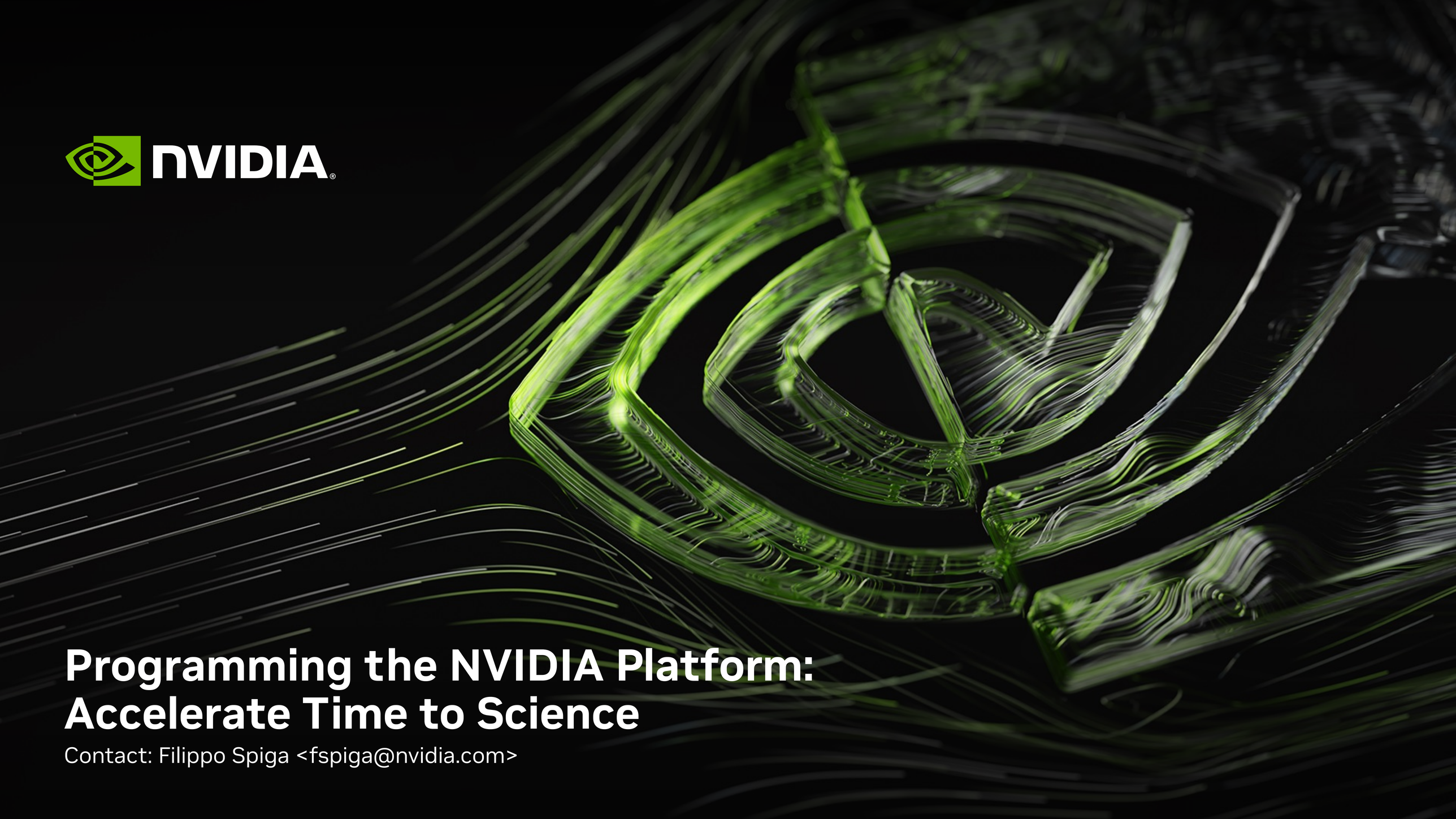


Programming the NVIDIA Platform: Accelerate Time to Science

Contact: Filippo Spiga <fspiga@nvidia.com>

10 years of evolution in GPU hardware

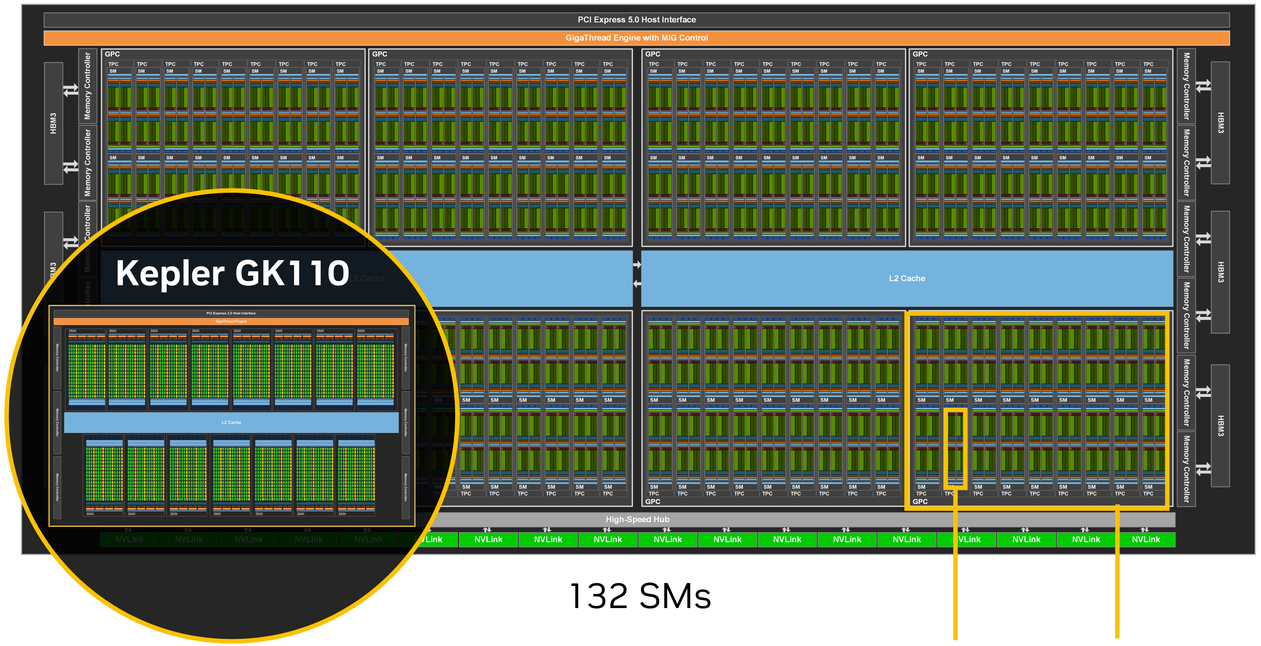
Kepler GK110 GPU (2012)

15 SMs

SM

3.52 TFLOPS single
1.17 TFLOPS double

Hopper H100 GPU (2022)



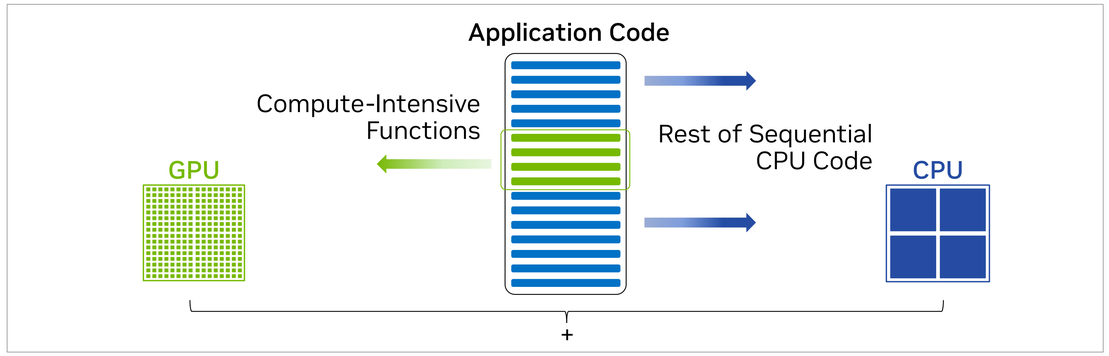
132 SMs

SM **GPC**

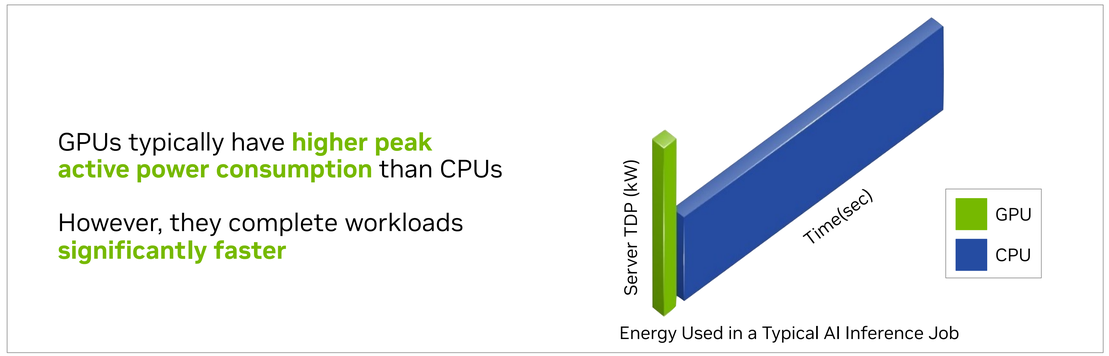
67 TFLOPS single **[19x]**
34 TFLOPS double **[29x]**
67 TFLOPS double with TC **[57x]**

