NVIDIA HPC Software – Expanding HPC with Python and AI

Becca Zandstein, Director Product Management NVIDIA | **CUG 2025**

NVIDIA HPC SDK

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

DEVELOPMENT

Programming Models
- Standard C++ & Fortran
- OpenACC & OpenMP
- CUDA

Compilers
- nvcc
- nvc
- nvc++
- nvfortran

Core Libraries
- libcu++
- Thrust/CCCL
- CUB

Math Libraries
- Core CUDA Math Libraries
- Scalable Math Libraries
- NVPL: CPU Libraries

Communication Libraries
- HPC-X
  - MPI
  - UCX
  - SHMEM
  - SHARP
  - UCC
- NVSHMEM
- NCCL

ANALYSIS

Profilers
- Nsight
  - Systems
  - Compute

Debugger
- cuda-gdb
  - Host
  - Device

Programming the NVIDIA Platform

Unmatched Developer Flexibility

Languages & Programming Models

Accelerated Standard Languages	Incremental Optimization	Platform Specialization
C++, python™, Fortran	OpenACC (More Science, Less Programming), OpenMP®	NVIDIA® CUDA® C++ | Fortran | Python

Libraries & Frameworks

Acceleration Libraries

Core	Math	Communication	Data Analytics	AI	DSLs

Hardware

GPU	CPU	Interconnect

Wherever You Run

PCs	Workstations	On Prem	Cloud	At the Edge

Python for HPC

CUDA Accelerated Python Whole-Platform Stack

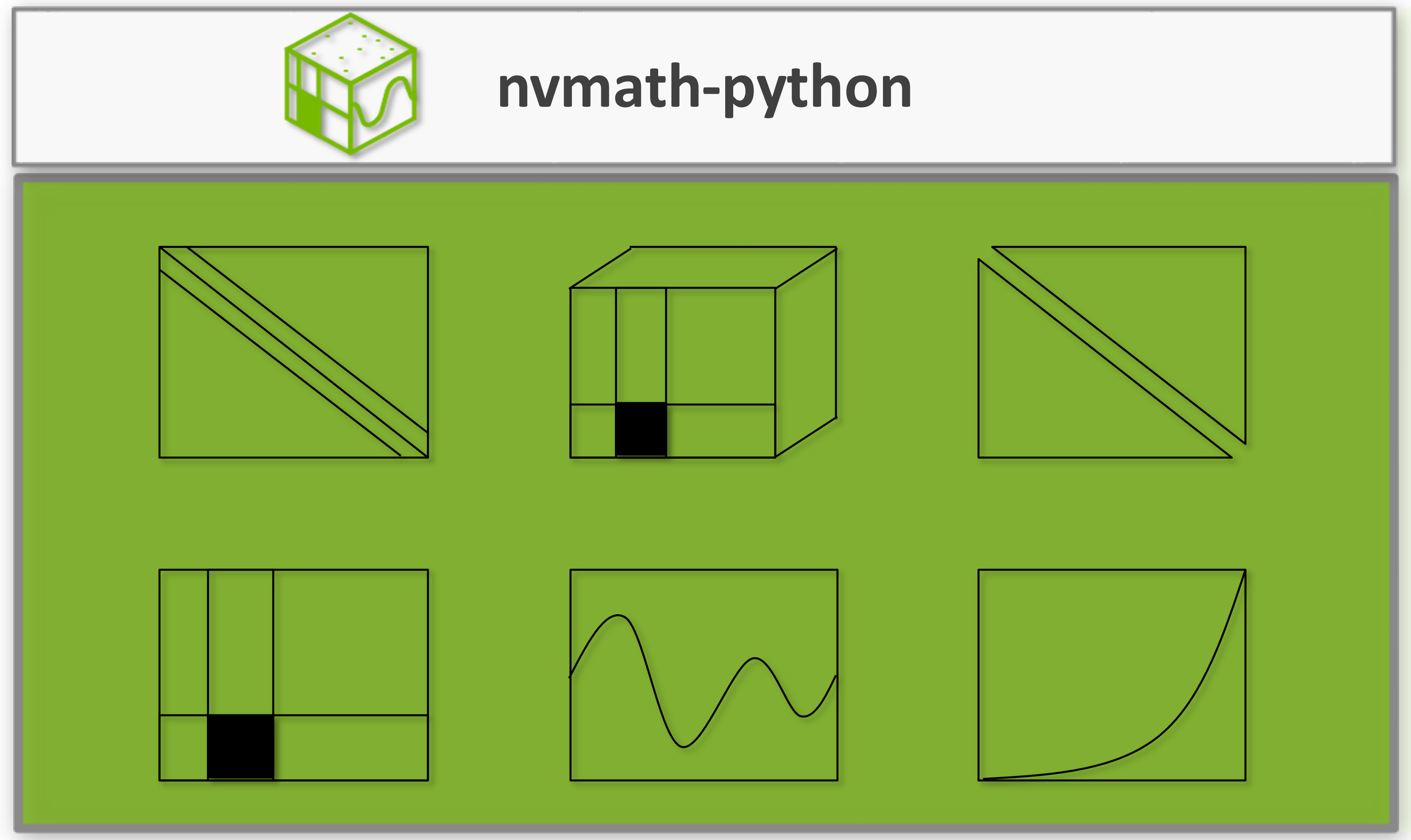
Interoperability

Mix and match layers

	CUDA Python		
Frameworks & DSLs	TensorRT	Omniverse	Warp
SDKs	RAPIDS	CUDA-Q	Holoscan
Domain-Specific Libraries	cuPyNumeric	cuDNN-python	
Accelerated Libraries	cuda.parallel	nvmath-python	
Communication Libraries	mpi4py	nvshmem4py	
Device Libraries	cuda.cooperative	nvmath-python	
Kernel Authoring	numba-cuda	cuTile	CUTLASS DSL
Compiler Stack			
Host Runtimes & Tools	cuda.bindings	cuda.core	Nsight Systems

nvmath-python

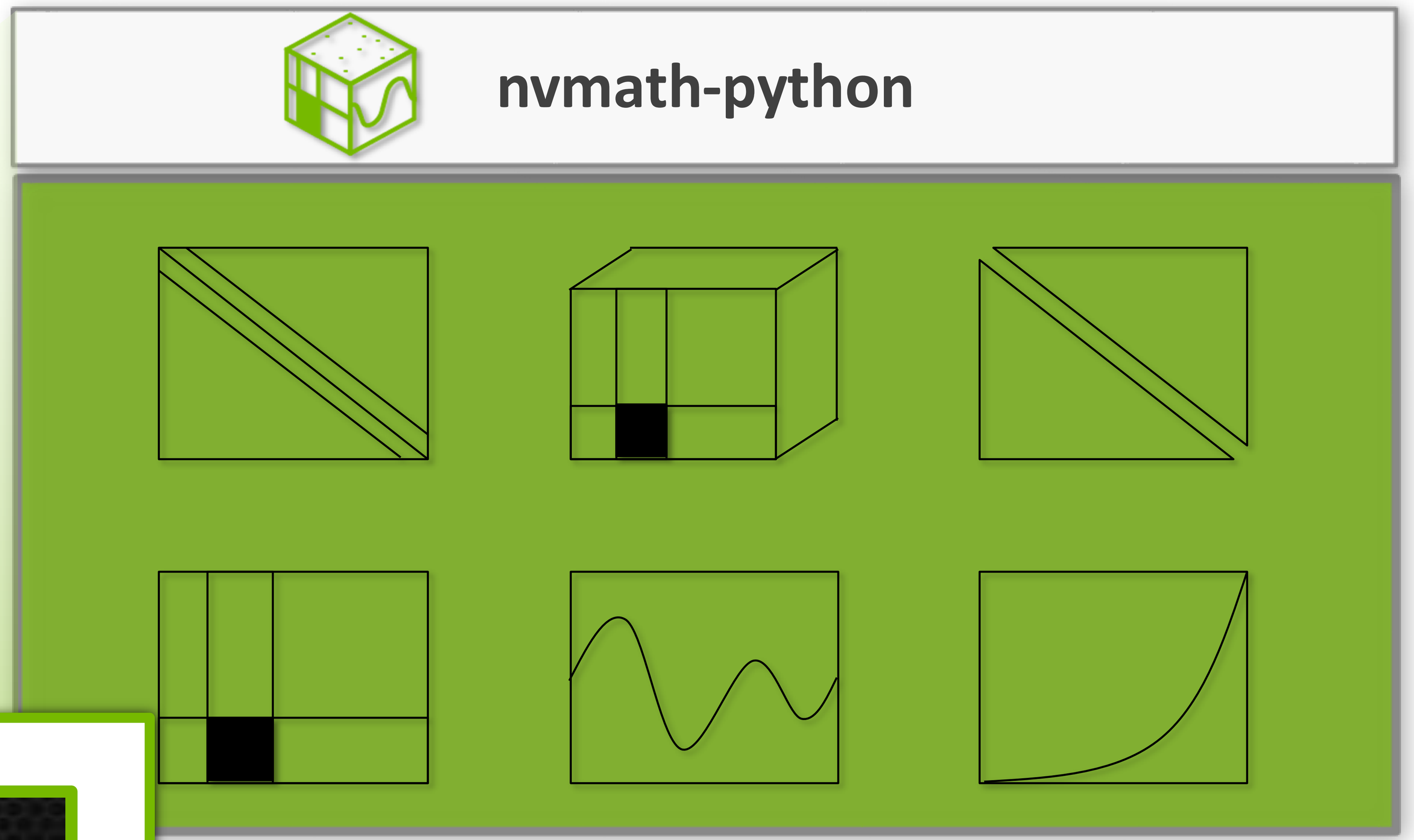
Reimagining math libraries for the Python ecosystem

