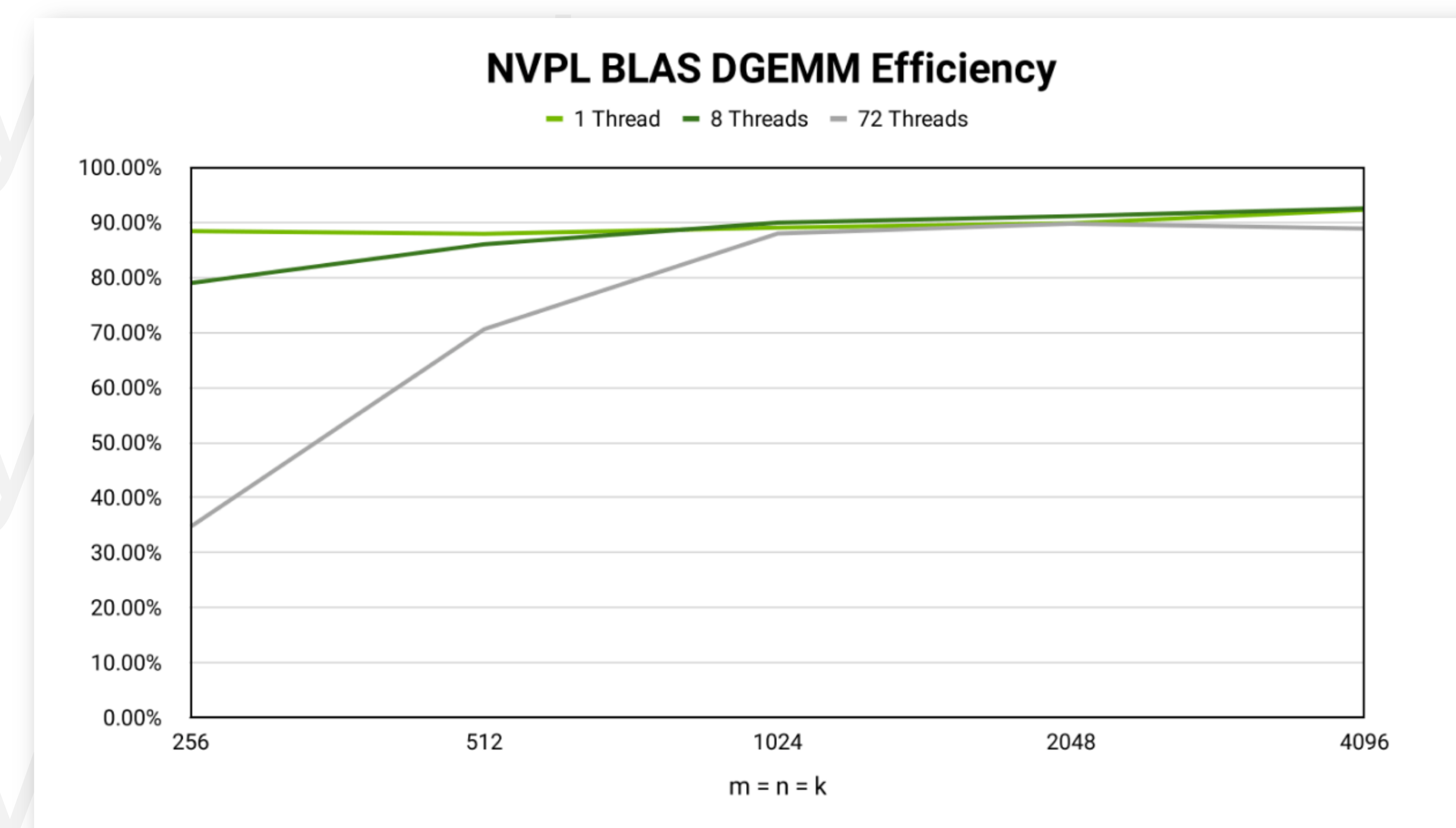
Optimized math libraries for NVIDIA CPUs

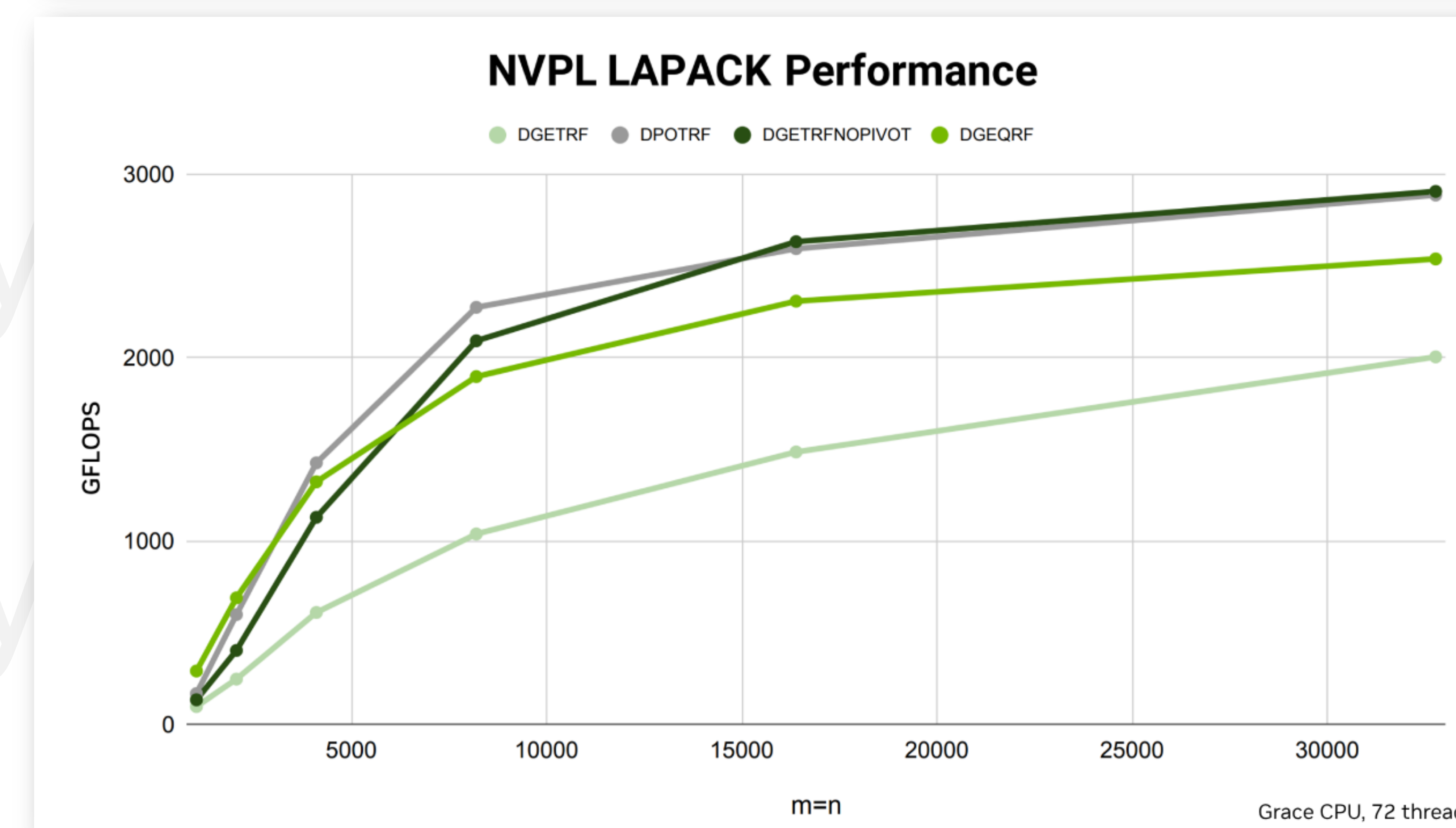
- Easily port applications to NVIDIA's Arm CPUs
- Drop-in replacement for any math library implementing standard interfaces (e.g. Netlib, FFTW)
- New interfaces for high-performance libraries

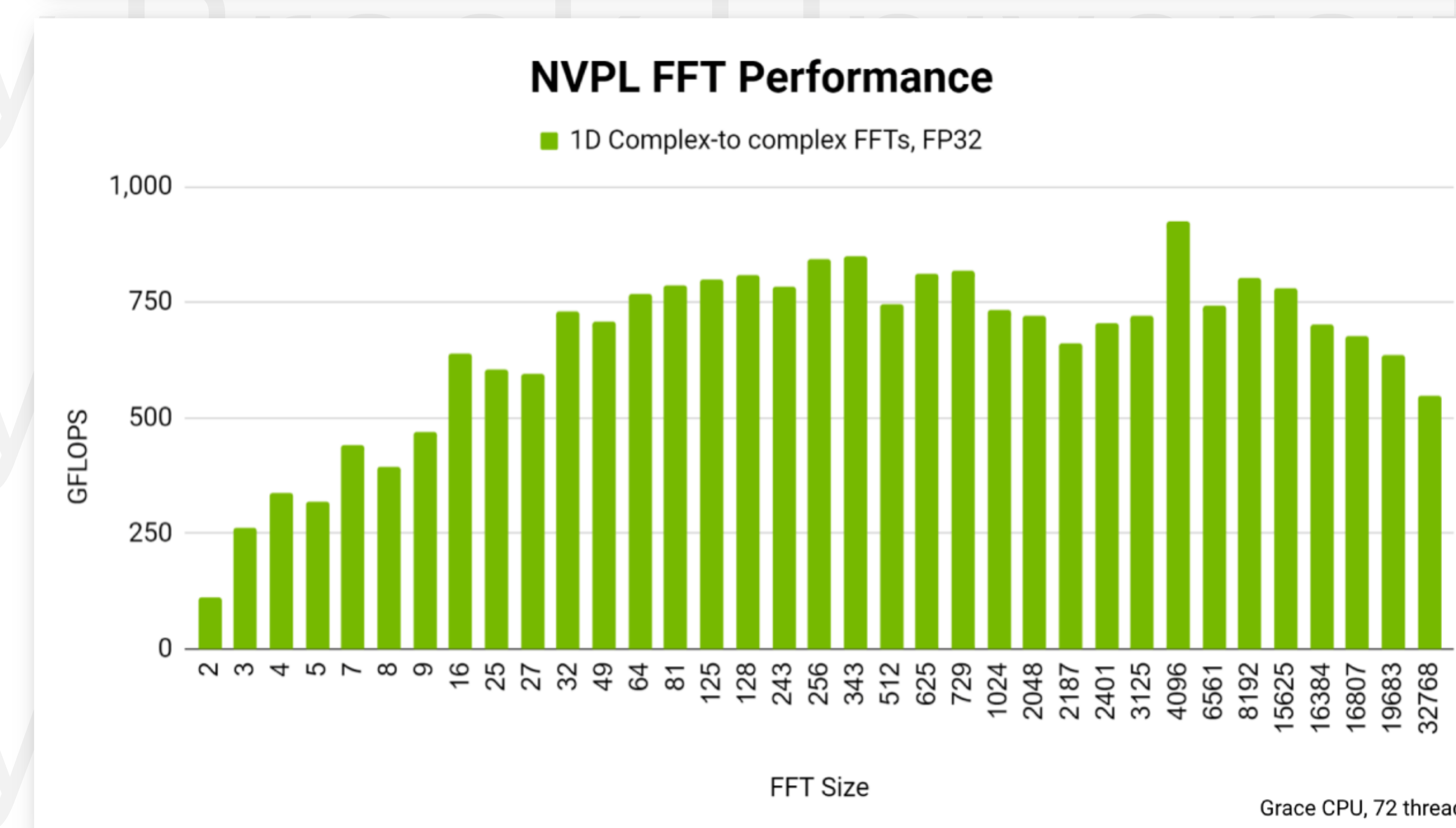
BLAS	LAPACK	PBLAS	SCALAPACK
TENSOR	SPARSE	RAND	FFT

Download Now

www.developer.nvidia.com/nvpl







Stony Brook University

Clang for NVIDIA Grace

An optimized build of LLVM Clang for the NVIDIA Grace CPU

Stony Brook University

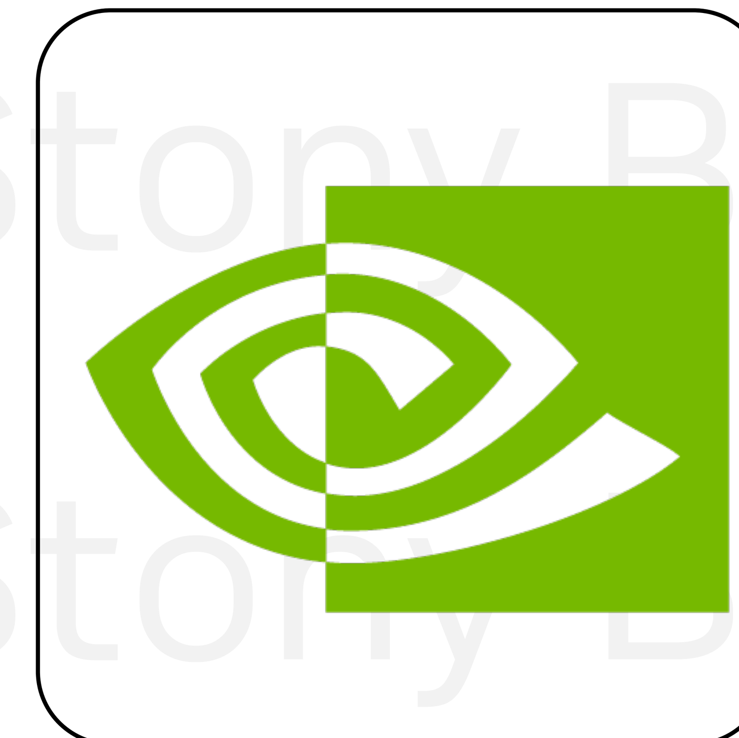
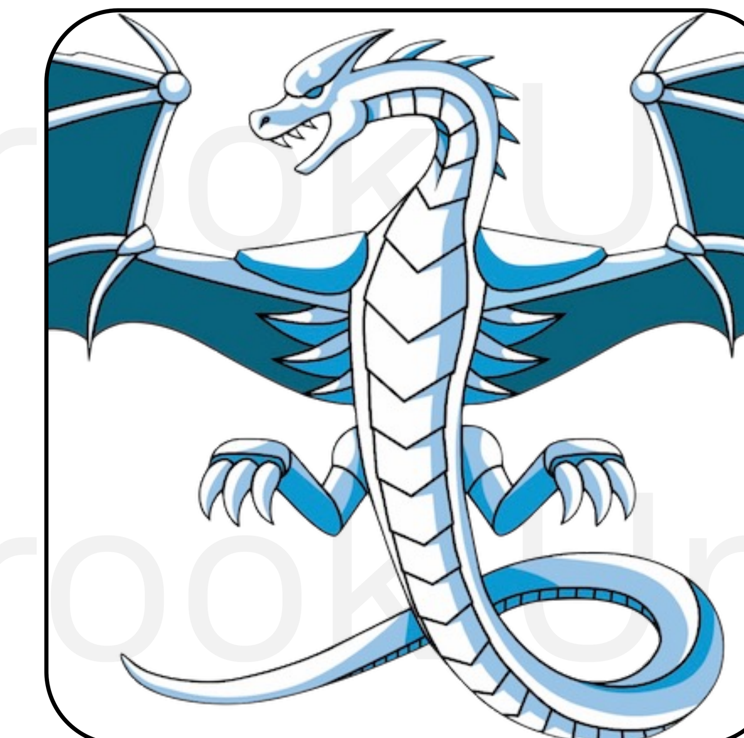
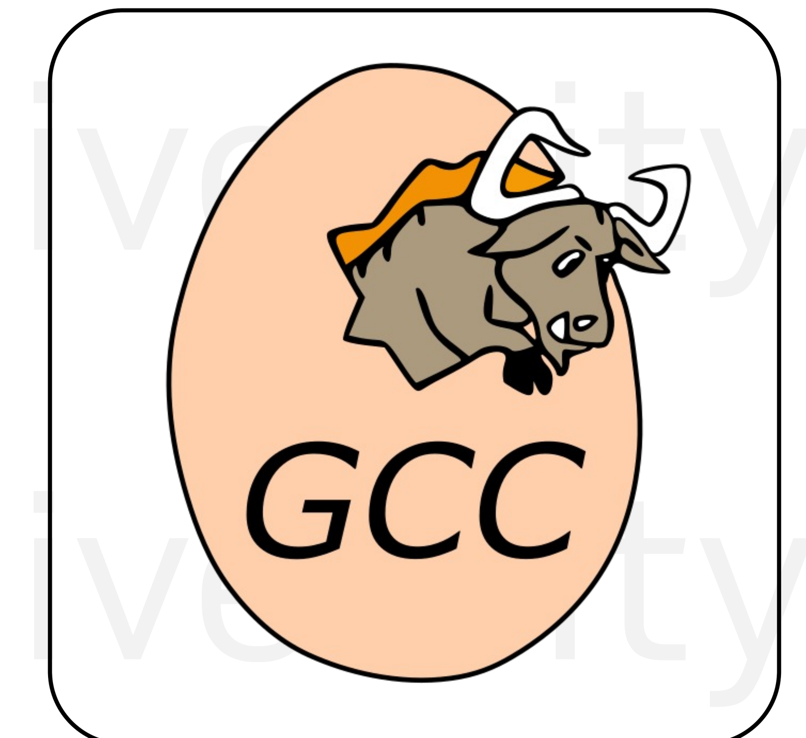
- Optimized builds of the open-source LLVM Clang compiler for rapid access to the latest LLVM improvements for the Grace CPU
- Certified CUDA host compiler
- Optimized compile times: 15% faster vs. mainline LLVM
- Current release based on **LLVM 18.1.1**
  - C compiler driver binary - **clang**
  - C++ compiler driver binary - **clang++**
  - LLVM Linker - **lld**
  - OpenMP Runtime support - **libomp**
- www.developer.nvidia.com/grace/clang