Why Accelerated Computing is Energy Efficient

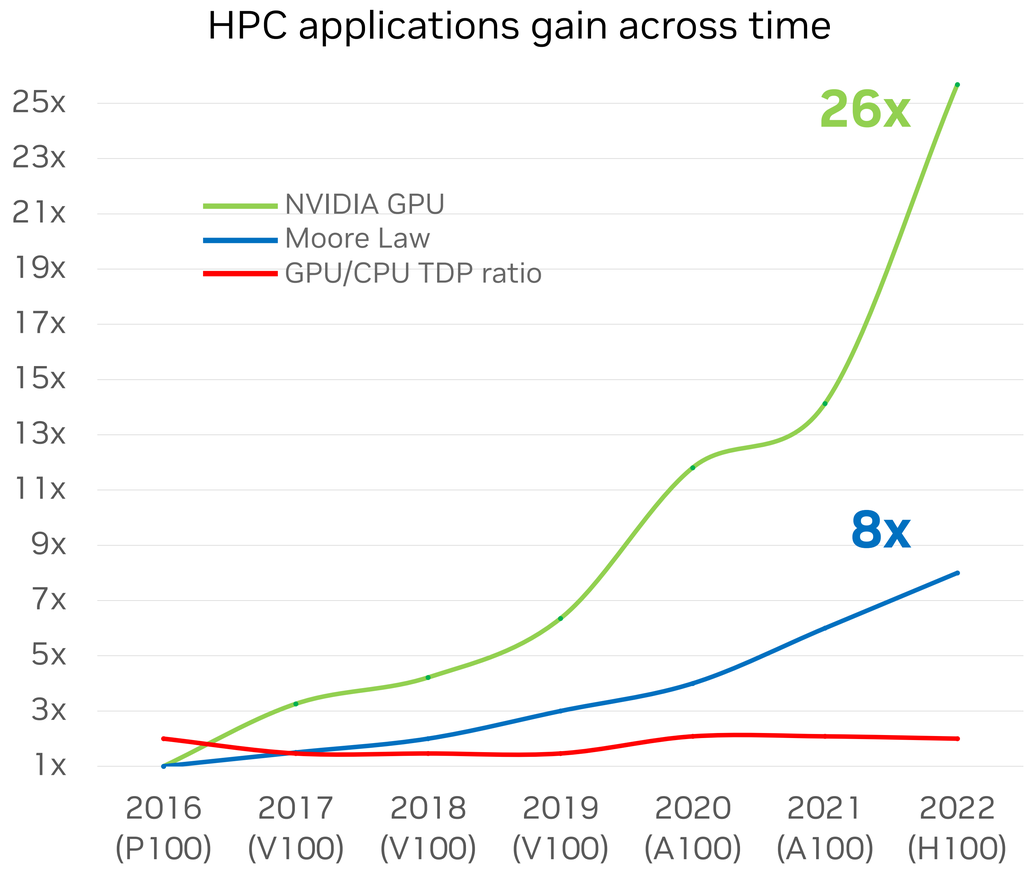
Hardware, Software, & Networking to Optimize Performance & Efficiency



Moving Compute Intense Sections to the GPU



Consume Less Overall Energy

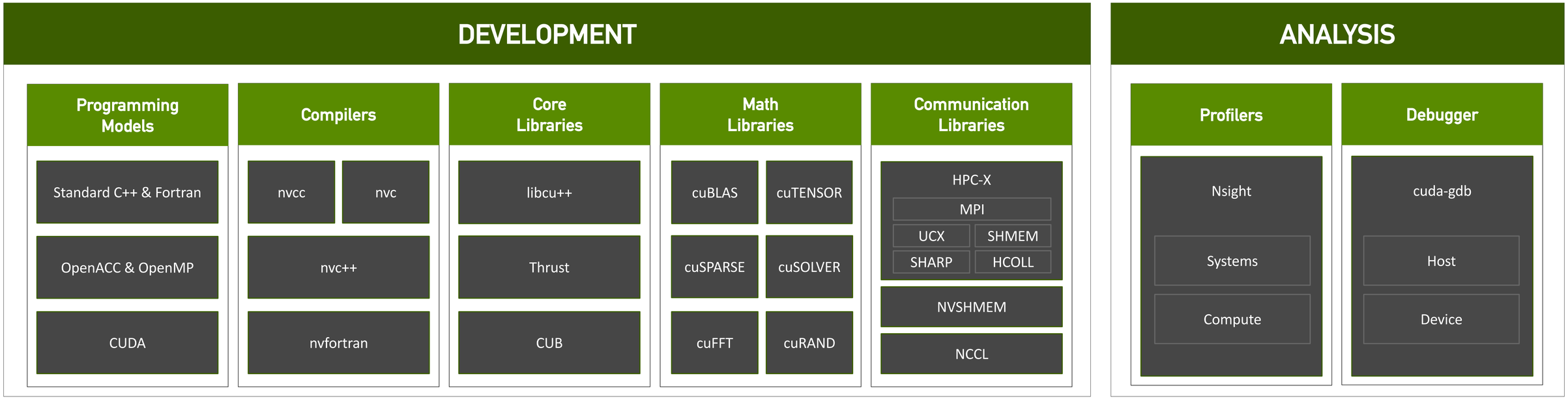


Center Panel: Geometric mean of application speedups vs. P100 | benchmark applications | Amber [PME-Cellulose_NVE], Chroma [HMC], GROMACS [ADH Dodec], MILC [Apex Medium], NAMD [stmv_nve_cuda], PyTorch (BERT Large Fine Tuner), Quantum Espresso [AUSURF112-jR]; TensorFlow [ResNet-50], VASP 6 [Si Huge],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

Contents

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

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

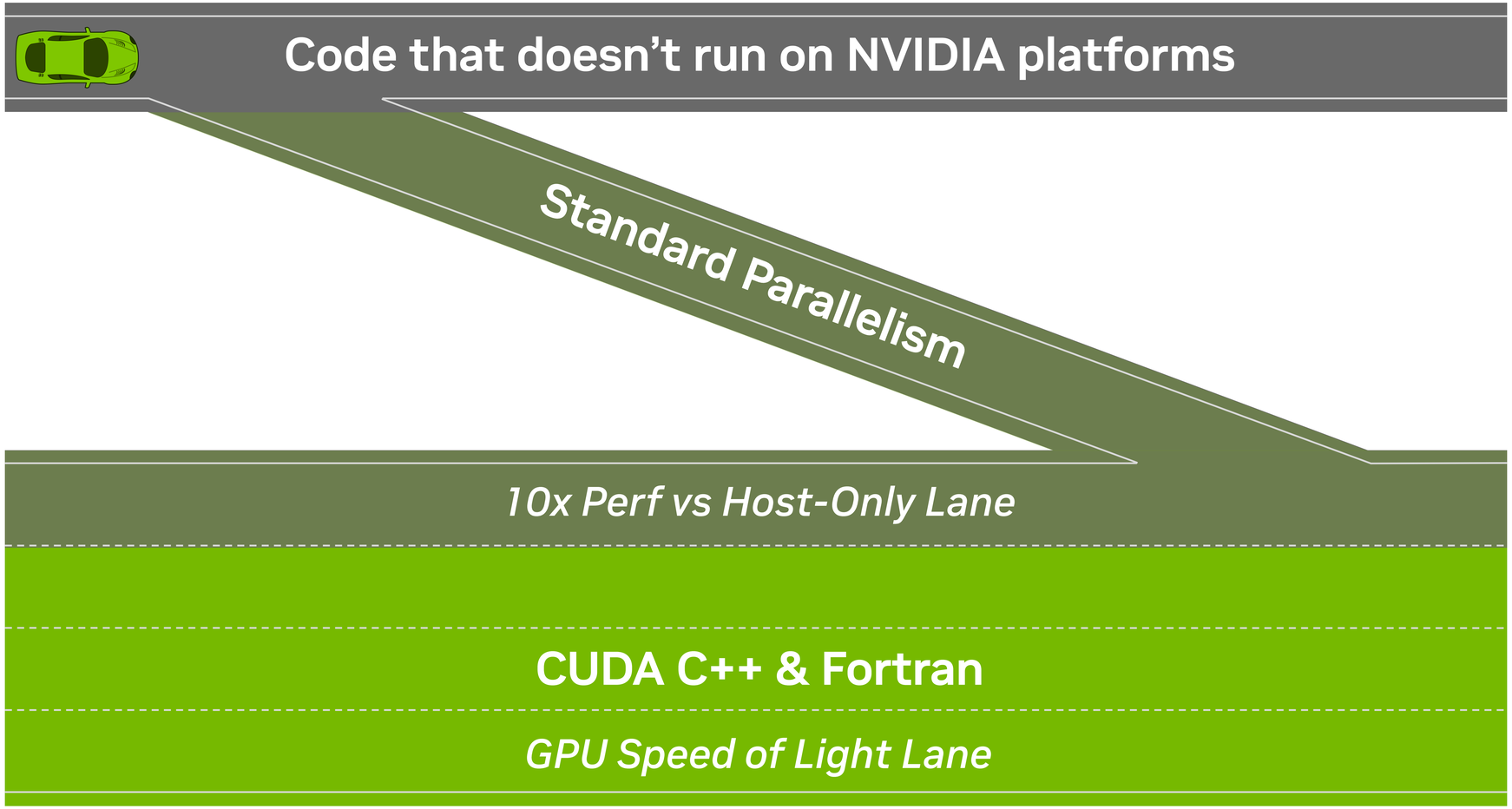
```
__global__
void saxpy(int n, float a,
            float *x, float *y) {
  int i = blockIdx.x*blockDim.x +
          threadIdx.x;
  if (i < n) y[i] += a*x[i];
}

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

Scientists Need On-Ramps

Promote Parallelism, not Heterogeneity



Accelerated Standard Languages

Parallel performance for wherever your code runs

ISO C++

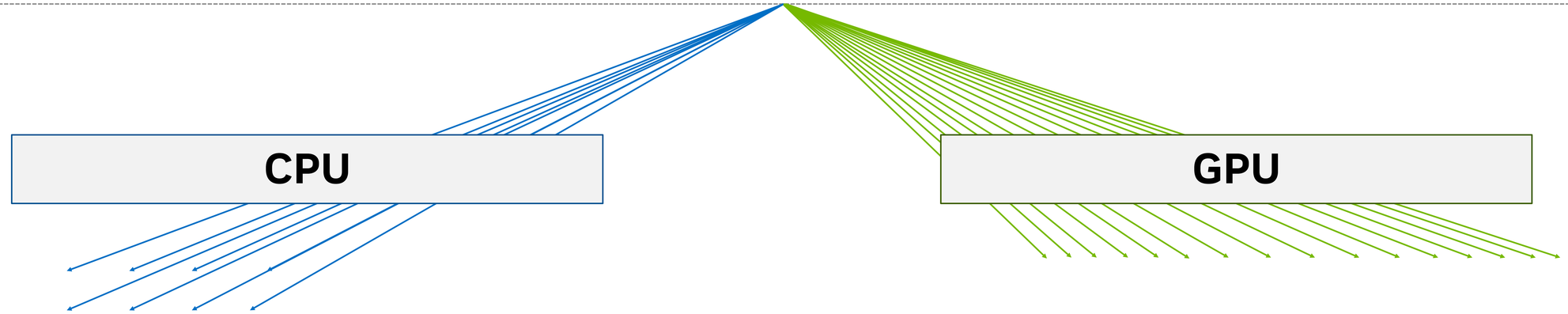
```
std::transform(par, x, x+n, y,
   y,[=](float x, float y){
      return y + a*x;
    }
);
```

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
```

ISO C++

C++17 & C++20

Parallel Algorithms

- Parallel and vector concurrency

Forward Progress Guarantees

- Extend the C++ execution model for accelerators

Memory Model Clarifications

- Extend the C++ memory model for accelerators

Ranges

- Simplifies iterating over a range of values

Scalable Synchronization Library

- Express thread synchronization that is portable and scalable across CPUs and accelerators

Preview support coming to NVC++

C++23

std::mdspan/mdarray

- HPC-oriented multi-dimensional array abstractions.
- **Preview Available Now**

Range-Based Parallel Algorithms

- Improved multi-dimensional loops

Extended Floating Point Types

- First-class support for formats new and old: std::float16_t/float64_t

And Beyond

Std::Execution

- Simplify launching and managing parallel work across CPUs and accelerators
- **Preview Available Now**

Linear Algebra

- C++ standard algorithms API to linear algebra
- Maps to vendor optimized BLAS libraries
- **Preview Available Now**