Architecture	Linux Distributions	CUDA Toolkit
AArch64	• Ubuntu 22.04 • RHEL 9 • CentOS 9 • SLES 15-SP4	12.2U2 and later

Advancing the State-of-the-Art in Compilers

NVIDIA invests in open source and commercial compilers for NVIDIA Grace

- **NVIDIA HPC Compilers**
  - Focused on application performance and programmer productivity
  - High velocity, constant innovation
  - Freely available with commercial support option
- **LLVM and Clang**
  - NVIDIA provides builds of Clang for Grace
    - https://developer.nvidia.com/grace/clang
  - Drop-in replacement for mainline Clang
  - 100% of Clang enhancements for Grace are contributed to mainline LLVM
- **GCC**
  - NVIDIA contributes to mainline GCC to support Grace
  - Working with all major Linux distros to improve availability of Grace optimizations in GCC

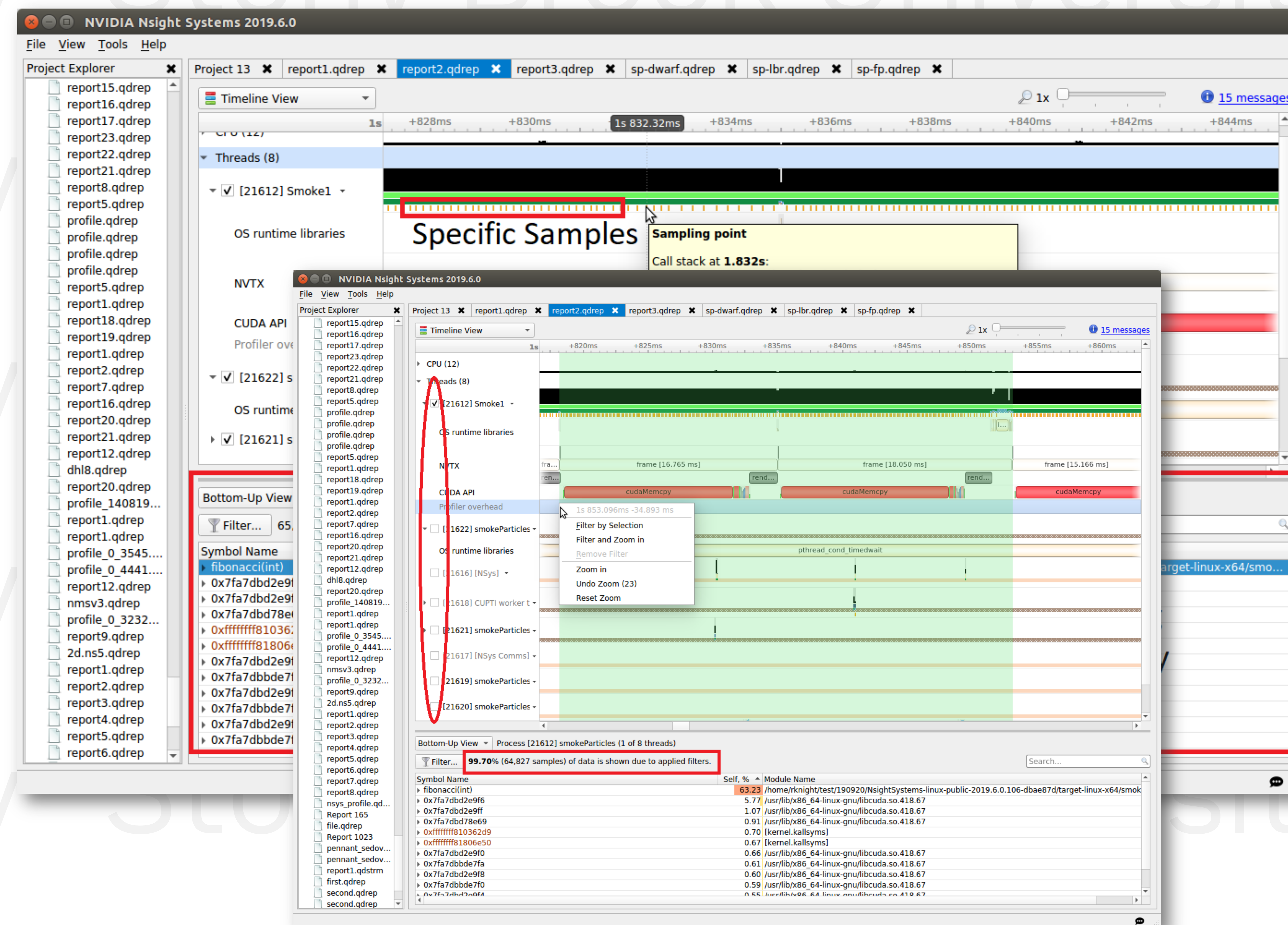




Stony Brook University

Debuggers and Profilers for GH200 and Grace CPU Superchip

Full capability on Grace-Hopper

- **NVIDIA Nsight has full feature-parity on GH200**
  - Anything you can do with Nsight tools on x86+Hopper, you can do on GH200 with the same workflow
- **GH200 has hundreds of performance counters (PMUs)**
  - **Computational intensity, bandwidth, instruction mix…**
- **Generally, all major debugging and profiling tools for x86+Hopper are available on GH200**
  - Similar capabilities are provided by other tools on Grace



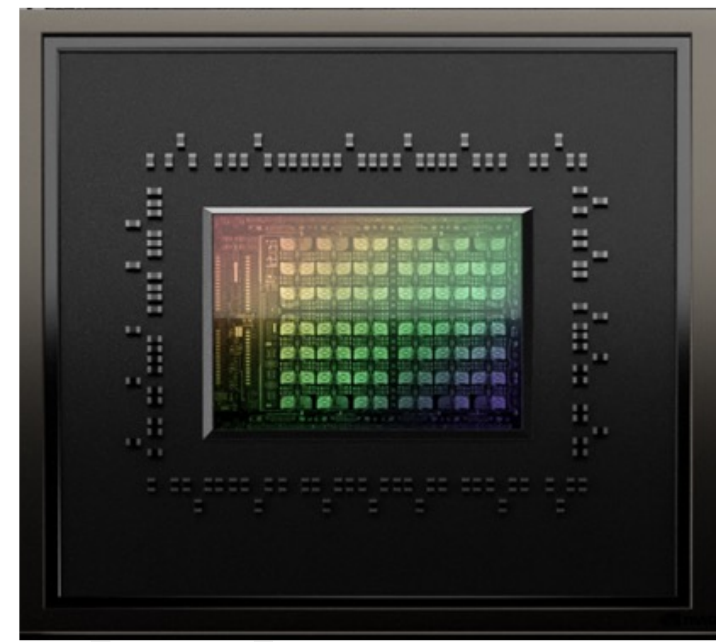
Stony Brook University

NVIDIA

Porting and Optimizing for NVIDIA Grace CPU

Expectation: It Just Works

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



Reuse
- NGC
- Your Linux Distro’s Package Manager

Recompile
- NVIDIA Compiler
- GCC
- LLVM
- Spack or EasyBuild

Run
- High core count
- Threads-per-process
- Update tests

Optimize
- NVIDIA Nsight
- Arm Forge
- Perf, PAPI, TAU, Score-P, …

Quick Launch

- NGC containerized applications, frameworks, and toolkits
- `./configure && make install`

Compilation Tips

- Most compiler flags are the same:
  - Use `-mcpu=native`
  - Don’t use `-march` or `–mtune`
  - You may need `–fsigned-char`
- Update your unit tests:
  - Aarch64 floating point is as accurate as all other platforms

Stony Brook University

Workloads of a National Computing Center

Annual core hours

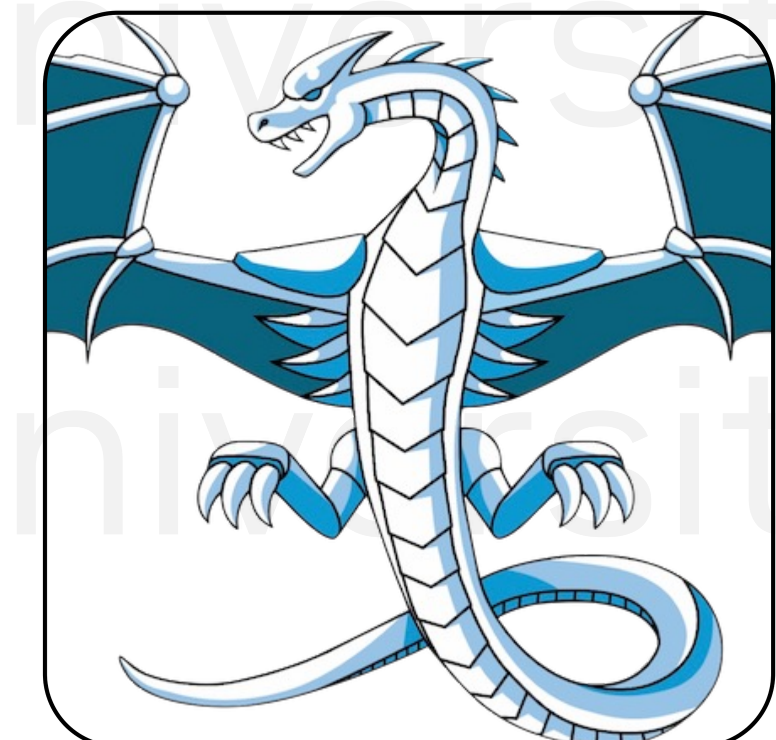
Stony Brook University

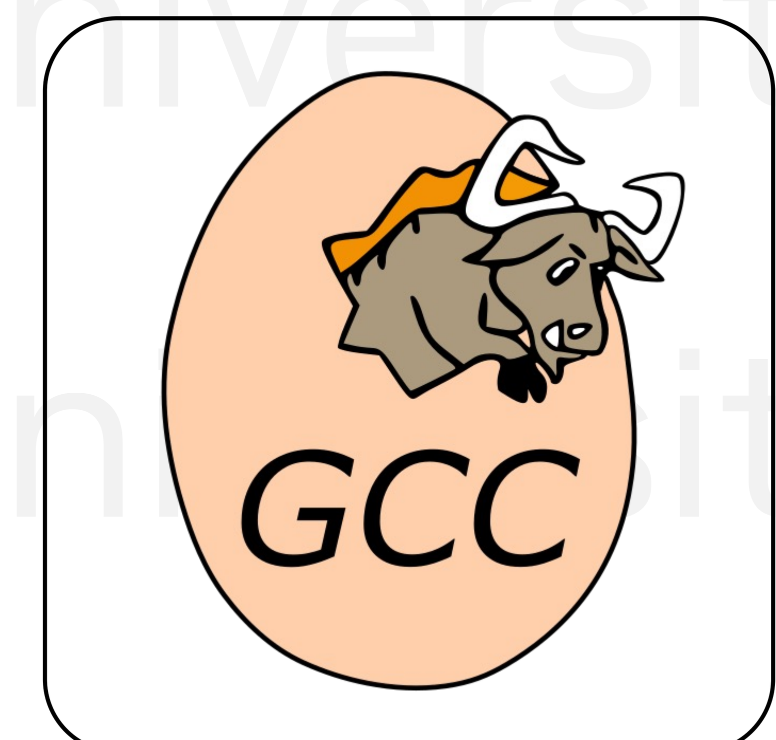
NVIDIA

Use Standards-compliant Multi-platform Compilers

You're not porting to Arm. You're porting *away* from ifort, xlf, etc.

- Use any portable multi-platform compiler: NVIDIA, GCC, LLVM, etc.
- Use the most recent compiler possible. **GCC 12+** is strongly recommended.
- Beware of non-standard build systems
  - `icc`, `ifort`, `xlf`, etc. may be hard-coded into the build system
  - Be explicit about which compiler to use. Don't let the build system make assumptions
- Beware of non-standard default compilers
  - Check default compiler commands (`cc`, `fc`, `gcc`, etc.) invoke a recent compiler
  - Use `` `mpicc -show` `` to verify that MPI compiler wrappers invoke the right compiler
- Log the build, then check the log afterward



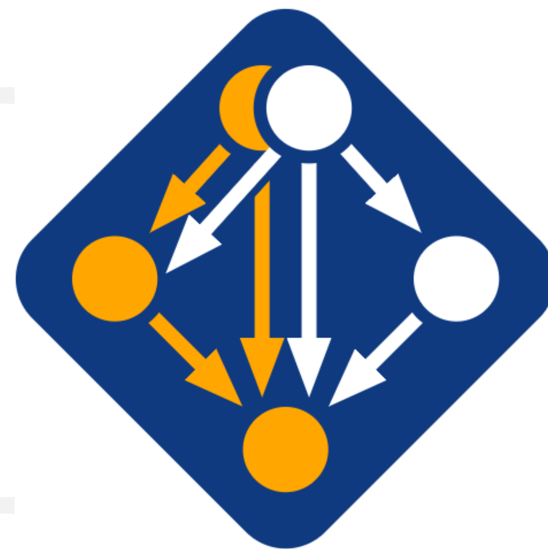


Stony Brook University

No Cross Compiling! Just Don't.

All popular build systems are supported – and *performant* – on Arm

- GCC and LLVM are excellent Arm compilers
  - Auto-vectorizing, auto-parallelizing, tested, in production
  - Arm & partners are the majority of GCC contributors
- All major build systems and tools work on Arm
  - CMake, Make, GNUMake, EasyBuild, Spack etc.
- Compiler & build system performance is excellent
  - Ampere Altra compilation performance is on is on-par with AMD EPYC 7742 – **you do not need to cross compile**



https://www.anandtech.com/show/16315/the-ampere-altra-review/8

Stony Brook University

Selecting GNU and LLVM Compiler Flags for Grace

Similar flags have different meanings across compilers and across platforms

- Remove all architecture-specific flags: `-mavx`, `-mavx2`, etc.
- Remove `-march` and `-mtune` flags
  - These flags have a different meaning on aarch64
  - See How to Optimize for Arm and not get Eaten by a Bear for details
- Use **`-Ofast -mcpu=native`**
  - If fast math optimizations are not acceptable, use `-O3 -ffp-contract=fast`
  - For even more accuracy, use `-ffp-contract=off` to disable floating point operation contraction (e.g. FMA)
  - Can also use `-mcpu=neoverse-v2`, but `-mcpu=native` will "port forward"
- Use `-flto` to enable link-time optimization
  - The benefits of link-time optimization vary from code to code, but can be significant
  - See https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html for details
- Apps may need `-fsigned-char` or `-funsigned-char` depending on the developer's assumption
- gfortran may benefit from `-fno-stack-arrays`

NVIDIA

__atomic_add_fetch(&var, num, __ATOMIC_RELAXED)

GCC 12.3 on Grace

Missing ISA Extensions: i8mm and bf16

```
        .arch armv9-a+crc
        .file   "foo.c"
        .text
        .align  2
        .global main
        .type   main, %function
main:
.LFB0:
        .cfi_startproc
        sub     sp, sp, #16
        .cfi_def_cfa_offset 16
        str     wzr, [sp, 8]
        mov     w0, 1
        str     w0, [sp, 12]
        ldr     w1, [sp, 12]
        add     x0, sp, 8
        ldadd   w1, w0, [x0]
        mov     w0, 0
        add     sp, sp, 16
        .cfi_def_cfa_offset 0
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (GNU) 12.3.0"
        .section        .note.GNU-stack,"",@progbits
```

Atomic Add

-march=armv9-a

Correct instruction, limited ISA

Armv8: No SVE!

```
        .arch armv8-a
        .file   "foo.c"
        .text
        .global __aarch64_ldadd4_relax
        .align  2
        .global main
        .type   main, %function
main:
.LFB0:
        .cfi_startproc
        stp     x29, x30, [sp, -32]!
        .cfi_def_cfa_offset 32
        .cfi_offset 29, -32
        .cfi_offset 30, -24
        mov     x29, sp
        str     wzr, [sp, 24]
        mov     w0, 1
        str     w0, [sp, 28]
        ldr     w2, [sp, 28]
        add     x0, sp, 24
        mov     x1, x0
        mov     w0, w2
        bl      __aarch64_ldadd4_relax
        mov     w0, 0
        ldp     x29, x30, [sp], 32
        .cfi_restore 30
        .cfi_restore 29
        .cfi_def_cfa_offset 0
        ret
        .cfi_endproc
```

libgcc call

-mtune=neoverse-v2

Library call instead of atomic instruction, limited ISA

Correct ISA

```
        .arch armv9-a+crc+profile+rng+memtag+sve2-bitperm+i8mm+bf16
        .file   "foo.c"
        .text
        .align  2
        .global main
        .type   main, %function

        .cfi_startproc
        sub     sp, sp, #16
        .cfi_def_cfa_offset 16
        str     wzr, [sp, 8]
        mov     w0, 1
        str     w0, [sp, 12]
        ldr     w1, [sp, 12]
        add     x0, sp, 8
        ldadd   w1, w0, [x0]
        mov     w0, 0
        add     sp, sp, 16
        .cfi_def_cfa_offset 0
        ret
        .cfi_endproc

        .size   main, .-main
        .ident  "GCC: (GNU) 12.3.0"
        .section        .note.GNU-stack,"",@progbits
```

Atomic Add

-mcpu=neoverse-v2 (or -mcpu=native)

Correct instruction, correct ISA

Stony Brook University

Porting Applications that use Math Libraries: MKL, OpenBLAS, etc.

Several library options to choose from

- Prefer Netlib BLAS/LAPACK and FFTW interfaces
  - Building on these interfaces enables compatibility

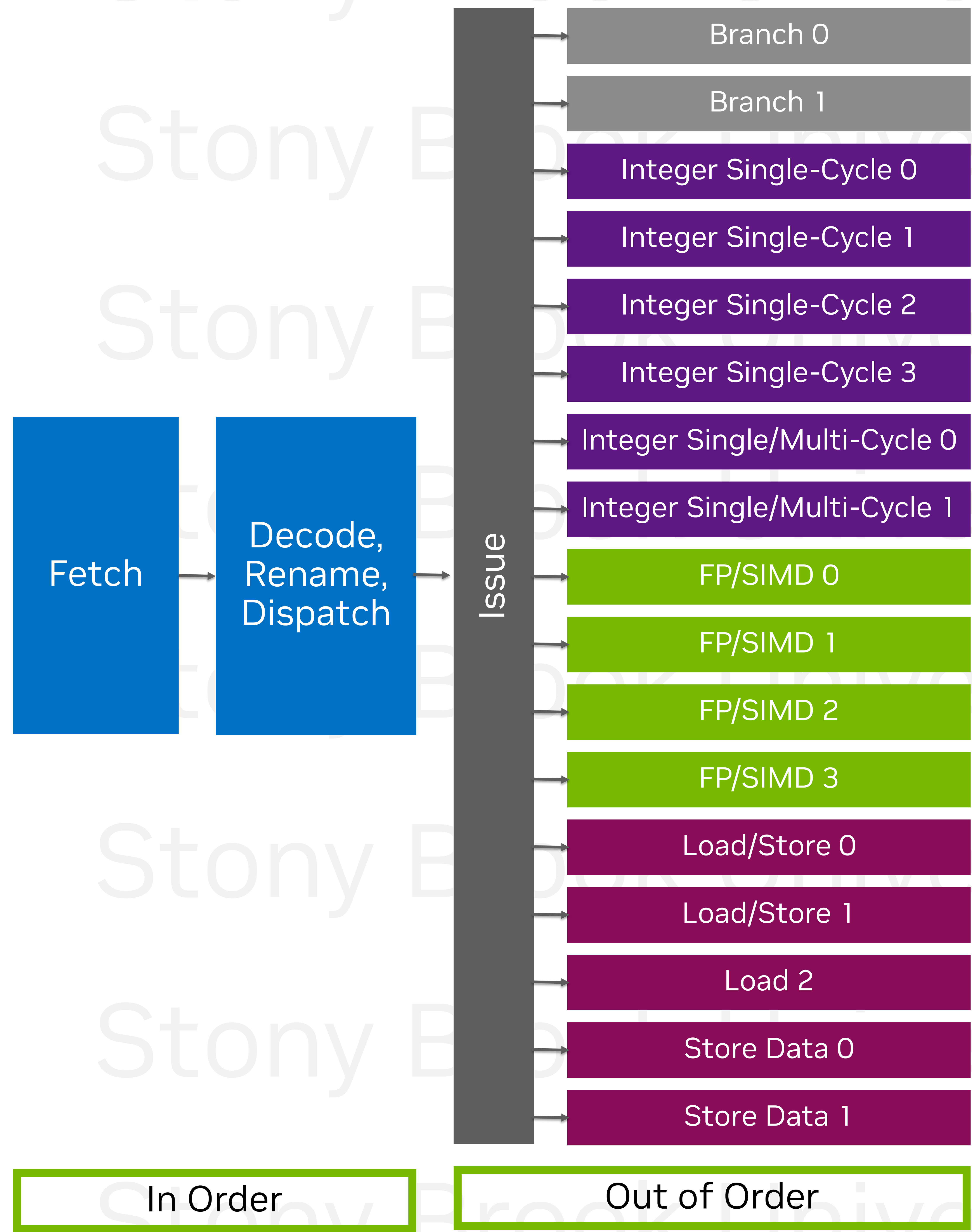
- **NVPL**
  - ```
    gcc -DUSE_CBLAS -ffast-math -mcpu=native -O3 \
        -I/PATH/TO/nvpl/include \
        -L/PATH/TO/nvpl/lib \
        -o mt-dgemm.nvpl mt-dgemm.c \
        -lnvpl_blas_lp64_gomp
    ```

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

```
libnvpl_blas_ilp64_gomp.so
libnvpl_blas_ilp64_seq.so
libnvpl_blas_lp64_gomp.so
libnvpl_blas_lp64_seq.so
libnvpl_fftw.so
libnvpl_lapack_ilp64_gomp.so
libnvpl_lapack_ilp64_seq.so
libnvpl_lapack_lp64_gomp.so
libnvpl_lapack_lp64_seq.so
libnvpl_rand_mt.so
libnvpl_rand.so
libnvpl_scalapack_ilp64.so
libnvpl_scalapack_lp64.so
libnvpl_sparse.so
libnvpl_tensor.so
```

- **ATLAS**, **OpenBLAS**, **BLIS**, … Community supported with some optimizations for Neoverse V2.
  - Works on Grace, but unlikely to outperform NVPL and ArmPL. A good compatibility option.

NVIDIA.



SIMD in NVIDIA Grace

- 4x128b SIMD units = 512b SIMD vector bandwidth
- Full core frequency at 100% 512b SIMD utilization
  - With all cores at 100%, a fully loaded socket may downclock about 200MHz
- Each SIMD unit can retire NEON or SVE2 instructions
- On this architecture, SVE2 and NEON have the **same peak performance** …
- … but SVE2 can vectorize more complex codes and supports more data types than NEON
- In practice, SVE2 typically outperforms NEON

Stony Brook University

NVIDIA

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
  - **Demonstration**: https://www.nvidia.com/en-us/on-demand/session/gtcspring22-s41702/

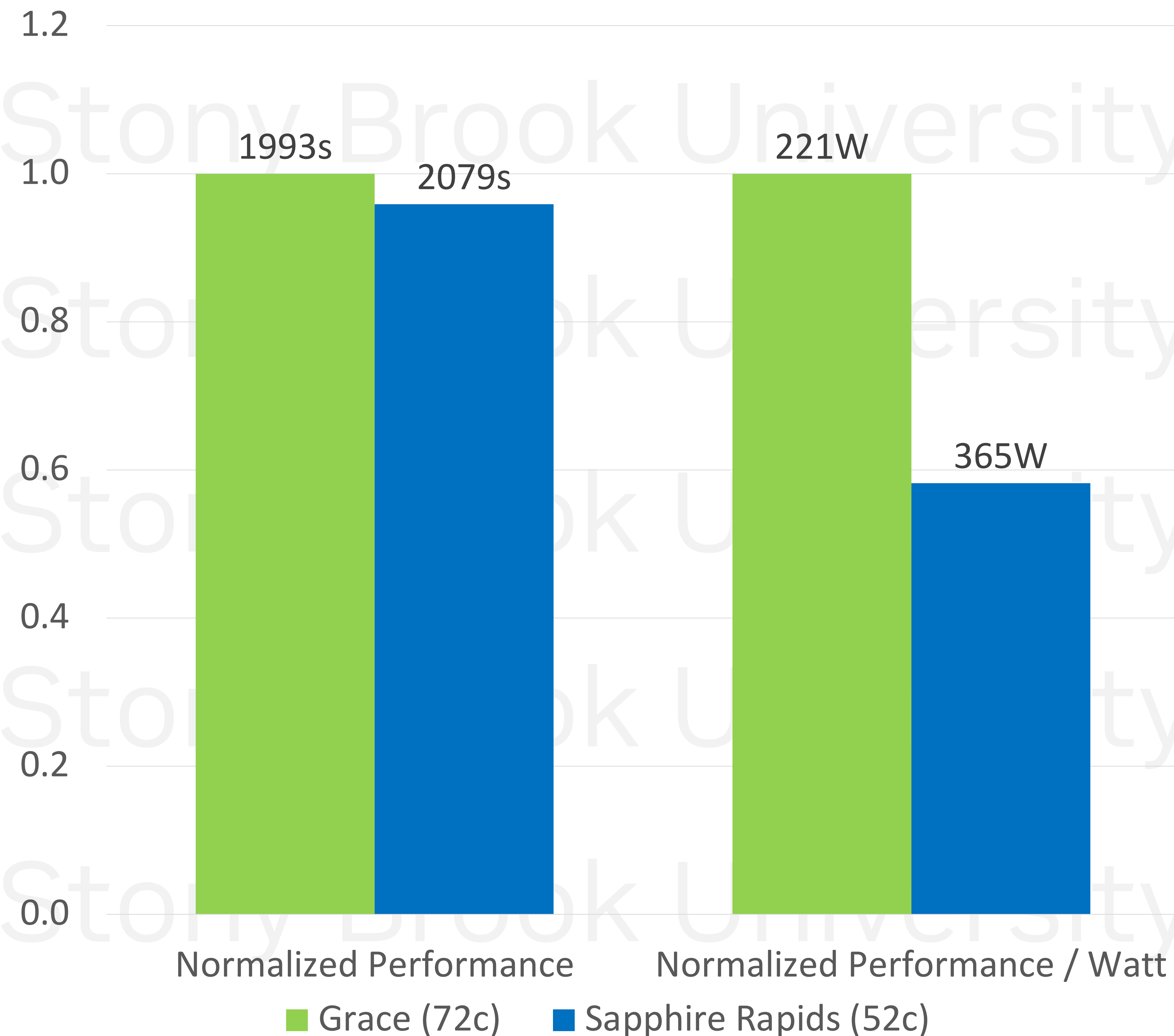
- Follow Arm's documentation on rewriting x86 vector intrinsics
  - Porting and Optimizing HPC Applications for Arm SVE [https://developer.arm.com/documentation/101726/latest]
  - Coding for NEON [https://developer.arm.com/documentation/101725/0300/Coding-for-Neon]

- 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

Stony Brook University

Porting x86 Intrinsics: BWA-MEM2

Auto-translation overhead is offset by CPU performance advantage



Scope of "porting" work, no optimization done:

~3 hours of developer time to investigate and add:

#include: AVX -> Vendor Nonspecific SIMD Wrapper

- https://github.com/simd-everywhere/simde

Tools:

- Compilers: Clang 16 (NVIDIA)
- Compile options: GCC 12 and SIMDe

Comparison:

- Precompiled binaries on x86
- HG002 dataset from Illumina paired-end sequencers
- Complete human genome at 30x coverage

Run configuration:

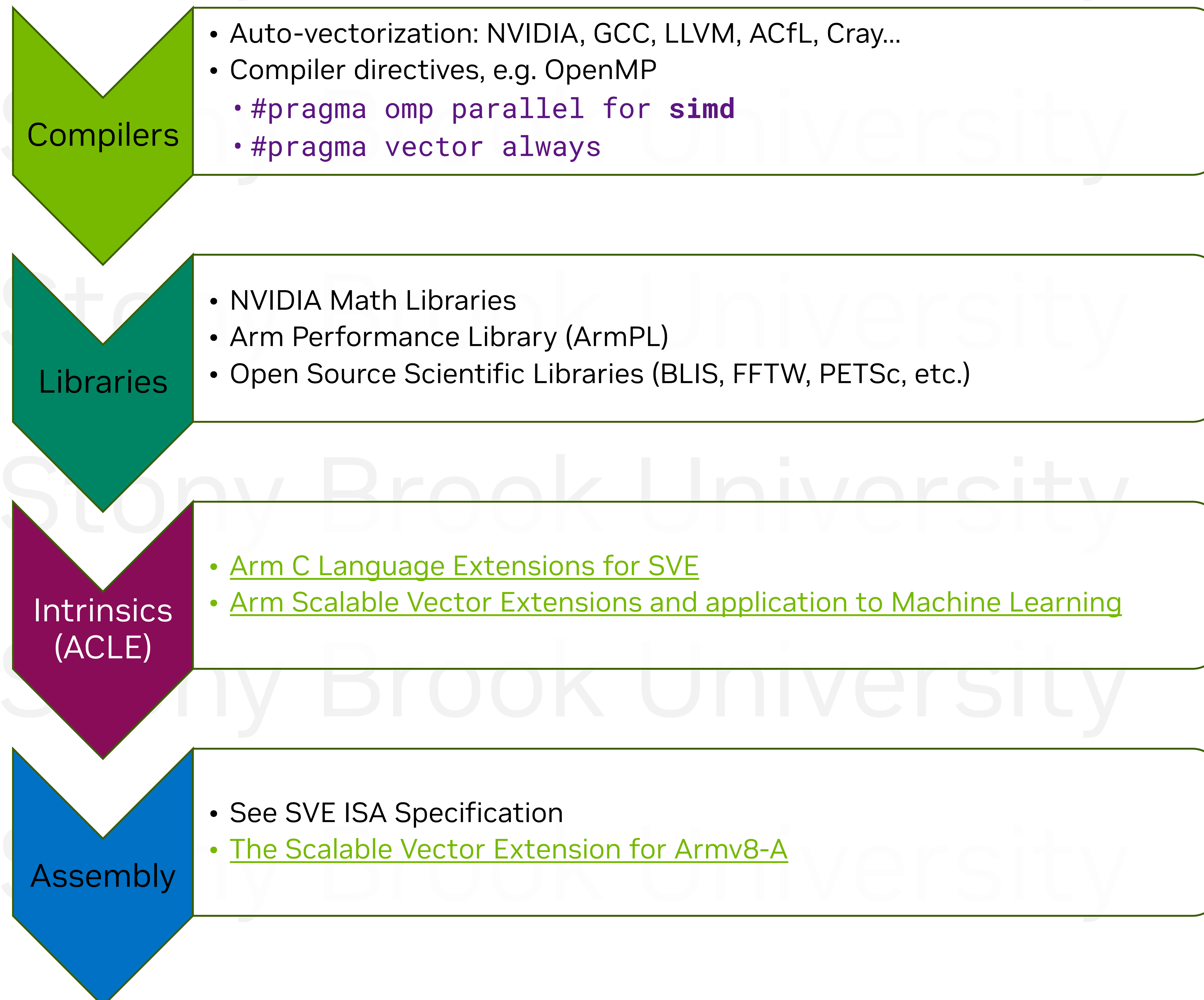
Similar configuration overhead to moving between Intel & AMD

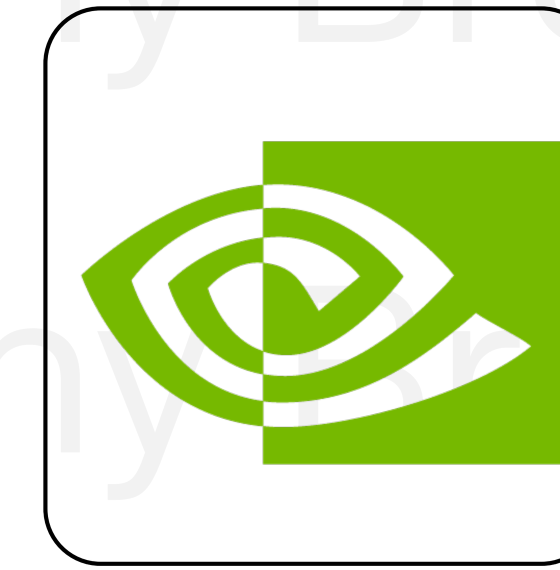
- Grace: jemalloc + transparent huge pages
- Intel: AVX512 intel-optimized version on SPR

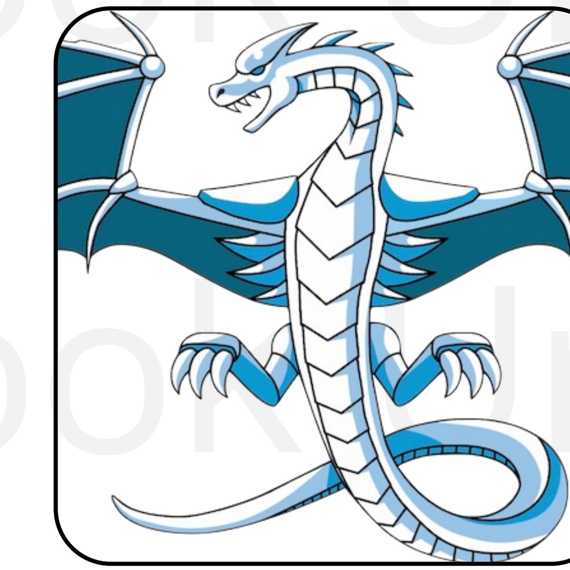
Stony Brook University

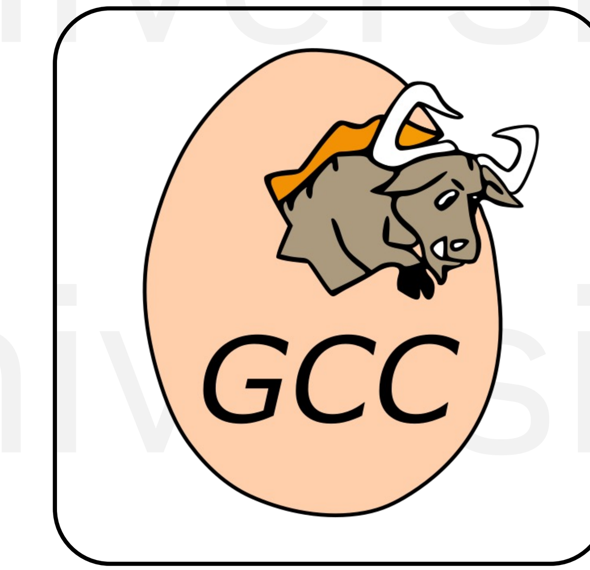
SIMD Programming Approaches

Follow these recommendations in order, e.g. prefer auto-vectorization over intrinsics