The Cost of Synchrony

Current C++ Parallel Algorithms are synchronous, incurring extra overheads when launched on a GPU

```
// C++17 Parallel Algorithms are Synchronous
for (int step = 0; step < n_steps; step++) {
  std::for_each(par_unseq, cells_begin, cells_end, update_h);
  std::for_each(par_unseq, cells_begin, cells_end, update_e);
}
```

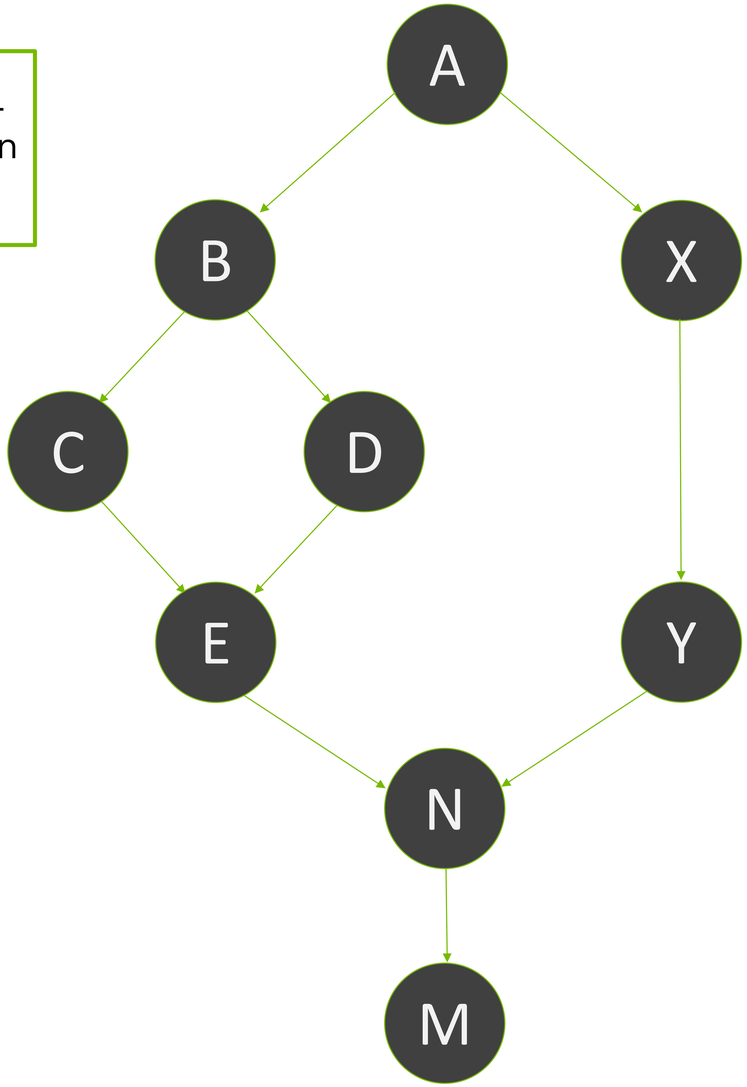
```
// CUDA enables asynchronously enqueuing kernels
for (int step = 0; step < n_steps; step++) {
  kernel<<<blocks, threads, 0, stream>>>(n_cells, update_h);
  kernel<<<blocks, threads, 0, stream>>>(n_cells, update_e);
}
cudaStreamSynchronize(stream);
```

Std::Execution

Lazily build large graphs of computation

```
sender auto async_graph(sender auto s) {
    auto A = s | std::then(printer{'A’}) | std::split();
    auto B = A | std::then(printer{'B'}) | std::split();
    auto C = B | std::then(printer{'C'});
    auto D = B | std::then(printer{'D'});
    auto E = std::when_all(C, D) | std::then(printer{'E'});
    auto X = A | std::then(printer{'X'})
               | std::then(printer{'Y'});
    return std::when_all(E, X);
```

Express a device-agnostic execution graph.




```
cuda::execution::scheduler gpu_scheduler;
sender auto work = std::schedule(gpu_scheduler) | async_graph | async_algo;
auto [r] = std::this_thread::sync_wait(work).value();
assert(r == 55);
```

Apply a specific scheduler to execute that graph.

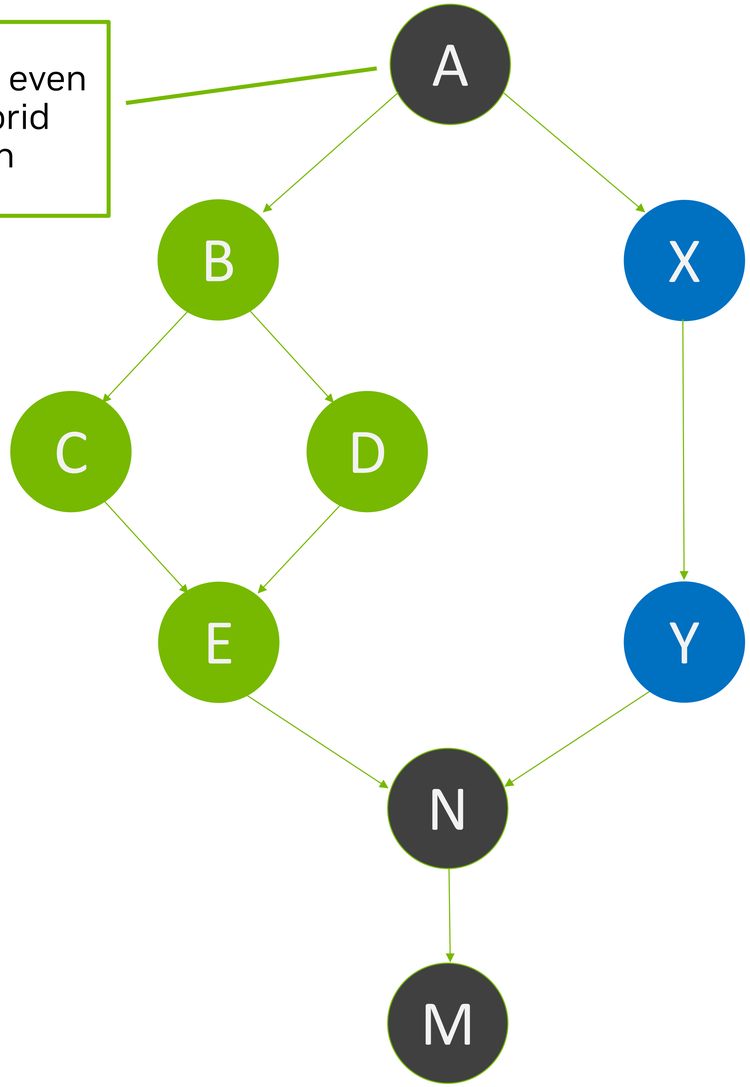
Wait for the graph to complete.

Std::Execution

Lazily build large graphs of computation

```
sender auto async_graph(sender auto s) {
    auto A = s | std::then(printer{'A’}) | std::split();
    auto B = A | std::then(printer{'B'}) | std::split();
    auto C = B | std::then(printer{'C'});
    auto D = B | std::then(printer{'D'});
    auto E = std::when_all(C, D) | std::then(printer{'E'});
    auto X = A | std::then(printer{'X'})
               | std::then(printer{'Y'});
    return std::when_all(E, X);
```

Graphs could even include hybrid execution



Instantiate a custom scheduler.

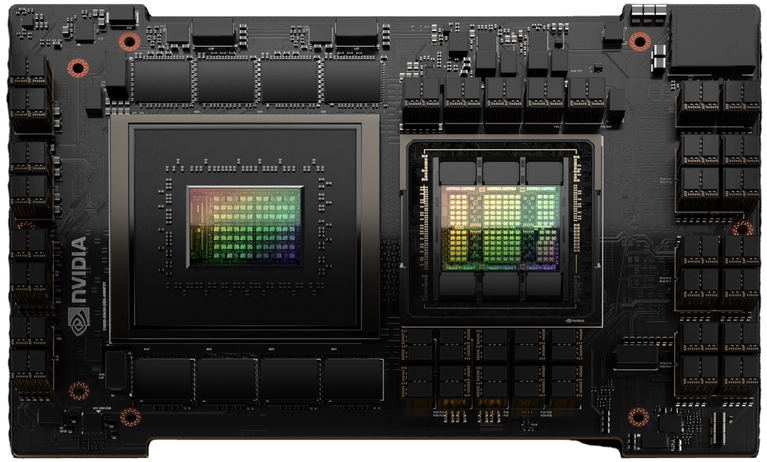
```
my_hybrid_scheduler hybrid_scheduler;
sender auto work = std::schedule(hybrid_scheduler) | async_graph | async_algo;
auto [r] = std::this_thread::sync_wait(work).value();
assert(r == 55);
```

Programming the NVIDIA Superchip

NVIDIA Grace for HPC & AI Infrastructure

Grace Hopper Superchip

Giant Scale AI & HPC



1000W max
(including LPDDR5x and HBM3 memory)

Grace CPU Superchip

CPU Computing

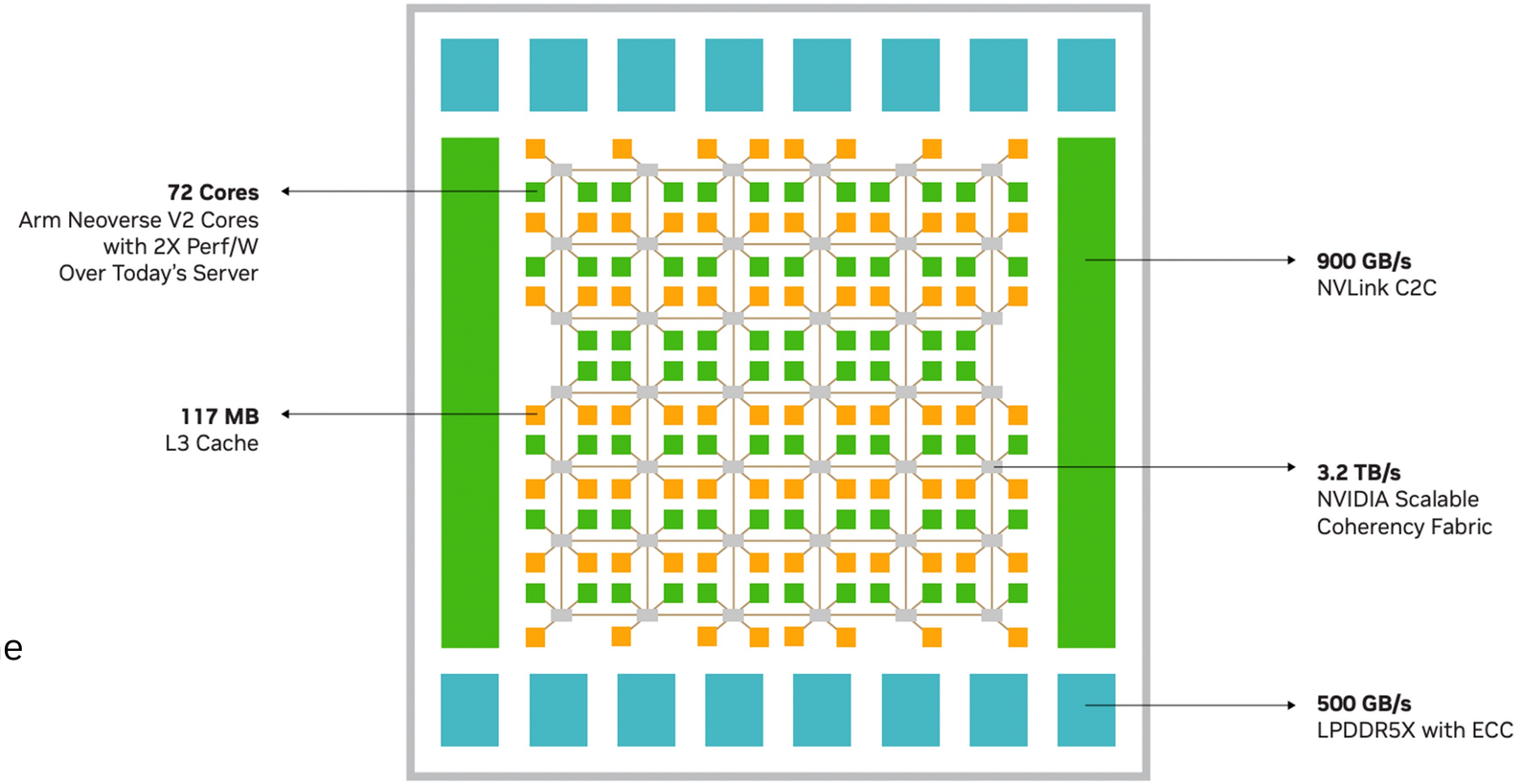
500W max
(including LPDDR5x memory)