```
// Complex dot product
// (a+ib)*(c+id) = (ac - bd) + i(ad + bc)
void complex_dot_product(Complex_t c[SIZE], Complex_t a[SIZE], Complex_t b[SIZE])
{
    uint32_t vl = svcntw();

    svbool_t p32_all = svptrue_b32();

    for (int i=0; i<SIZE; i+=vl) {
        svfloat32x2_t va = svld2(p32_all, (float32_t*)&a[i]);
        svfloat32x2_t vb = svld2(p32_all, (float32_t*)&b[i]);
        svfloat32x2_t vc = svld2(p32_all, (float32_t*)&c[i]);

        vc.v0 = svmla_m(p32_all, vc.v0, va.v0, vb.v0); //c.re += a.re * b.re
        vc.v1 = svmla_m(p32_all, vc.v1, va.v1, vb.v0); //c.im += a.im * b.re
        vc.v0 = svmls_m(p32_all, vc.v0, va.v1, vb.v1); //c.re -= a.im * b.im
        vc.v1 = svmla_m(p32_all, vc.v1, va.v0, vb.v1); //c.im += a.re * b.im

        svst2(p32_all, (float32_t*)&c[i], vc);
    }
}
```

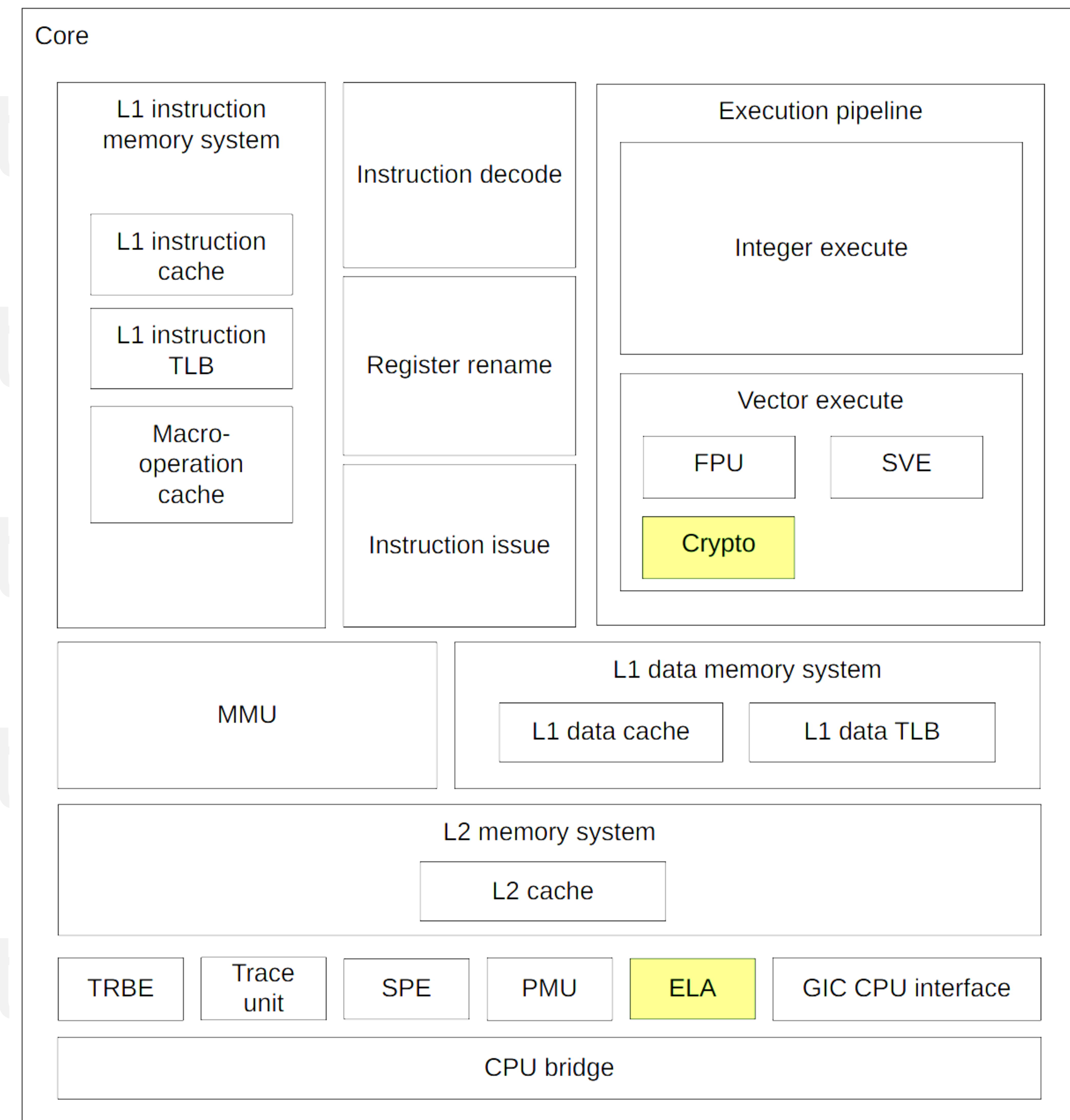
Stony Brook University

Neoverse V2 Core Details

Arm Neoverse V2 Core

Overview

- Arm® Neoverse™ V2 Core Technical Reference Manual
- Arm Neoverse V2 implements Armv9.0-A architecture
  - Extends Armv8.[0-5]-A architecture
  - 128-bit vector length SVE & SVE2
  - 128-bit vector length ASIMD (a.k.a., NEON)
  - Learn the architecture - Understanding the Armv8.x and Armv9.x extensions
- Separate L1 data and instruction caches
  - L1 instruction memory system
    - 64KB, 4-way set associative, 64B cache line
    - Fully associative L1 instruction TLB, support for {4,16,64}KB and 2MB page sizes
    - 1536-entry, 4-way skewed associative L0 MOP cache
    - Dynamic branch predictor
  - L1 data memory system
    - 64KB, 4-way set associative, 64B cache line
    - Fully associative L1 data TLB, support for {4,16,64}KB page sizes and {2,512}MB block sizes
- Private, unified data and instruction L2 cache
  - 1-2MB (1MB for Grace), 8-way set associative



Stony Brook University
NVIDIA

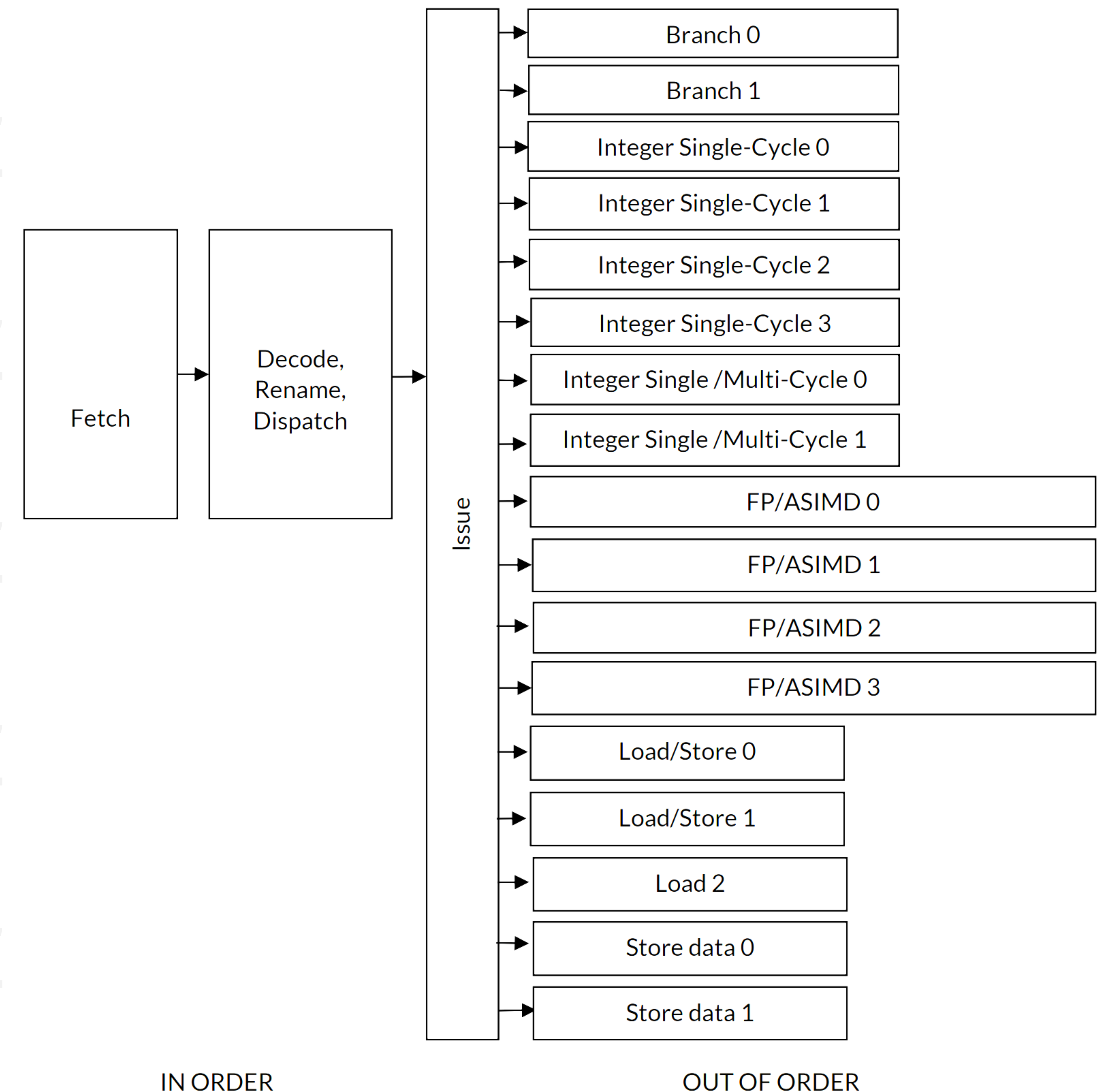
Arm Neoverse V2 Core

Core pipeline & general information

- Arm Neoverse V2 Core Software Optimization Guide
- Up to 8 instructions decoded into internal MOPs
- Each MOPs can be split into 2 uOPs
- Total of 16 uOP with some limitations
  - 2, 4 or 6 uOPs depending on the pipelines used
- FP/ASIMD pipelines process FP, NEON, SVE and SVE2
- Some instructions might use more than one pipeline
  - e.g., gather load will use a Load and FP/ASIMD pipelines

- Instruction latency/throughput is variable
  - 4x Scalar/NEON/SVE FP64 FMA per cycle

- Intrinsics – Arm Developer
  - Includes intrinsic to assembly code information



Stony Brook University
NVIDIA

Arm Neoverse V2 Core

Write streaming mode

- Avoids polluting cache when writing big chunks of data with no reads
  - e.g., memset to initialize data structures
- Enabled when core detects when a full cache line has been written before the line fill completes
  - i.e., you write to cache faster than you load cache lines with memory locations being written to
- While in streaming mode
  - Loads behave as normal
  - Writes lookup cache, if miss, write to L2 cache or system memory instead of generating a line fill
- Streaming mode is disabled when
  - System detects a cacheable write burst that is not a full cache line
  - There is a load operation from the same line that is being written to L2 cache
- Examples
  - **a**[i] = **b**[i] + **c**[i] will cause the core to go into streaming mode
  - **a**[i] = **a**[i] + **c**[i] will keep the core in non-streaming mode

Stony Brook University

Arm Neoverse V2 Core

Special considerations for compilers and low-level library developers

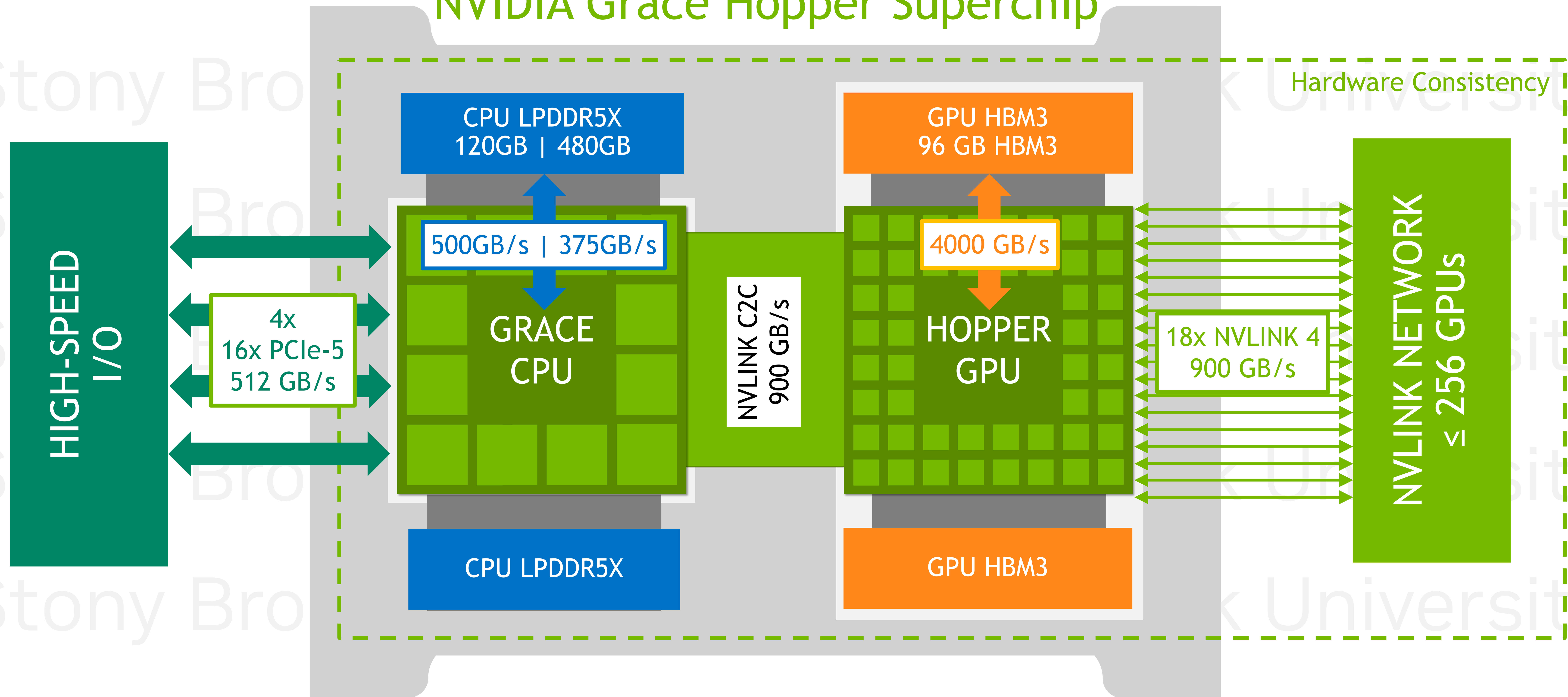
- Memory alignment
  - Generally, no penalty for unaligned memory accesses
  - Penalty can occur when
    - Crossing cache line (64B) boundary
    - Quad word loads that are not 4B aligned
    - Stores that cross a 32B boundary
- Memory routines
  - memcpy
    - Unroll loop to include multiple loads and store ops per iteration
    - Align loads to 16B boundary when possible
    - Use non-writeback forms of LDP/STP (load/store pair of registers)
  - memset
    - Unroll loop to include multiple stores per iteration
    - For memset to 0, use DC ZVA instructions instead of store (might be not recommended for small memsets)
- Branch instruction alignment
  - Avoid placing more than four branch instructions within an aligned 32B instruction memory region
- AES encryption/decryption
  - At least 8 data chunks should be interleaved to achieve full pipeline utilization
  - Pairs of dependent AESE/ASEMC and AESD/AESIMC instructions should be adjacent in code and using the same destination register

Stony Brook University

Optimizing for Coherent Memory

Grace Hopper Superchip

GPU can access CPU memory at CPU memory speeds



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

Stony Brook University

Stony Brook University Stony Brook University

GPU Memory is Visible to the Operating System

Standard operating system commands work on the GPU