Designed from the ground-up to be a Superchip, always paired

GRACE IS A COMPUTE & DATA MOVEMENT ARCHITECTURE

NVIDIA Scalable Coherency Fabric (SCF) and distributed cache design

- Up to 512GB of LPDDR5X memory
  - **32 channels**
  - **Up to 546 GB/s of memory BW**
  - **Competitive power/perf**
- NVIDIA Scalable Coherency Fabric
  - 3,225.6 GB/s bi-section BW
  - **117MB of distributed L3 cache**
  - Scalable to 72+ cores per die
  - Background data movement via Cache Switch Network
- Supports up to 4-die coherency over Coherent NVLINK

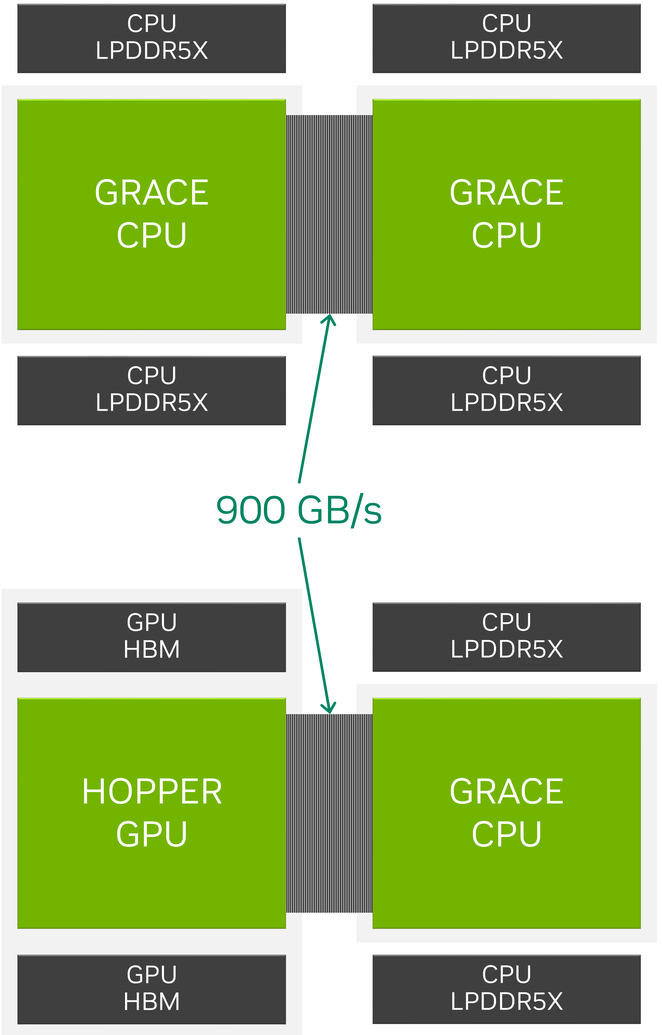


Example possible fabric topology for illustrative purposes

NVLINK-C2C

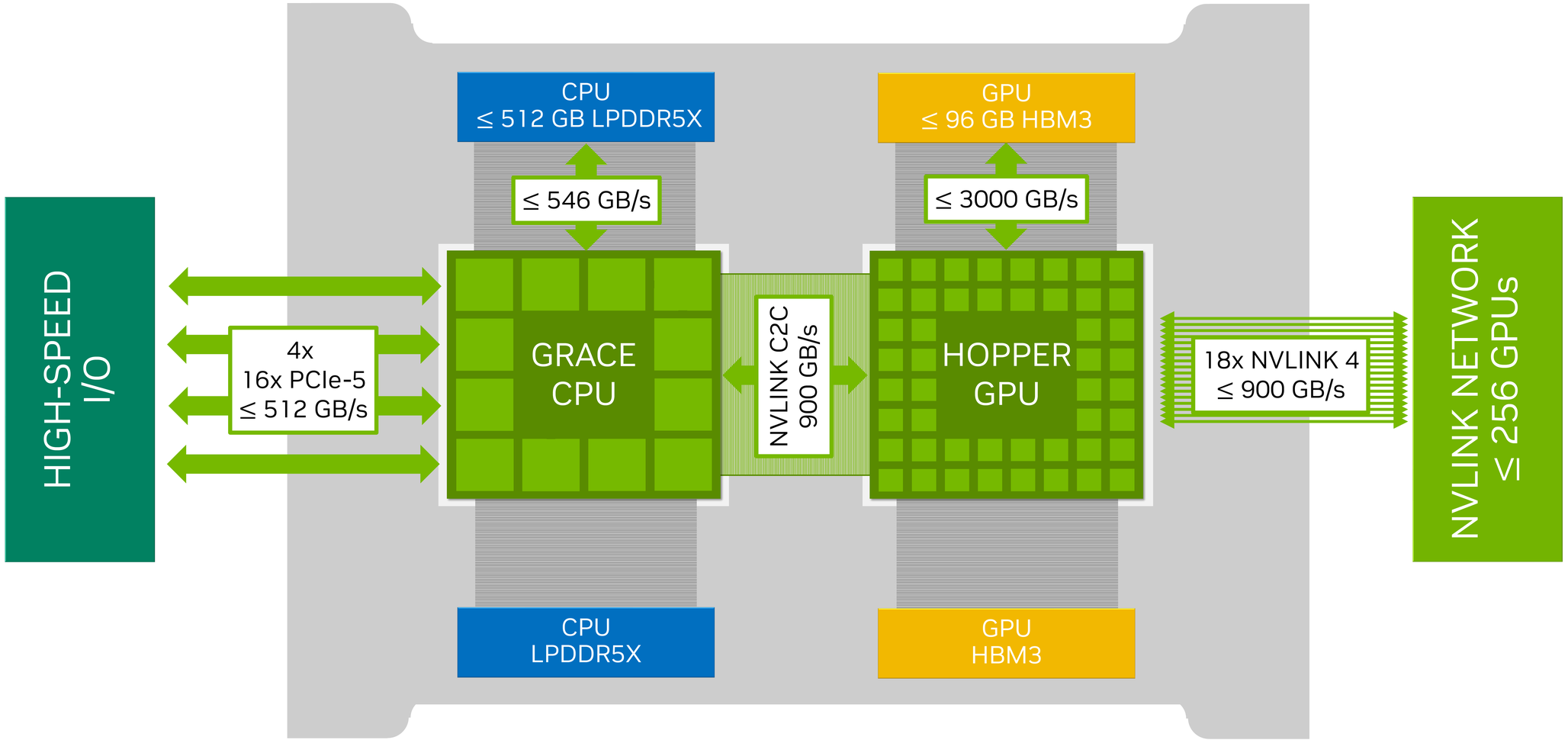
High Speed Chip to Chip Interconnect

- Used to create the Grace Hopper, and Grace Superchips
  - **Native atomics, including standard C++ atomic support**
  - Enables coherency
- Up to 900 GB/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**
- Unified Memory with shared page tables
  - **Shared CPU and GPU virtual address space (AST)**



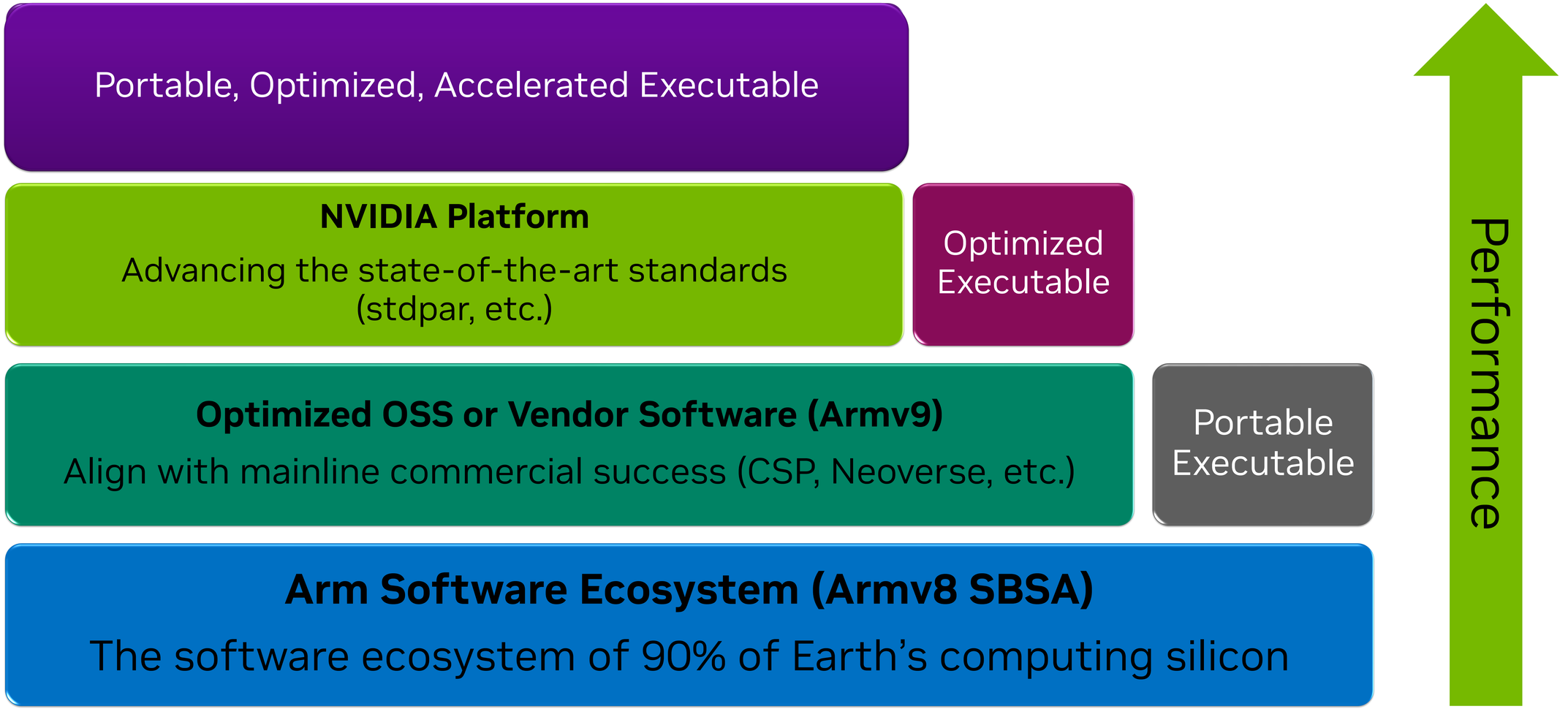
Grace Hopper Superchip Platform

Speeds and Feeds



Grace Software Ecosystem is Built on Standards

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

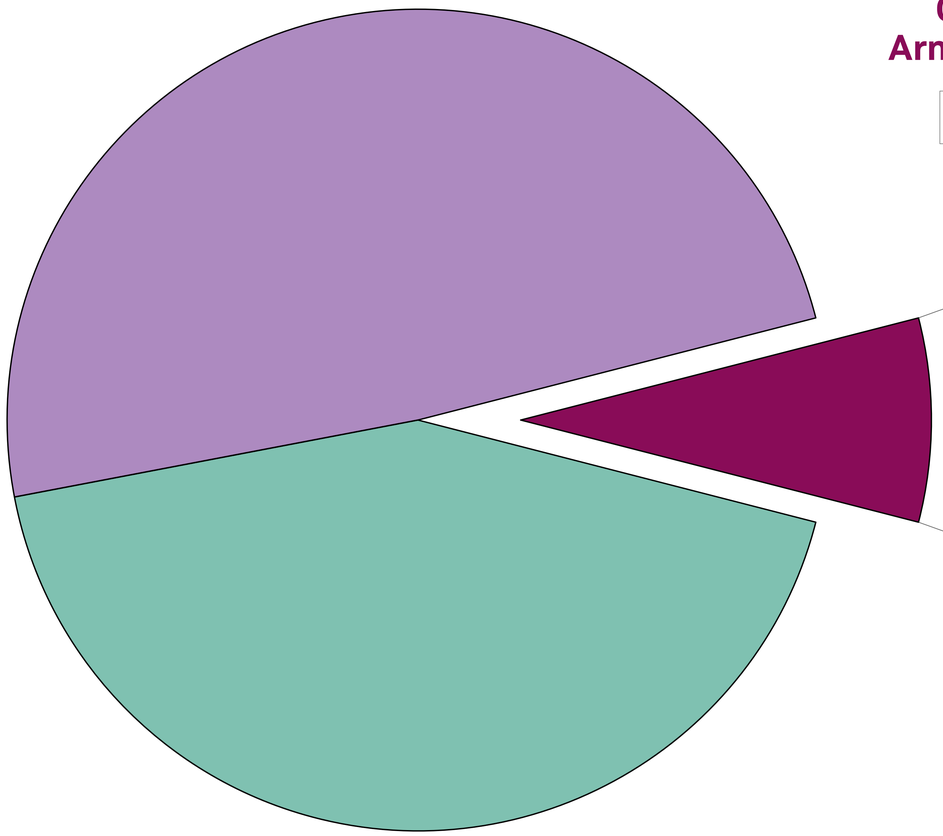


Running on Arm: Many Non-Trivial Cases Really Are Trivial

Vector intrinsics, dependencies, and nonstandard features are easily addressed

Straightforward, easy work < 1 day

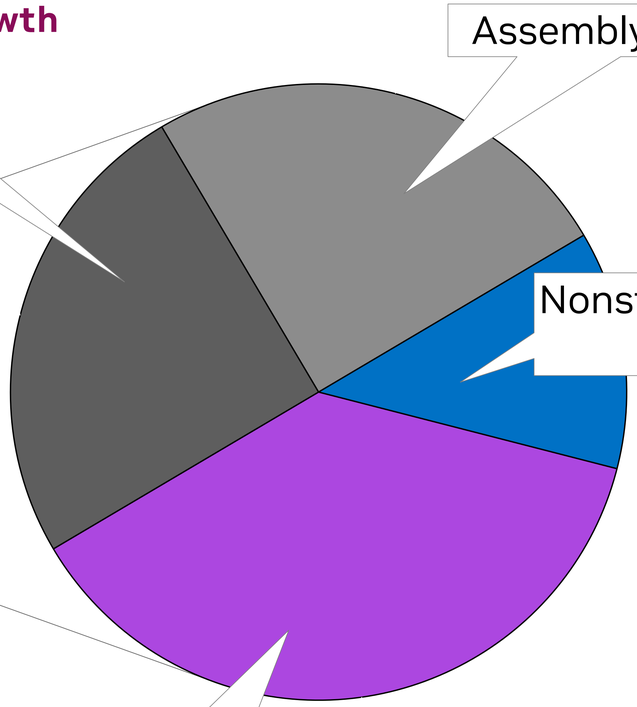
Recompile and reconfigure runtime parameters



Job done!

Found on Arm at another HPC center

Cloud momentum
Arm ecosystem growth



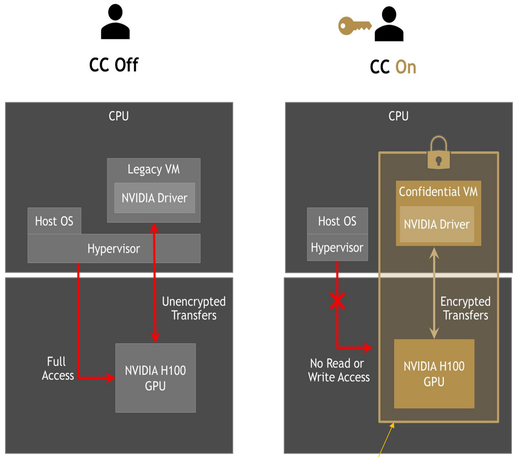
Intrinsic conversion tools – SIMDe, SSE2NEON, etc.

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

HOPPER TECHNOLOGICAL BREAKTHROUGHS

CONFIDENTIAL COMPUTING

Secure Data and AI Models In-Use

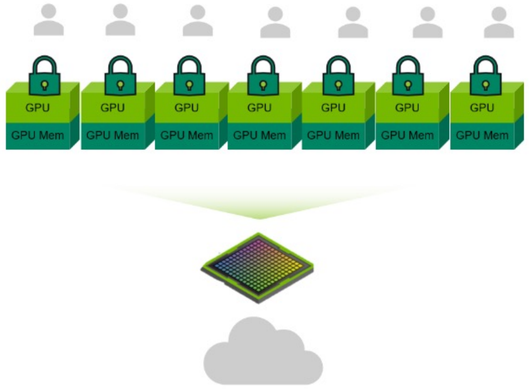


Confidential VM
(virtualization-based TEE)

Secure during compute, not just in storage or in motion

MULTI-GPU INSTANCE

7 Secure Tenants on 1 GPU



New vs A100:

- *6x compute, more mem bandwidth*
- *Dedicated NVDEC & NVJPG per MIG*
- *Each instance has its own telemetry*
- *Secure confidential compute MIG*

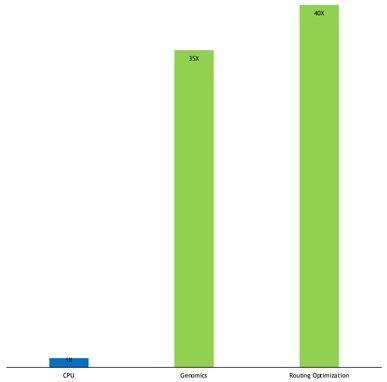
3x Compute
1.5x Bandwidth

NEW DYNAMIC PROGRAMING INSTRUCTIONS

Accelerate Dynamic Programming Algorithms

A BROAD RANGE OF USE CASES

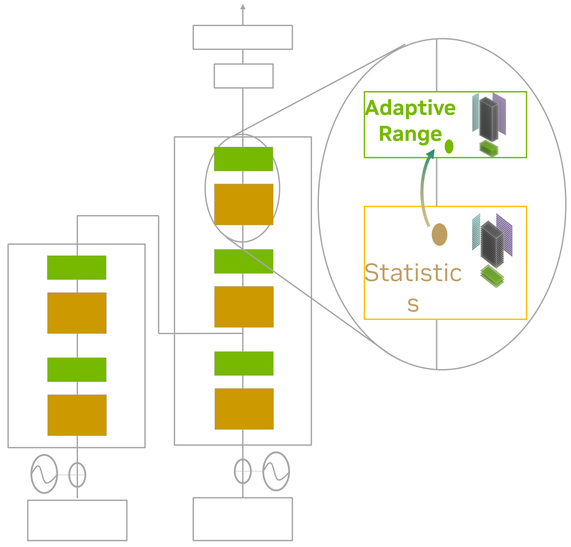
REAL-TIME PERFORMANCE



40x vs CPU
7x vs A100

TRANSFORMER ENGINE

Tensor Core Optimized for Transformer Models



Statistics and Adaptive Range Tracking

16-bit 8-bit

6x on LLM

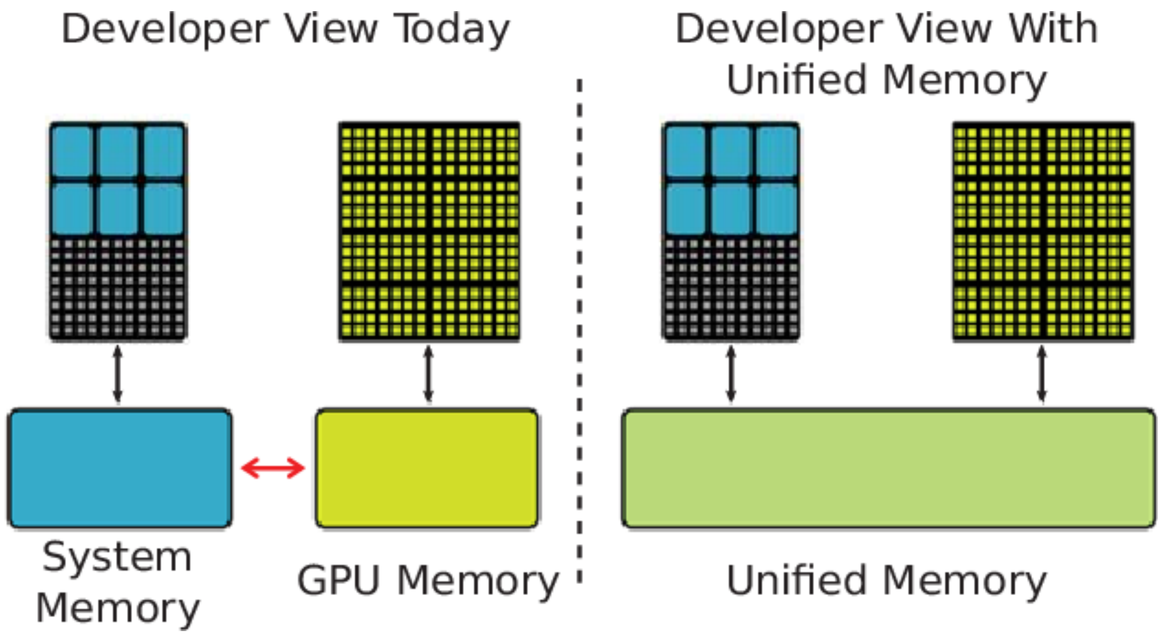
Dynamic Programming Instructions HGX H100 4-GPU vs dual socket 32 core IceLake. HGX H100 performance projections.