```
nvidia@localhost: ~
nvidia@localhost:~$ numactl -H
available: 9 nodes (0-8)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 5
2 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71
node 0 size: 490310 MB
node 0 free: 475425 MB
node 1 cpus:
node 1 size: 96768 MB
node 1 free: 96767 MB
node 2 cpus:
node 2 size: 0 MB
node 2 free: 0 MB
node 3 cpus:
node 3 size: 0 MB
node 3 free: 0 MB
node 4 cpus:
node 4 size: 0 MB
node 4 free: 0 MB
node 5 cpus:
node 5 size: 0 MB
node 5 free: 0 MB
node 6 cpus:
node 6 size: 0 MB
node 6 free: 0 MB
node 7 cpus:
node 7 size: 0 MB
node 7 free: 0 MB
node 8 cpus:
node 8 size: 0 MB
node 8 free: 0 MB
node distances:
node   0   1   2   3   4   5   6   7   8
  0:  10  80  80  80  80  80  80  80  80
  1:  80  10 255 255 255 255 255 255 255
  2:  80 255  10 255 255 255 255 255 255
  3:  80 255 255  10 255 255 255 255 255
  4:  80 255 255 255  10 255 255 255 255
  5:  80 255 255 255 255  10 255 255 255
  6:  80 255 255 255 255 255  10 255 255
  7:  80 255 255 255 255 255 255  10 255
  8:  80 255 255 255 255 255 255 255  10
nvidia@localhost:~$ free -g
              total        used        free      shared  buff/cache   available
Mem:            573          11         558           1           3         541
Swap:             0           0           0
nvidia@localhost:~$
```

CPU

GPU

MIG


```
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 5
2 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71
node 0 size: 490310 MB
node 0 free: 475425 MB
node 1 cpus:
node 1 size: 96768 MB
node 1 free: 96767 MB
```


Hopper GPU appears to the OS as a NUMA node with no CPU cores

Total system memory capacity is CPU (480GB) + GPU (96GB)

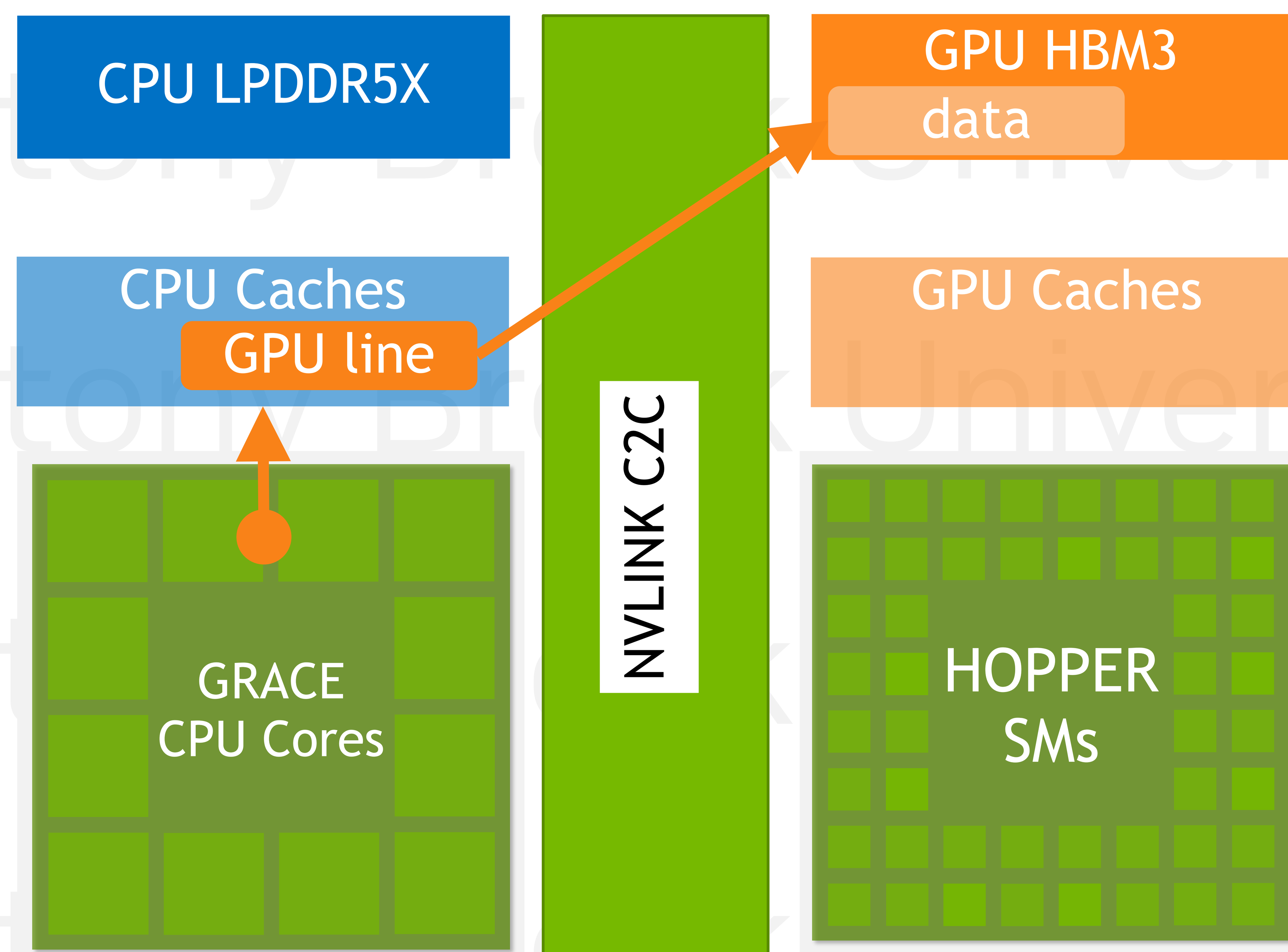

```
nvidia@localhost:~$ free -g
              total        used        free      shared  buff/cache   available
Mem:            573          11         558           1           3         541
Swap:             0           0           0
```


```
nvidia@localhost:/home/nvidia/jlinford/mt-dgemm/src$ numactl -m1 ./mt-dgemm.nvpl 5000 1 1 1 0 1 1
Matrix size input by command line: 5000
Repeat multiply 1 times.
```

Can use `numactl` to put CPU application data in GPU memory

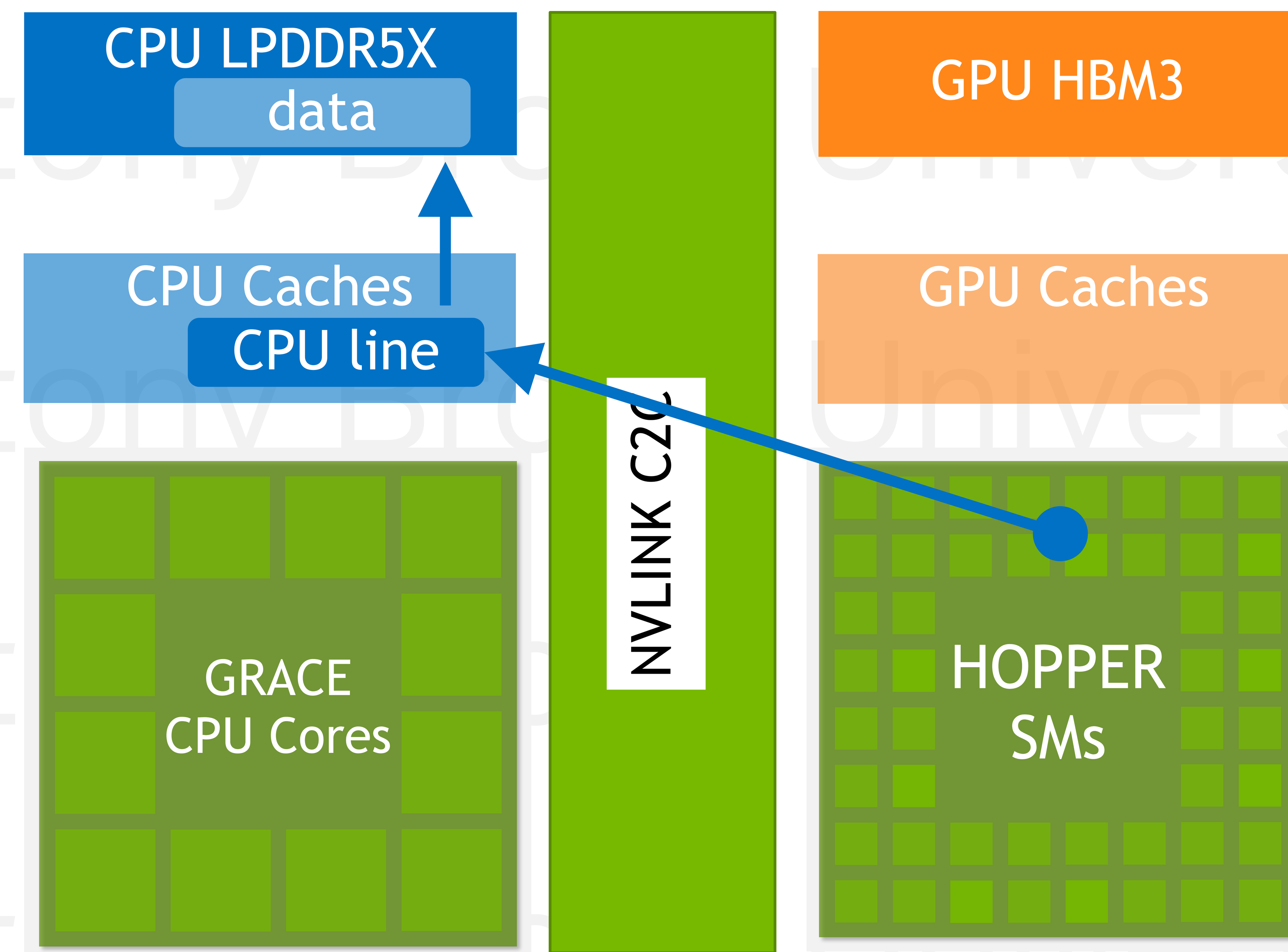
Global Access to All Data

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



Grace directly reading Hopper's memory

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

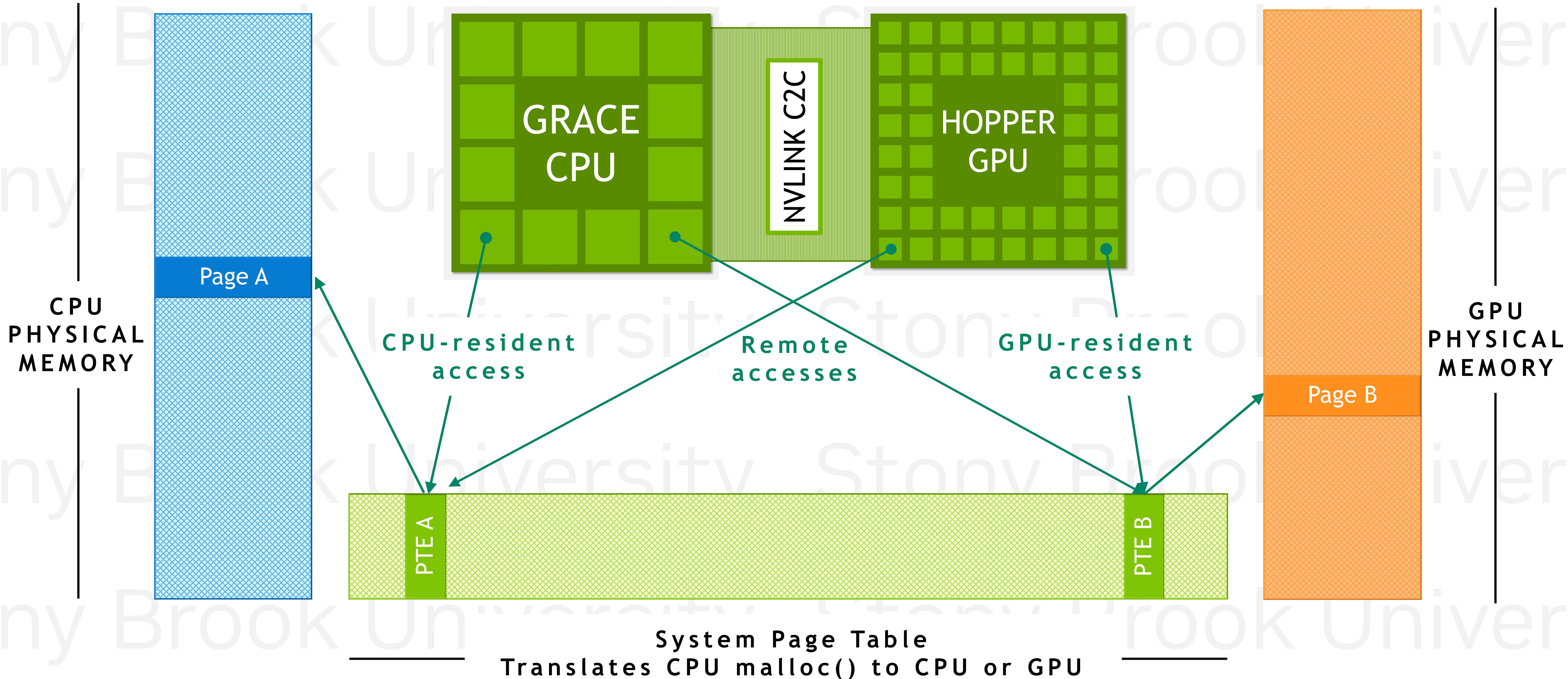


Hopper directly reading Grace's memory

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

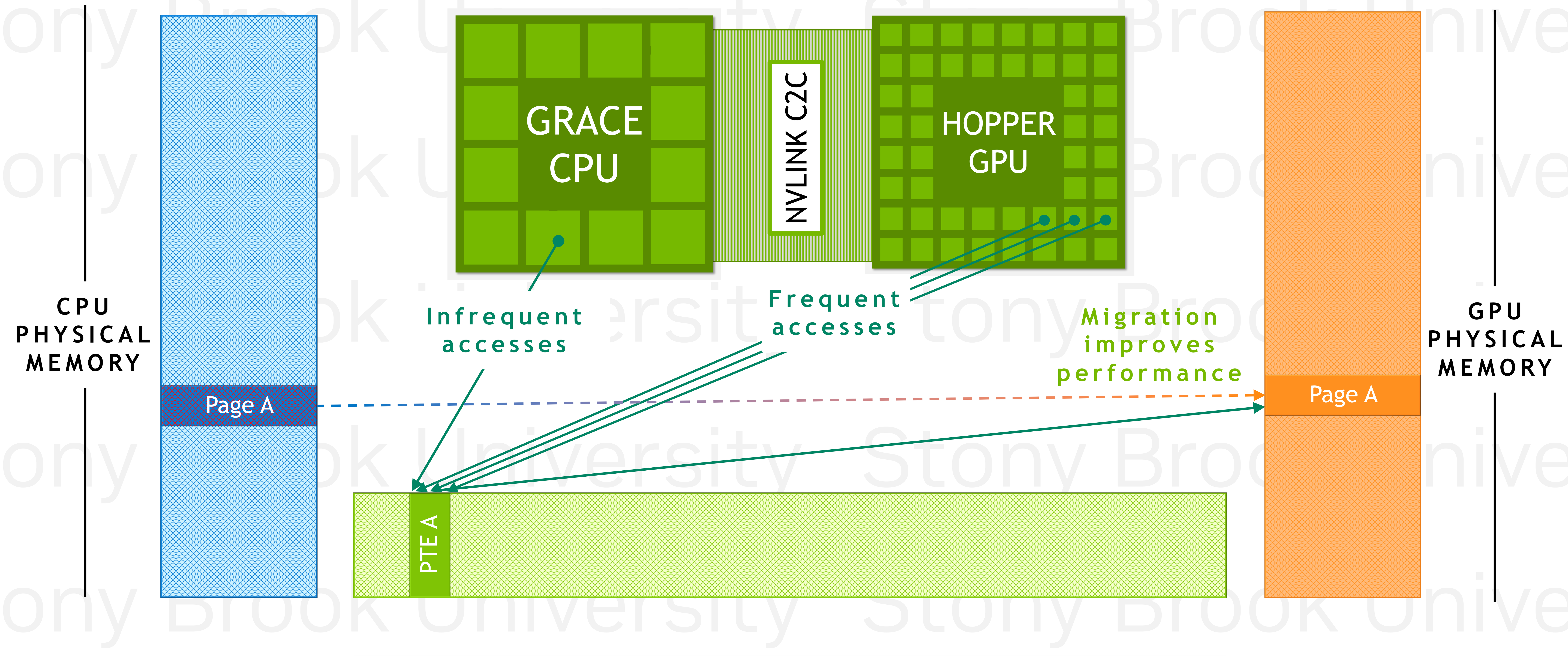
Stony Brook University

Grace Hopper Address Translation Service (ATS)



Stony Brook University

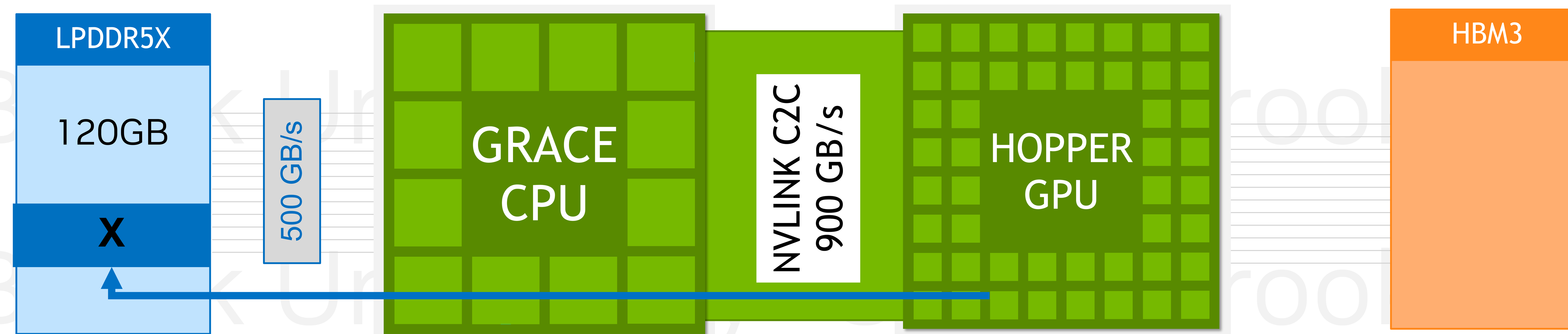
Grace Hopper Automatic Page Migration



Stony Brook University

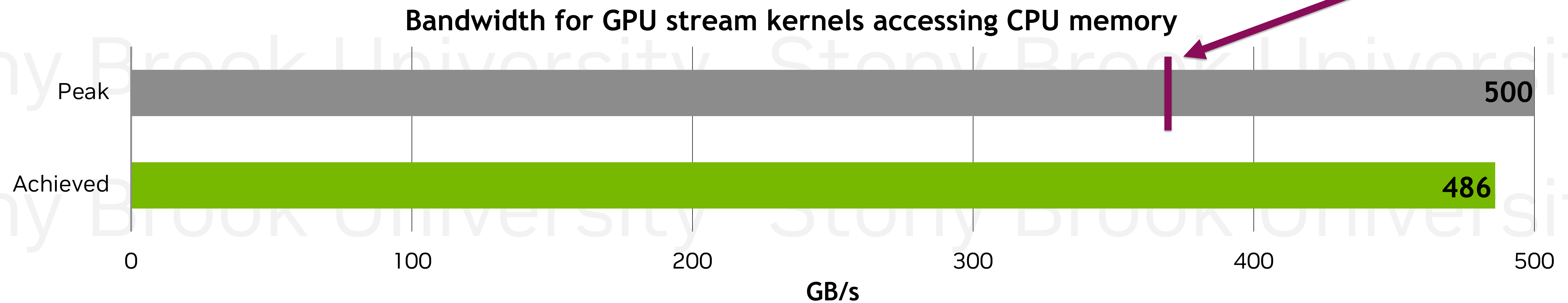
NVIDIA

High Bandwidth Memory Access & Automatic Data Migration



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

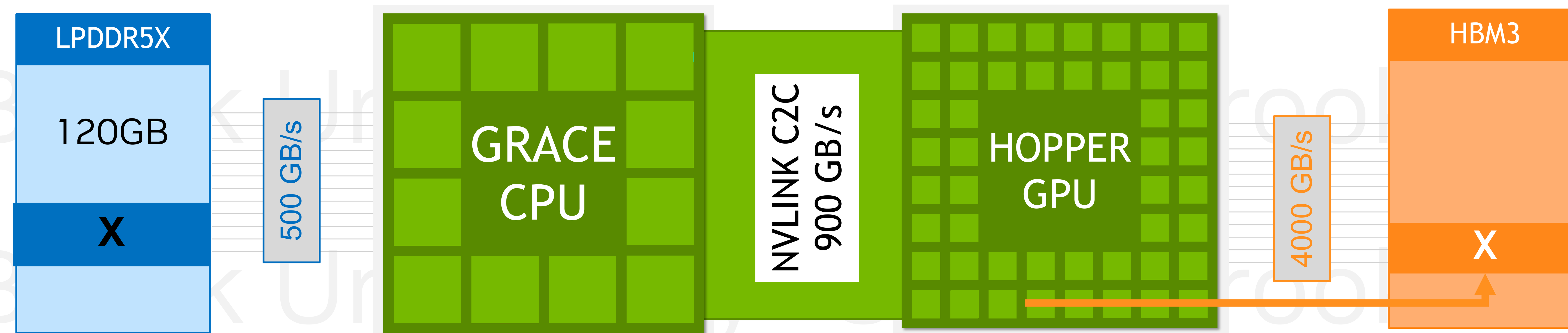
Note: 480GB CPUs have 25% lower LPDDR5x bandwidth



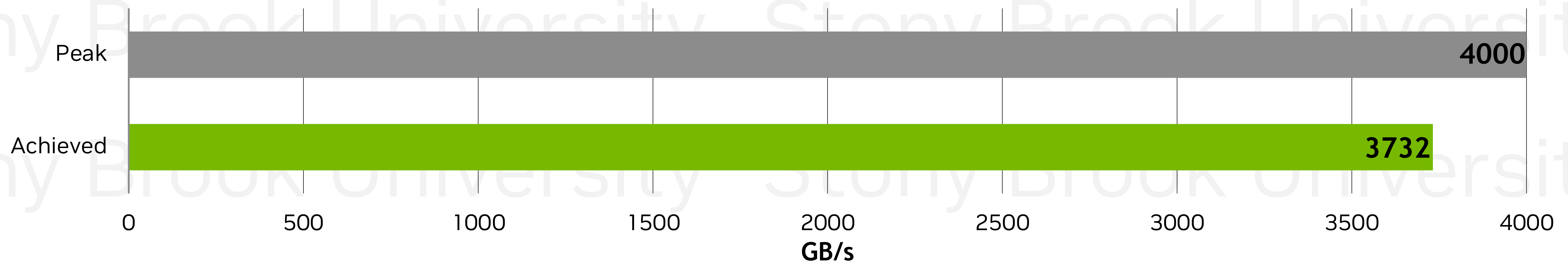
Stony Brook University

NVIDIA

High Bandwidth Memory Access & Automatic Data Migration



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



Stony Brook University

NVIDIA

Memory Allocators Impact Data Placement and Movement

CUDA 12.4

	cudaMalloc	cudaMallocManaged	Malloc/mmap
Placement	GPU		
Page size	2MB		
Which processor can access ?	GPU		
How does access happen ?	GPU MMU		
What can the driver do for my app ?	-		
What can I do for my app ?	Think how to potential leverage coherent systems		

Stony Brook University

Memory Allocators Impact Data Placement and Movement

CUDA 12.4

	cudaMalloc	cudaMallocManaged	Malloc/mmap
Placement	GPU	First touch	
Page size	2MB	hybrid, 64K for CPU and 2MB for GPU	
Which processor can access ?	GPU	Both CPU and GPU	
How does access happen ?	GPU MMU	Fault on first access and move page [2]	
What can the driver do for my app ?	-	Fault or Access counters [2]	
What can I do for my app ?	Think how to potential leverage coherent systems	Use CUDA APIs to manage memory	

1. mTHP will allow 2MB page sizes Linux kernel 6.9 patch or hugeTLB
2. Unless Memadvise with preferred location and setAccessedBy are set
3. Pages don't migrate back to CPU due to lack of access counters

Stony Brook University

Memory Allocators Impact Data Placement and Movement

CUDA 12.4

	`cudaMalloc`	`cudaMallocManaged`	`System (malloc/mmap/…)`
Placement	GPU	First touch	First touch
Page size	2MB	hybrid, 64K for CPU and 2MB for GPU	64K (system page) [1]
Which processor can access ?	GPU	Both CPU and GPU	Both CPU and GPU
How does access happen ?	GPU MMU	Fault on first access and move page [2]	Direct access over C2C using ATS [2]
What can the driver do for my app ?	-	Fault or Access counters [2]	Using access counter to migrate memory CPU -> GPU [3]
What can I do for my app ?	Think how to potential leverage coherent systems	Use CUDA APIs to manage memory	Use CUDA APIs to manage memory

1. mTHP will allow 2MB page sizes Linux kernel 6.9 patch or hugeTLB
2. Unless Memadvise with preferred location and setAccessedBy are set
3. Pages don't migrate back to CPU due to lack of access counters

Stony Brook University

NVIDIA

Grace UVM Migration Enhancements: CUDA C++ & CUDA Fortran

Maximum portable performance to NVIDIA HW out-of-the-box & without any changes

- **No programming model changes!**
  → **No new APIs**
  → **No changes to existing APIs**
  → **No source code changes**

- Unified Memory
  - Available on *most* platforms supported by CUDA 12.x: GH, P9+V100, PCIe x86 & Arm, etc.
  - Same Unified Memory Programming Model for all platforms: "memory accesses just work" + "hints".

- Unified Memory **Hints**
  - *Hints* only impact performance, not results.
  - `cudaMemAdvise` hints: PreferredLocation, AccessedBy.
  - `cudaMemPrefetch` hints: prefetch to NUMA node.
  - Works with `cudaMallocManaged` memory on all supported Unified Memory platforms
  - Work with system allocated memory (e.g. `malloc`) on Grace Hopper and systems with HMM

CUDA Explicit Memory Allocators

Memory	Placement	Access-based Migration	Accessible From	
			CPU	GPUs
System-allocated (malloc, mmap)	First-touch (GPU | CPU)	✅	✅	✅
CUDA managed (cudaMallocManaged)		✅	✅	✅
CUDA device memory (cudaMalloc)	GPU	❌	❌	✅
CUDA host memory (cudaMallocHost)	CPU	❌	✅	✅
…and many others: interprocess, virtual, fabric, …				

CUDA Unified Memory Hints