Unified Memory

Dramatically Lowering Developer Effort

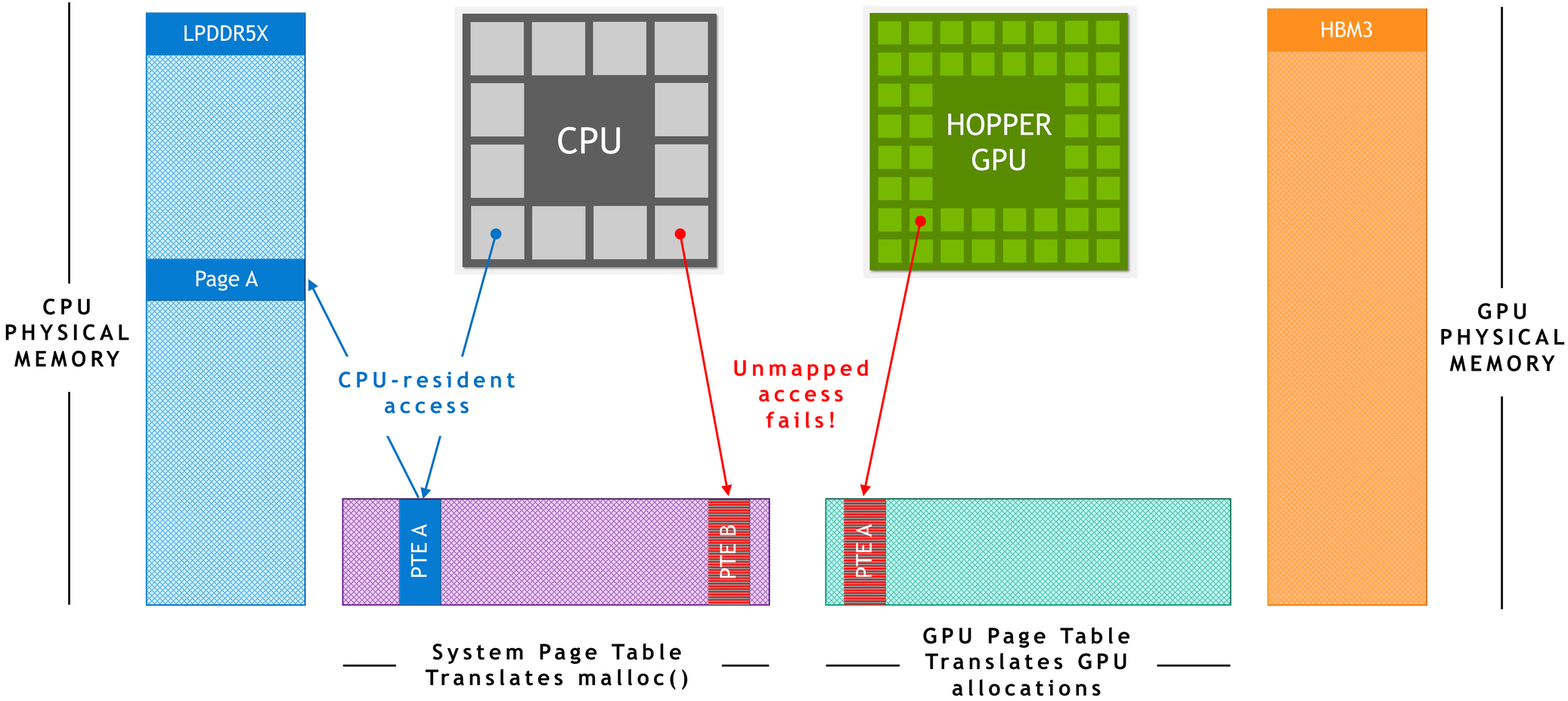
CUDA **Unified Memory** allows a program to dynamically allocate data that can be accessed from the CPU or from the GPU, with the same pointer and address

- 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



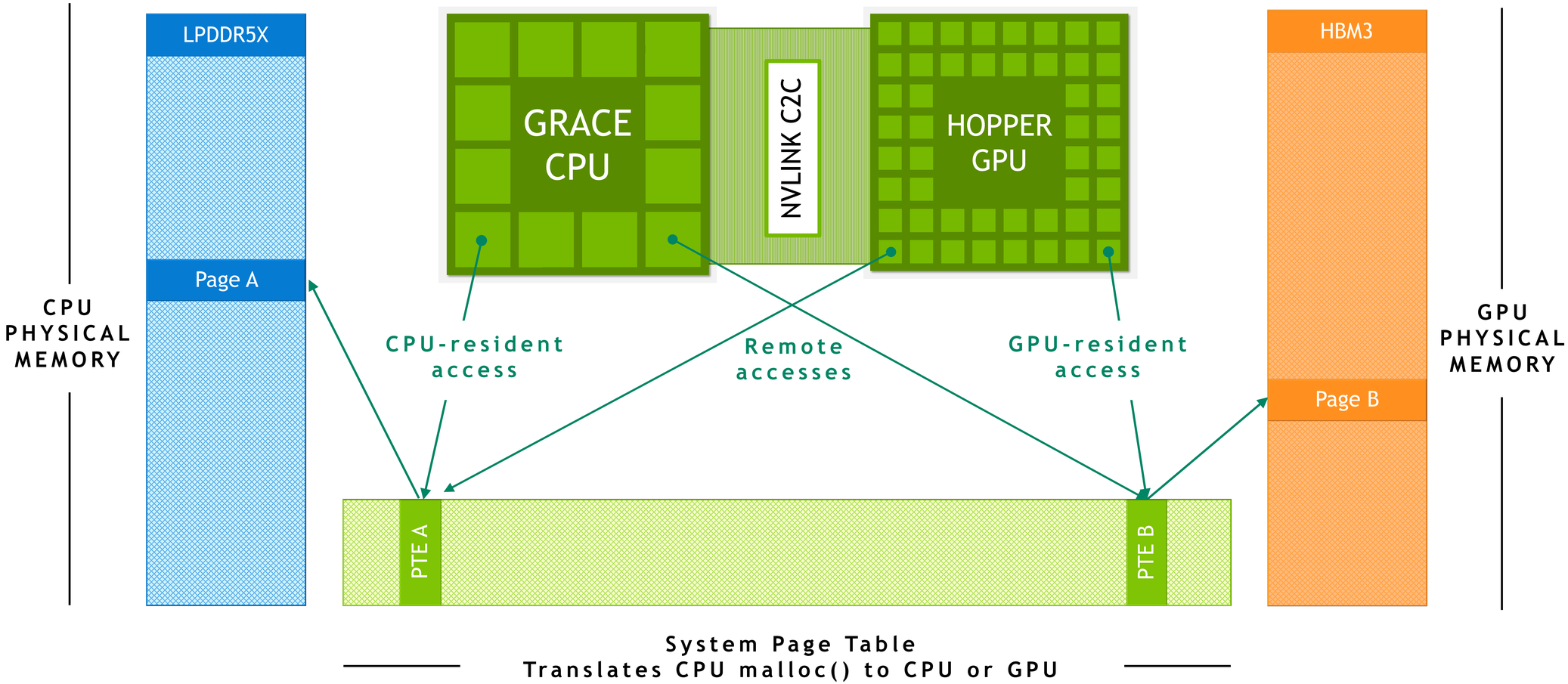
HW/SW memory view on X86 + GPU

Conceptually separate page tables



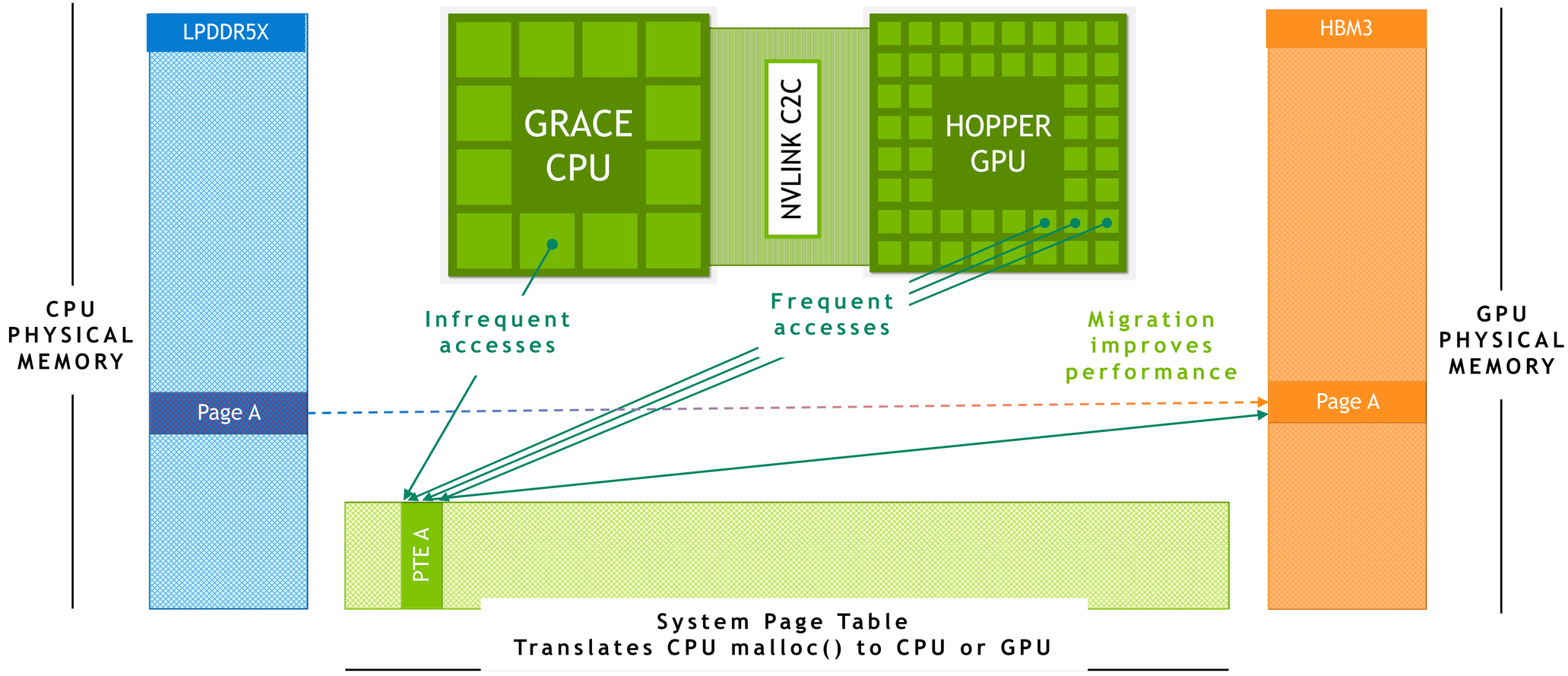
HW/SW memory view on Grace Hopper Superchip