```
cudaMemAdvise(ptr, nbytes, advice, device);
```

Advices	PreferredLocation	AccessedBy	ReadMostly

Devices	GPU id	CPU	CPU Numa Node

```
cudaMemPrefetchAsync(ptr, nbytes, destination, stream);
```

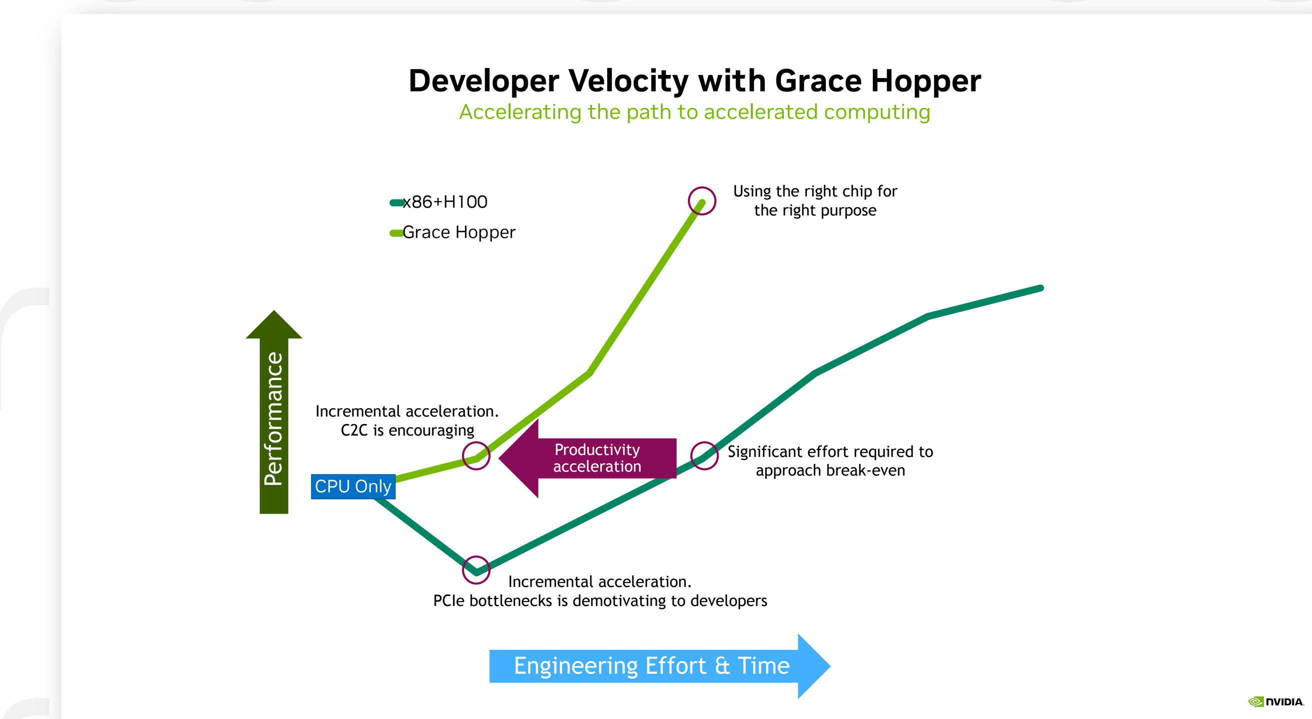
Destinations	GPU id	CPU	CPU Numa Node

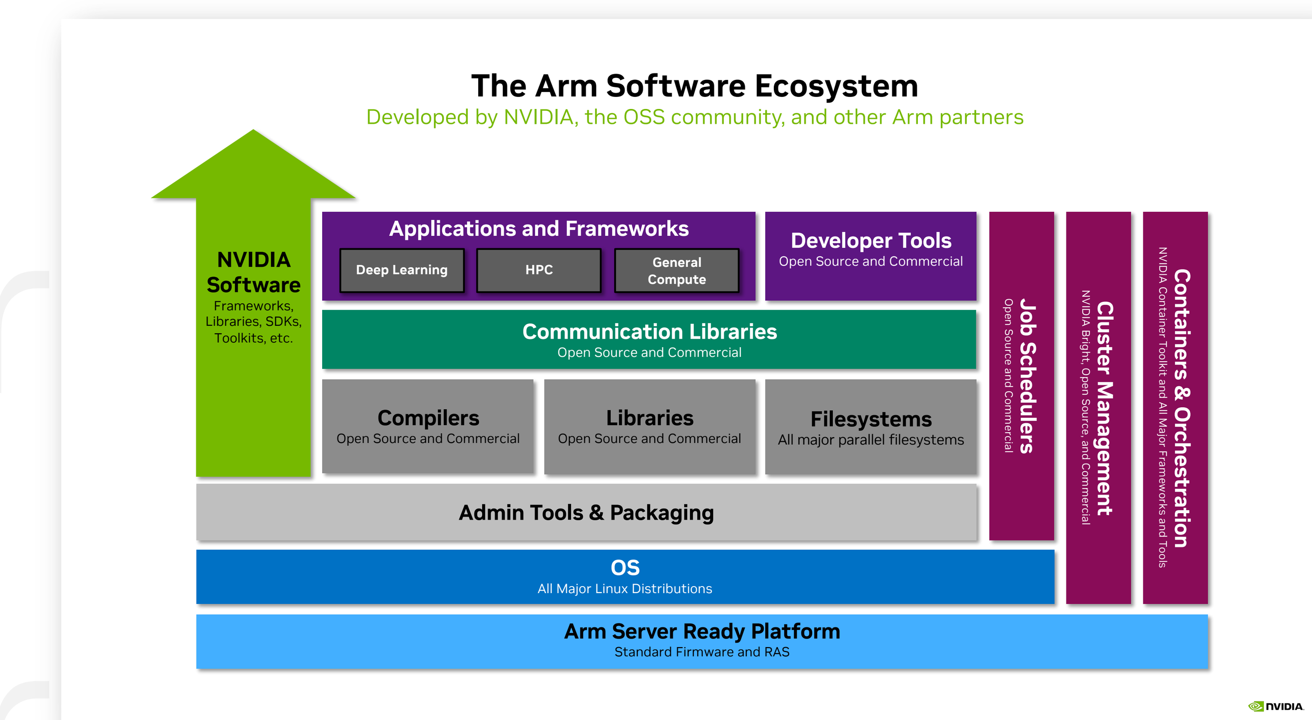
Stony Brook University

Wrap Up

Conclusions

- NVIDIA Grace Hopper improves developer velocity
- Use the compilers, libraries, and tools you already use
  - … as long as they are standards-compliant and multi-platform
- Expect software to work; expect software to perform well
- Tune Grace CPU performance with compilers and libraries
  - Update compiler flags
  - Use compiler autovectorization
  - Use de-facto standard library APIs like Netlib BLAS and FFTW
- Tune for coherent memory with memory allocators and CUDA UM hints
  - Use CUDA-managed memory to unlock coherent memory capability
  - Use CUDA unified memory hints to improve performance





Grace UVM Migration Enhancements: CUDA C++ & CUDA Fortran

Maximum portable performance to NVIDIA HW out-of-the-box & without any changes

- No programming model changes!
  → No new APIs
  → No changes to existing APIs
  → No source code changes
- Unified Memory
  - Available on *most* platforms supported by CUDA 12.x: GH, P9+V100, PCIe x86 & Arm, etc.
  - Same Unified Memory Programming Model for all platforms: "memory accesses just work" + "hints".
- Unified Memory **Hints**
  - *Hints* only impact performance, not results.
  - `cudaMemAdvise` hints: PreferredLocation, AccessedBy.
  - `cudaMemPrefetch` hints: prefetch to NUMA node.
  - Works with `cudaMallocManaged` memory on all supported Unified Memory platforms
  - Work with system allocated memory (e.g. `malloc`) on Grace Hopper and systems with HMM

CUDA Explicit Memory Allocators

Memory	Placement	Access-based Migration	Accessible From	
			CPU	GPUs
System-allocated (malloc, mmap)	First-touch (GPU | CPU)	✓	✓	✓
CUDA managed (cudaMallocManaged)		✓	✓	✓
CUDA device memory (cudaMalloc)	GPU	✗	✗	✓
CUDA host memory (cudaMallocHost)	CPU	✗	✓	✓
…and many others: interprocess, virtual, fabric, …				

CUDA Unified Memory Hints

```
cudaMemAdvise(ptr, nbytes, advice, device);
```

Advices	PreferredLocation	AccessedBy	ReadMostly

Devices	GPU id	CPU	CPU Numa Node

```
cudaMemPrefetchAsync(ptr, nbytes, destination, stream);
```

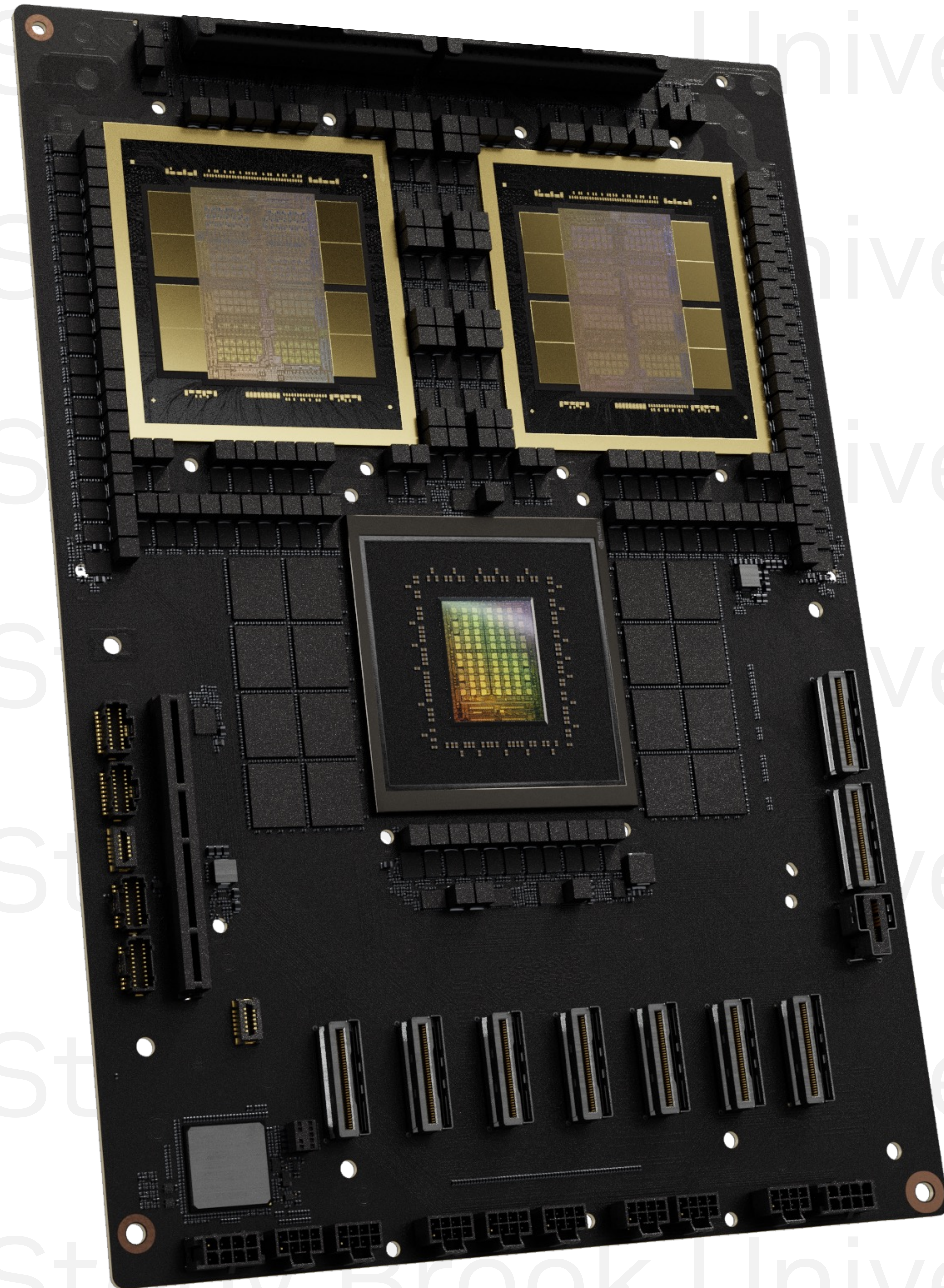
Destinations	GPU id	CPU	CPU Numa Node

NVIDIA 45

Stony Brook University

GB200 SUPERCHIP

Optimized for Supercomputer-Scale Science



72 Grace CPU Arm cores

40 PetaFLOPS FP4 AI Inference

20 PetaFLOPS FP8 AI Training

16 TB/s of GPU memory bandwidth

864 GB Fast Memory

Stony Brook University

DGX GB200

Delivers New Unit of Compute

DGX GB200
- 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

Stony Brook University

Stony Brook University Stony Brook University

Grace CPU Benchmarking Guide

https://nvidia.github.io/grace-cpu-benchmarking-guide/

nvidia.github.io/grace-cpu-benchmarking-guide/index.html

NVIDIA Grace CPU Benchmarking Guide

How to Use This Guide

This guide is for end users and application developers working with the NVIDIA® Grace CPU who want to achieve optimal performance for key benchmarks and applications (workloads). It includes procedures, sample code, reference performance numbers, recommendations, and technical best practices directly related to the NVIDIA Grace CPU. Following the instructions given in this guide will help you realize the best possible performance for your particular system.

This guide is a is a living document and frequently updated with the latest recommendations, so it is best read online at https://nvidia.github.io/grace-cpu-benchmarking-guide/. If you want to help improve the guide, you can create a Github issue at https://github.com/NVIDIA/grace-cpu-benchmarking-guide/issues/new.

Understanding Workload Performance

Workload performance depends on many aspects of the system, so the measured performance of your system **may be different** from the performance figures presented here. These figures are provided as guidelines and should not be interpreted as performance expectations or targets. Do not use this guide for platform validation.

The guide is divided into the following sections:

- Platform Configuration: This section helps you tune your system for benchmarking. The instructions will help optimize the platform configuration.
- Foundational Benchmarks: After checking the platform configuration, this section helps you complete a sanity check and confirm that the system is healthy.
- Common Benchmarks: This section has information about the industry-recognized benchmarks and mini-apps that represent the performance of key workloads.
- Applications: This section has information about maximizing the performance of full applications.
- Developer Best Practices: This section has general best practices information to develop for NVIDIA Grace.

The sections can be read in any order, but we strongly recommend you begin by tuning and sanity checking your platform.

Copyright © 2023 NVIDIA CORPORATION & AFFILIATES. All rights reserved.

nvidia.github.io/grace-cpu-benchmarking-guide/platform/software.html

NVIDIA Grace CPU Benchmarking Guide

Benchmarking Software Environment

Begin by installing all available software updates, for example, `sudo apt update && sudo apt upgrade` on Ubuntu. Use the command `ld --version` to check that GNU binutils version is 2.38 or later. For best performance, GCC should be at version 12.3 or later. `gcc --version` will report the GCC version.

Many Linux distributions provide packages for GCC 12 compilers that can be installed alongside the system GCC. For example, `sudo apt install gcc-12` on Ubuntu. See your Linux distribution's instructions for installing and using various GCC versions. In case your distribution does not provide these packages, or you are unable to install them, instructions for building and installing GCC are provided below.

A Recommended Software Stack

This guide shows a variety of compilers, libraries, and tools. Suggested minimum versions of the major software packages used in this guide are shown below, but any recent version of these tools will work well on NVIDIA Grace. Installation instructions for each package are provided in the associated link.

Package	Minimum Version	Link
NVIDIA HPC SDK	23.11	https://developer.nvidia.com/hpc-sdk
NVIDIA Clang for Grace	16.0.5	https://developer.nvidia.com/grace/clang
GNU Binutils	2.41	https://ftp.gnu.org/gnu/binutils/binutils-2.41.tar.xz
GNU GCC	12.3	https://ftp.gnu.org/gnu/gcc/gcc-12.3.0/gcc-12.3.0.tar.xz
UCX	1.14.1	https://github.com/openucx/ucx/releases/tag/v1.14.1
OpenMPI	4.1.5	https://www.open-mpi.org/software/ompi/v4.1/
MVAPICH2	2.3.7	https://mvapich.cse.ohio-state.edu/download/mvapich/mv2/mvapich2-2.3.7-1.tar.gz
PAPI	7.1.0	https://icl.utk.edu/papi/
Arm Compiler for Linux	23.04	https://developer.arm.com/downloads/-/arm-compiler-for-linux

Building and Installing GCC 12.3 from Source

Prefer Linux Distribution GCC 12 Packages

Many Linux distributions provide packages for GCC 12 compilers that can be installed alongside the

nvidia.github.io/grace-cpu-benchmarking-guide/benchmarks/HPL/index.html

NVIDIA Grace CPU Benchmarking Guide

High Performance Linpack (HPL)

The NVIDIA HPC-Benchmarks provides a multiplatform (x86 and aarch64) container image based on NVIDIA Optimized Frameworks container images that includes NVIDIA's HPL benchmark. HPL-NVIDIA solves a random dense linear system in double precision arithmetic on distributed-memory computers and is based on the netlib HPL benchmark. Please visit the NVIDIA HPC-Benchmarks page in the NGC Catalog for detailed instructions.

The HPL-NVIDIA benchmark uses the same input format as the standard Netlib HPL benchmark. Please see the Netlib HPL benchmark for getting started with the HPL software concepts and best practices.

Downloading and using the container

The container image works well with Singularity, Docker, or Pyxis/Enroot. For a general guide on pulling and running containers, see Running A Container in the NVIDIA Containers For Deep Learning Frameworks User's Guide. For more information about using NGC, refer to the NGC Container User Guide.

Running the benchmarks

The script `hpl-aarch64.sh` can be invoked on a command line or through a slurm batch script to launch HPL-NVIDIA for NVIDIA Grace CPU. As of HPC-Benchmarks 23.10, `hpl-aarch64.sh` accepts the following parameters:

- Required parameters:
  - `--dat path`: Path to `HPL.dat` input file
- Optional parameters:
  - `--cpu-affinity <string>`: A colon-separated list of cpu index ranges
  - `--mem-affinity <string>`: A colon separated list of memory indices
  - `--ucx-affinity <string>`: A colon separated list of UCX devices
  - `--ucx-tls <string>`: UCX transport to use
  - `--exec-name <string>`: HPL executable file

Several sample input files are available in the container at `/workspace/hpl-linux-aarch64`.

Run with Pyxis/Enroot

To run HPL-NVIDIA on two nodes of NVIDIA Grace CPU using your custom HPL.dat file:

```
CONT='nvcr.io#nvidia/hpc-benchmarks:23.10'
```

nvidia.github.io/grace-cpu-benchmarking-guide/benchmarks/NPB/index.html

NVIDIA Grace CPU Benchmarking Guide

NAS Parallel Benchmarks

The NAS Parallel Benchmarks (NPB) are a small set of programs designed to help evaluate the performance of parallel supercomputers. The NPB 1 benchmarks are derived from computational fluid dynamics (CFD) applications and consist of five kernels and three pseudo-applications. Problem sizes in NPB are predefined and indicated as different classes. Reference implementations of NPB are available in commonly-used programming models like MPI and OpenMP.

Building the Benchmarks

1. Download and unpack the NPB source code from nas.nasa.gov:

```
wget https://www.nas.nasa.gov/assets/npb/NPB3.4.2.tar.gz
tar xvzf NPB3.4.2.tar.gz
cd NPB3.4.2/NPB3.4-OMP
```

2. Create the `make.def` file to configure the build for NVIDIA HPC compilers:

```
cat > config/make.def <<'EOF'
FC = nvfortran
FLINK = $(FC)
F_LIB =
F_INC =
FFLAGS = -O3 -mp
FLINKFLAGS = $(FFLAGS)
CC = nvc
CLINK = $(CC)
C_LIB = -lm
C_INC =
CFLAGS = -O3 -mp
CLINKFLAGS = $(CFLAGS)
UCC = gcc
BINDIR = ../bin
RAND = randi8
WTIME = wtime.c
EOF
```

3. Create the `suite.def` file to build all benchmarks with the D problem size:

```
cat > config/suite.def <<'EOF'
```

nvidia.github.io/grace-cpu-benchmarking-guide/platform/index.html

NVIDIA Grace CPU Benchmarking Guide

Platform Configuration

Before benchmarking, you should check whether the platform configuration is optimal for the target benchmark. The optimal configuration can vary by benchmark, but there are some common high-level settings of which you should be aware. Most platforms benefit from the settings shown below.

Info

Refer to the NVIDIA Grace Performance Tuning Guide and the platform-specific documentation at https://docs.nvidia.com/grace/ for instructions on how to tune your platform for optimal performance.

Warning

The settings shown on this page are intended to maximize system performance and may affect system security.

Linux Kernel

The following Linux kernel command line options are recommended for performance:

- `init_on_alloc=0`: Do not fill newly allocated pages and heap objects with zeroes by default.
- `acpi_power_meter.force_cap_on=y`: Enable ACPI power meter and with power capping.
- `numa_balancing=disable`: Disable automatic NUMA balancing.

You can confirm these command line options are set by reading `/proc/cmdline`:

```
cat /proc/cmdline | tr ' ' '\n'
```

```
BOOT_IMAGE=/boot/vmlinuz-6.2.0-1012-nvidia-64k
root=UUID=76c84c6d-a59f-4a8d-903e-4cb9ef69b970
ro
rd.driver.blacklist=nouveau
nouveau.modeset=0
earlycon
module_blacklist=nouveau
acpi_power_meter.force_cap_on=y
numa_balancing=disable
init_on_alloc=0
preempt=none
```

nvidia.github.io/grace-cpu-benchmarking-guide/applications/WRF/index.html

NVIDIA Grace CPU Benchmarking Guide

Weather Research and Forecasting Model

The Weather Research and Forecasting (WRF) Model is a next-generation mesoscale numerical weather prediction system designed for both atmospheric research and operational forecasting needs. It features two dynamical cores, a data assimilation system, and a software architecture facilitating parallel computation and system extensibility.

Arm64 is supported by the standard WRF distribution as of WRF 4.3.3. The following is an example of how to perform the standard procedure to build and execute on NVIDIA Grace. See http://www2.mmm.ucar.edu/wrf/users/download/get_source.html for more details.

Install WRF

Initial Configuration

Verify that the most recent NVIDIA HPC SDK is available in your environment. The simplest way to do this is to load the `nvhpc` module file.

```
module load nvhpc
```

```
nvc --version
```

```
nvc 23.7-0 linuxarm64 target on aarch64 Linux -tp neoverse-v2
NVIDIA Compilers and Tools
Copyright (c) 2022, NVIDIA CORPORATION & AFFILIATES.  All rights reserved.
```

NVIDIA HPC SDK includes optimized MPI compilers and libraries, so you'll also have the appropriate MPI compilers in your path:

```
$ which mpirun
/opt/nvidia/hpc_sdk/Linux_aarch64/23.7/comm_libs/mpi/bin/mpirun

$ mpicc -show
nvc -I/opt/nvidia/hpc_sdk/Linux_aarch64/23.7/comm_libs/openmpi/openmpi-3.1.5/include -Wl,-rpath -Wl,$OR
```

Also verify that your GCC compiler is version 12.3 or later.

```
gcc --version
```

nvidia.github.io/grace-cpu-benchmarking-guide/developer/atomics.html

NVIDIA Grace CPU Benchmarking Guide

Locks, Synchronization, and Atomics

Efficient synchronization is critical to achieving good performance in applications with high thread counts, but synchronization is a complex and nuanced topic. See below for a high level overview (refer to the Synchronization Overview and Case Study on Arm Architecture whitepaper from Arm for more information).

The Arm Memory Model

One of the most significant differences between Arm and the x86 CPUs is their memory model. The Arm architecture has a weak memory model that allows for more compiler and hardware optimization to boost system performance. This differs from the x86 architecture Total Store Order (TSO) model. Different memory models can cause low-level codes (for example, drivers) to function well on one architecture but encounter performance problem or failure on the other.

Note

The unique features of the Arm memory model are only relevent if you are writing low level code, such as assembly language. Most software developers will not be affected by a change in memory model.

The details about Arm's memory model are below the application level and will be completely invisible to most users. If you are writing in a high-level language such as C, C++, or Fortran, you do not need to know the nuances of Arm's memory model. The one exception to this general rule is code that uses boutique synchronization constructs instead of standard best practices, for example, using `volatile` as a means of thread synchronization.

Deviating from established standards or ignoring best practices results in code that is almost guaranteed to be broken. It should be rewritten using system-provided locks, conditions, etc. and the `stdatomic` tools. (Refer to https://github.com/ParRes/Kernels/issues/611 for an example of this type of bug.)

Arm is not the only architecture that uses a weak memory model. If your application already runs on CPUs that are not x86-based, you might encounter fewer bugs that are related to the weak memory model. Specifically, if your application has been ported to a CPU implementing the POWER architecture, for example, IBM POWER9, the application will work on the Arm memory model.

Large-System Extension (LSE) Atomic Instructions

All server-class Arm64 processors, such as NVIDIA Grace, have support for the Large-System Extension (LSE), which was first introduced in Armv8.1. LSE provides low-cost atomic operations that can improve system

nvidia.github.io/grace-cpu-benchmarking-guide/developer/languages/c-c++...

NVIDIA Grace CPU Benchmarking Guide

C/C++ on NVIDIA Grace

There are many C/C++ compilers available for NVIDIA Grace including:

- NVIDIA HPC Compiler
- Clang/LLVM
- GCC
- Arm Compiler for Linux (ACfL)
- HPE Cray Compiler

Selecting a Compiler

The compiler you use depends on your application's needs. If in doubt, try the NVIDIA HPC Compiler first because this compiler will always have the most recent updates and enhancements for Grace. If you prefer Clang, NVIDIA provides builds of Clang for NVIDIA Grace that are supported by NVIDIA and certified as a CUDA host compiler. GCC, ACfL, and HPE Cray Compilers also have their own advantages. As a general strategy, default to an NVIDIA-provided compiler and fall back to a third-party as needed.

When possible, use the latest compiler version that is available on your system. Newer compilers provide better support and optimizations for Grace, and when using newer compilers, many codes will demonstrate significantly better performance.

Recommended Compiler Flags

The NVIDIA HPC Compilers accept PGI flags and many GCC or Clang compiler flags. These compilers include the NVFORTRAN, NVC++, and NVC compilers. They work with an assembler, linker, libraries, and header files on your target system, and include a CUDA toolchain, libraries and header files for GPU computing. Refer to the NVIDIA HPC Compiler's User's Guide for more information. The freely NVIDIA HPC SDK is the best way to quickly get started with the freely available NVIDIA HPC Compiler.

Most compiler flags for GCC and Clang/LLVM operate the same on Arm64 as on other architectures except for the `-mcpu` flag. On Arm64, this flag both specifies the appropriate architecture and the tuning strategy. It is generally better to use `-mcpu` instead of `-march` or `-mtune` on Grace. You can find additional details in this presentation given at Stony Brook University.

CPU	Flag	GCC version	LLVM verison
NVIDIA Grace	-mcpu=neoverse-v2	12.3+	16+
Ampere Altra	-mcpu=neoverse-n1	9+	10+
Any Arm64	-mcpu=native	9+	10+

NVIDIA