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



HW/SW memory view on Grace Hopper Superchip

Automatic Migrations

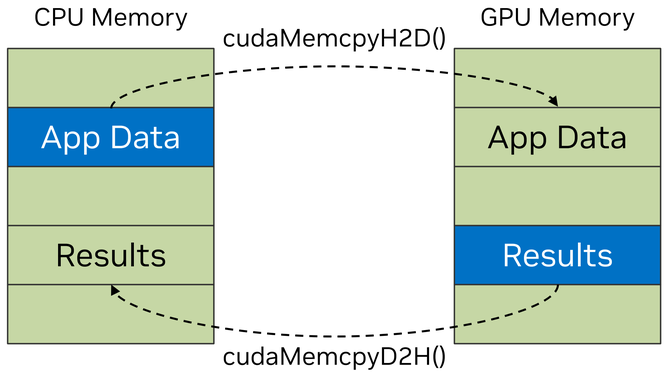


Explicit Data Movement (bulk/blocking)

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)

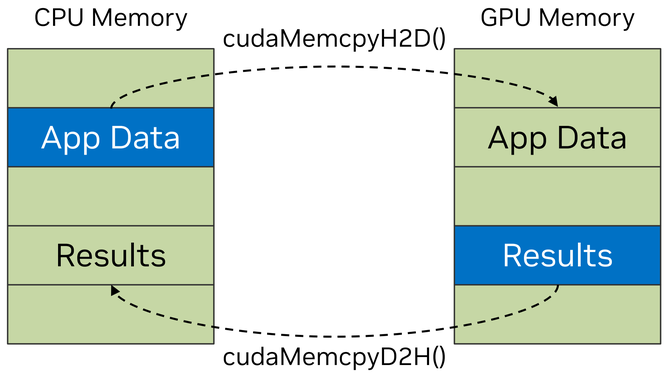
Grace: Faster transfers; up to 450 GB/s C2C transfers (per direction)

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

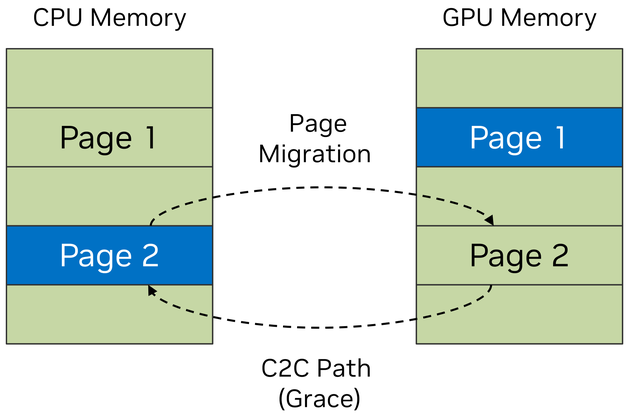
Grace: Faster transfers; up to 450 GB/s C2C transfers (per direction)

Managed Memory

Portable codes across x86 and Grace Hopper

```
__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

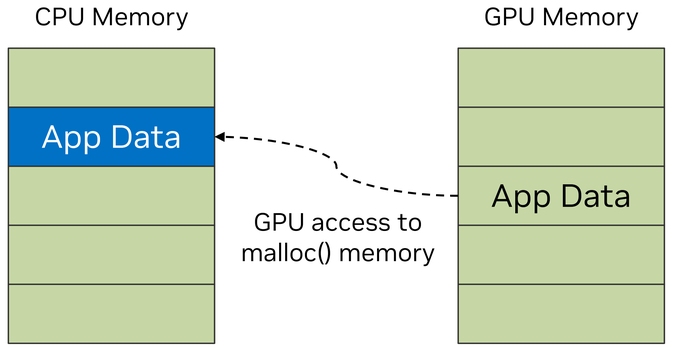
Grace: Migrations not required and faster migrations when they happen

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

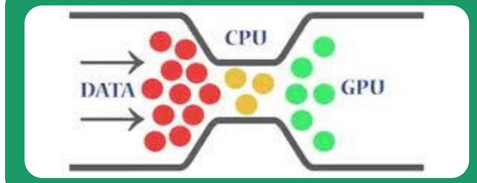
Where is Grace+Hopper better than X86+hopper?

HPC use-cases

OpenFOAM®

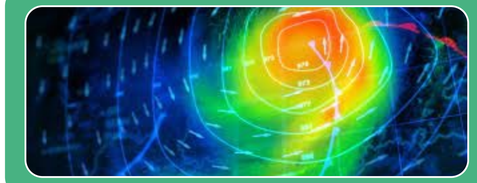
Partially Ported Apps

- OpenFOAM – solver only (bar is lower to better price/perf)



Apps that bottleneck on PCI connectivity

- ABINIT example with pencil-shaped ZGEMM
- Large AI Training



Apps that can leverage tight cache coherence

- Data Assimilation step in weather models can stay on Grace
- Multigrid solvers

New-to-GPU Apps

- Can more effectively leverage standard language acceleration

ADDITIONAL WEB RESOURCES

NVIDIA GTC 2023

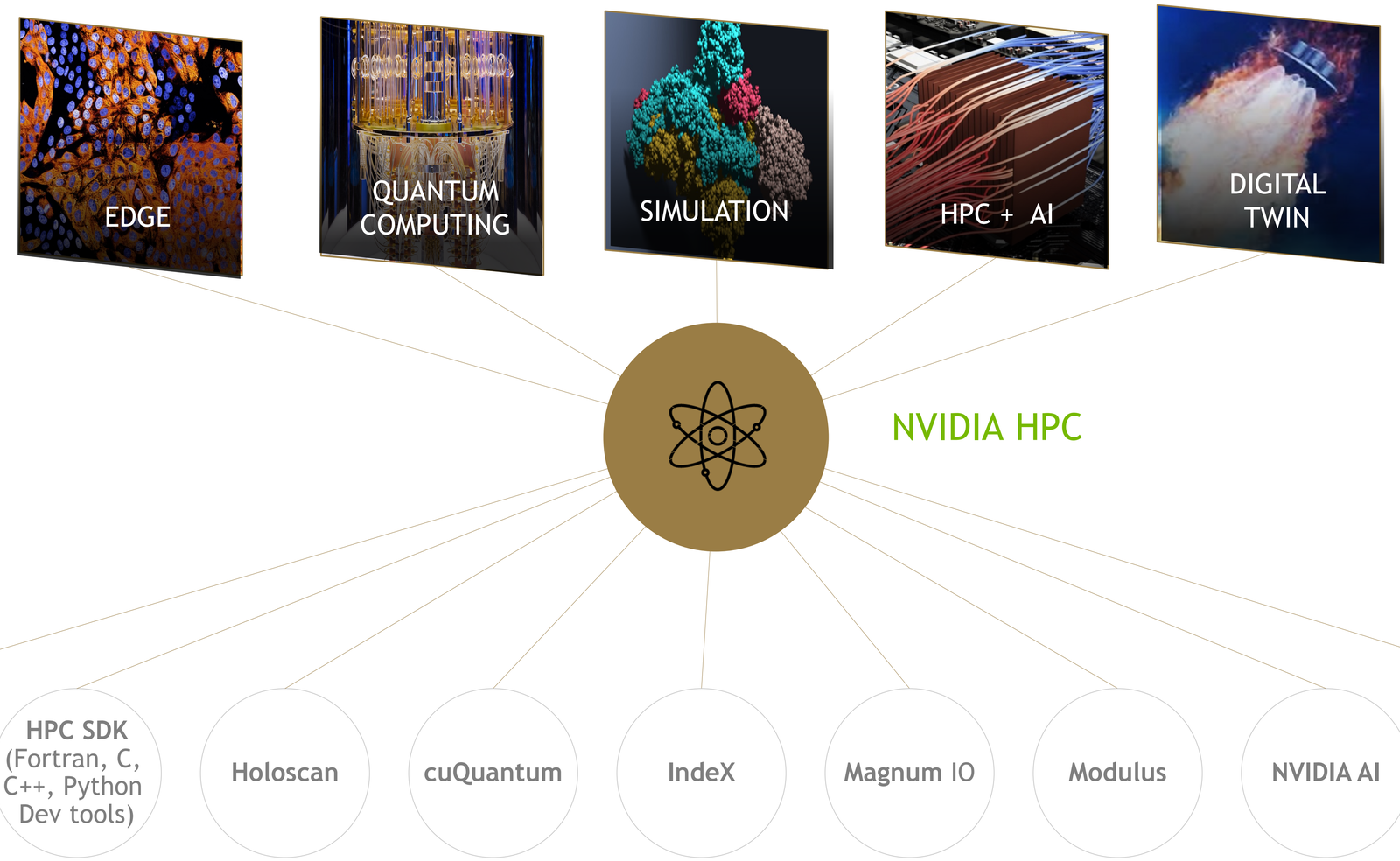
NVIDIA Grace CPU Superchip Whitepaper

NVIDIA Grace Hopper Superchip Whitepaper


NVIDIA

(Bonus)
Beyond HPC: the Edge

NVIDIA's HPC PLATFORM



RISE OF HPC AT THE EDGE

Posing a New Set of Challenges for HPC

10X – 100X MORE DATA

50+ GIANT SCALE INSTRUMENTS WW

ELT ESO

ALS @LBNL

LIGO

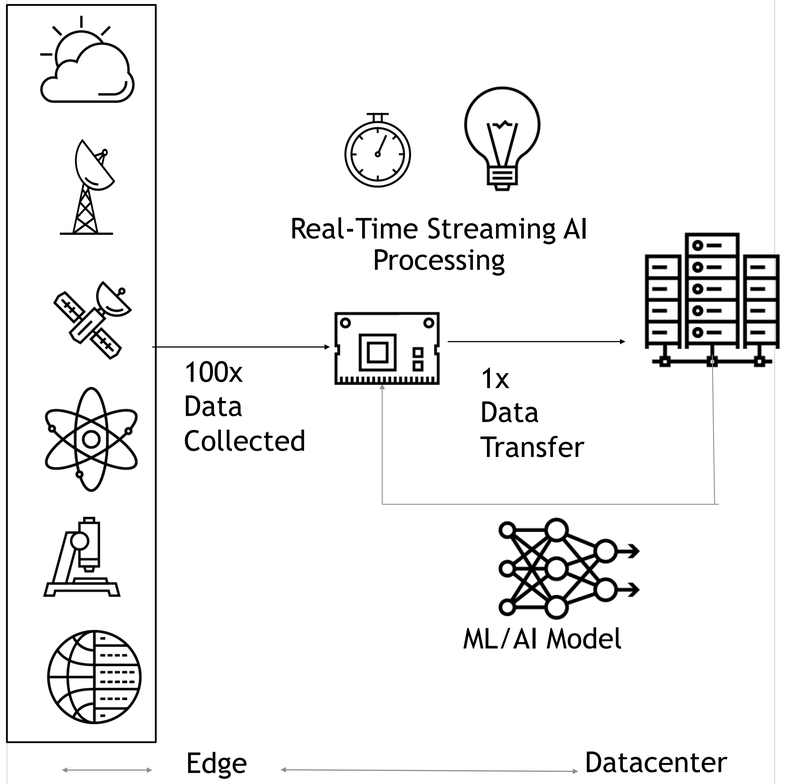
APS @ANL

SKA

Diamond, UK

AI SUPERCOMPUTING AT THE EDGE

ENABLES REAL-TIME INSIGHTS AND CONTROL



STREAMING DEPLOYMENT IS HARD

FOR DATA SCIENTISTS, RESEARCHERS AND DEVOPS

Streaming Data Performance

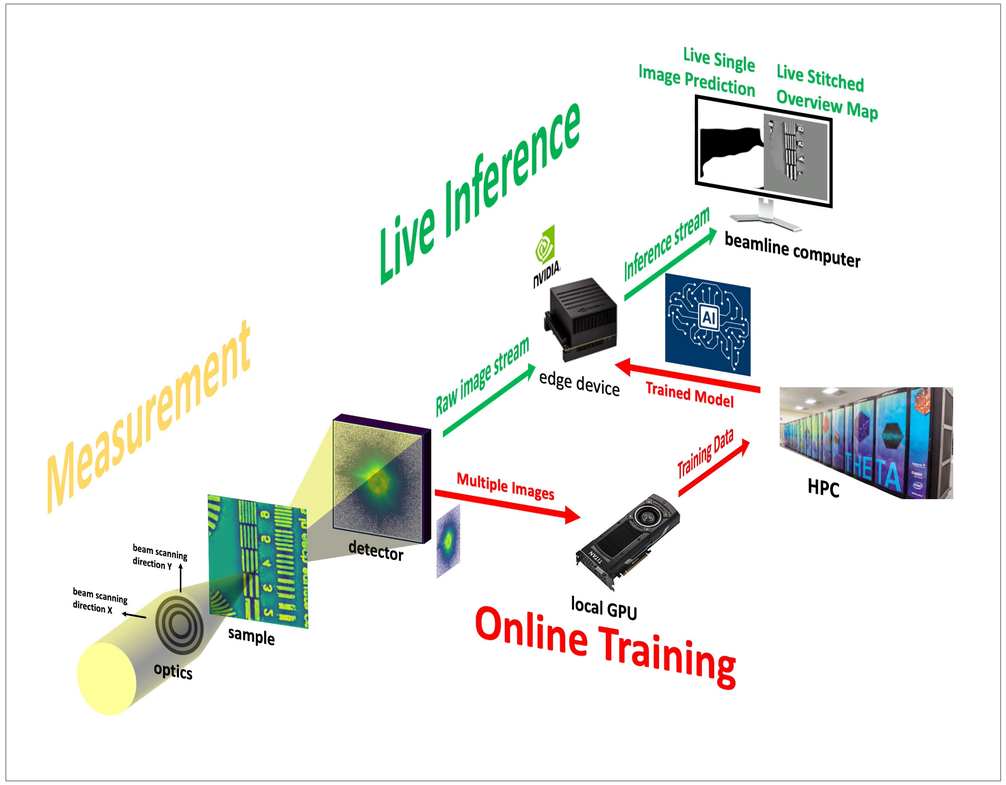
Developer Ease-of Use

Easily Scale Implementation

Combining multiple datastreams

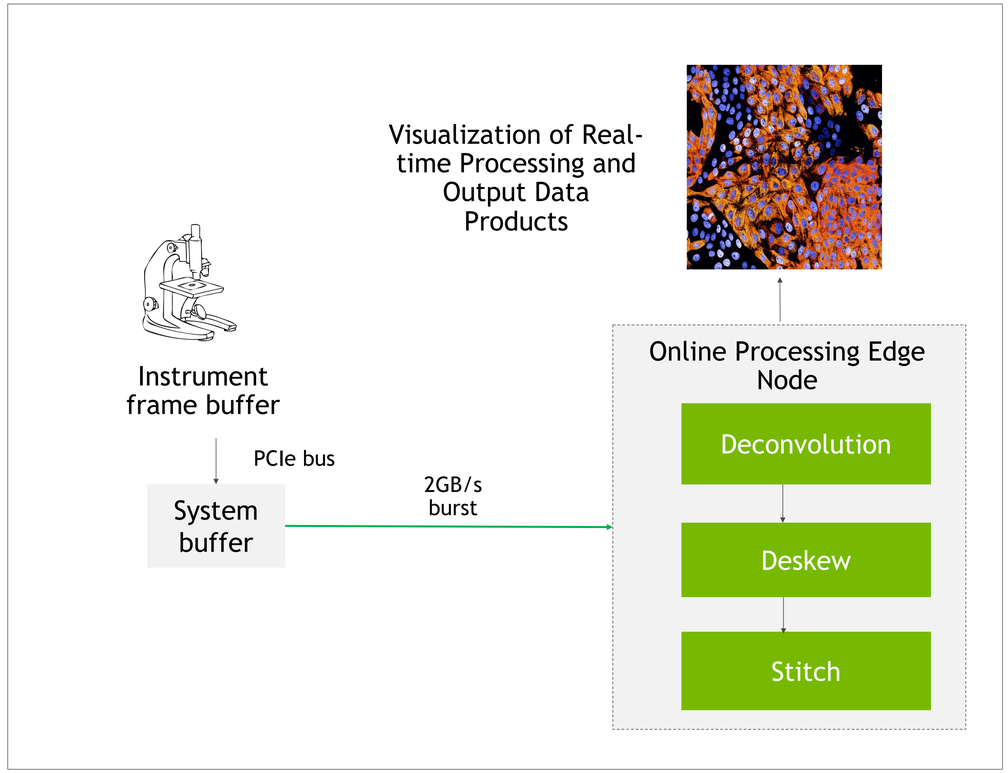
SUPERCHARGING SCIENCE EXPERIMENTS AND INSTRUMENTS

ANL/ APS ACCELERATES X-RAY PTYCHOGRAPHY 300X WITH PTYCHONN



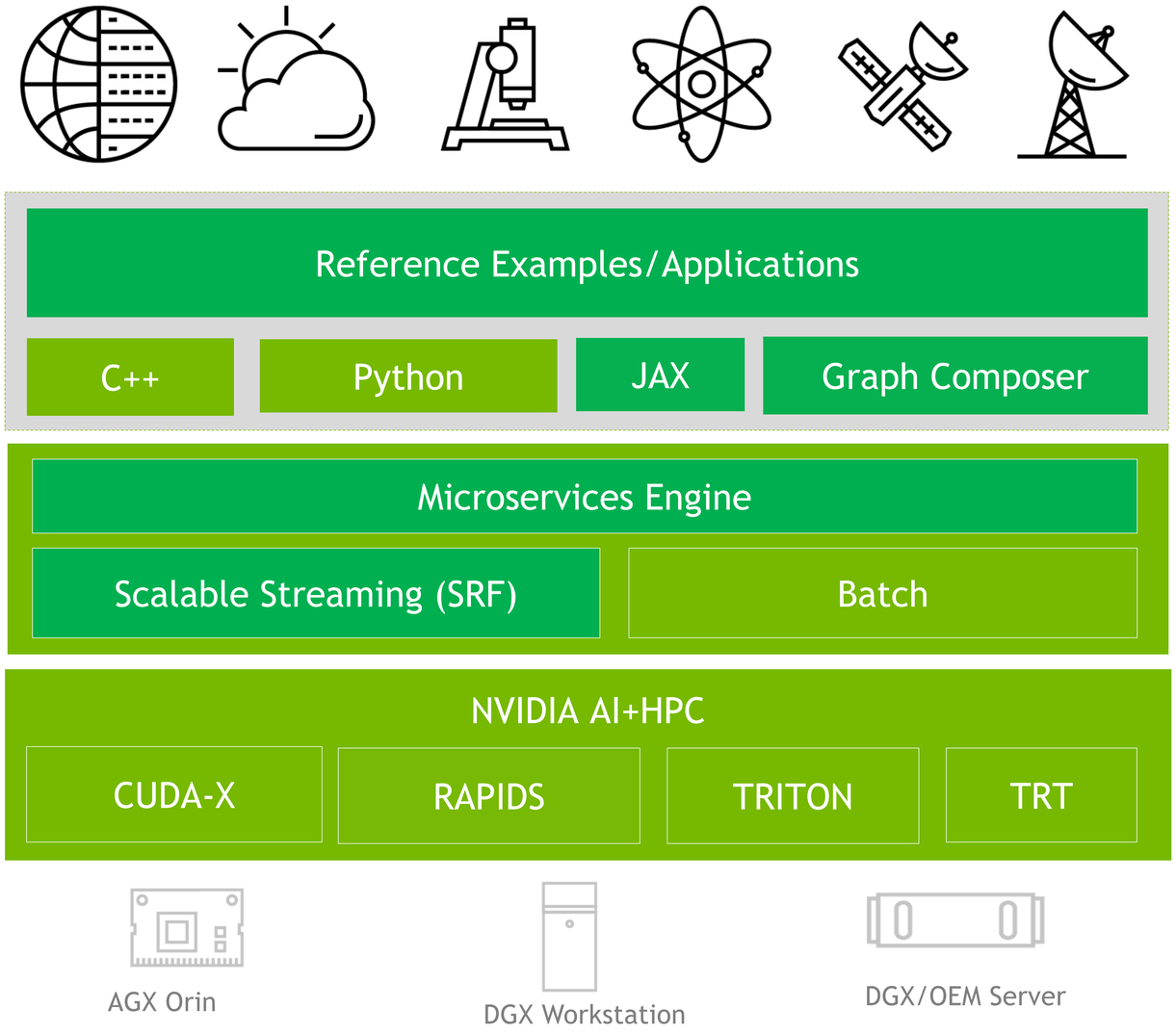
PtychoNN paper: AI-enabled high-resolution scanning coherent diffraction imaging

ADVANCED BIOIMAGING CENTER @UC-BERKELEY REAL TIME LIVE CELL IMAGING LIGHT SHEET MICROSCOPY



Link to keynote video - https://youtu.be/rXG27G3bWzY

HOLOSCAN SDK : INTEGRATING DATA STREAMING FROM THE EDGE TO THE DATACENTER FOR ACCELERATING SCIENTIFIC DISCOVERY



- Build their applications using a mix of C++, Python, JAX
- Develop AI microservices combining low-latency data streaming while passing more complex tasks to data center resources
- Scale from embedded to datacenter

Holoscan SDK
(GitHub project)