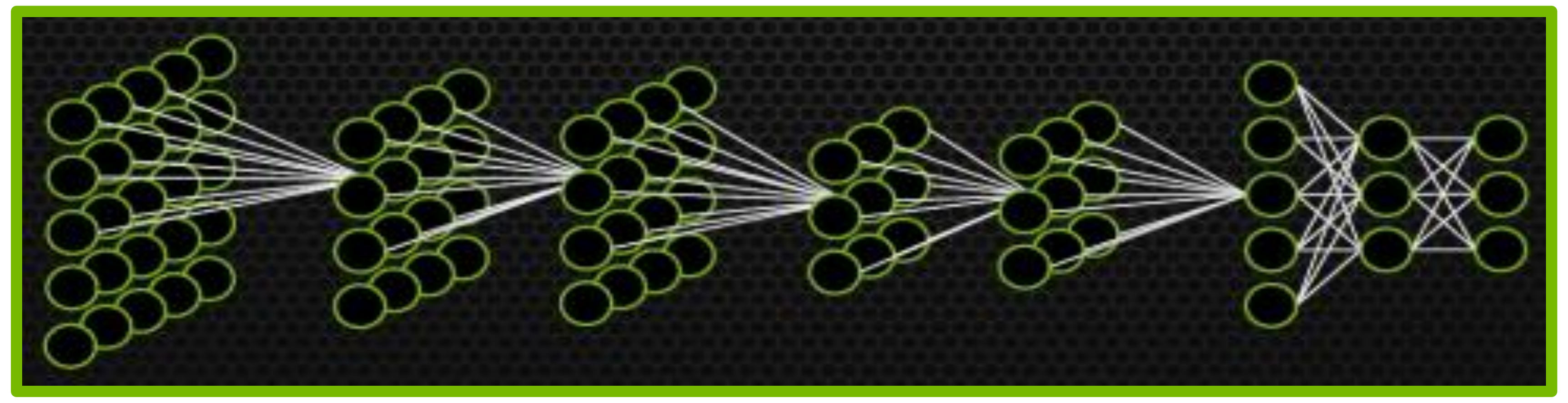


Core numerical computing operations for solving the most challenging problems in scientific computing and AI

nvmath-python

Reimagining math libraries for the Python ecosystem





Intuitive APIs designed for usability without performance compromises, allowing for autotuning. Actionable error messages and full logging support.

$$D = \mathrm{ReLU}(A \cdot B + bias)$$

Stateless API for greatest convenience & productivity

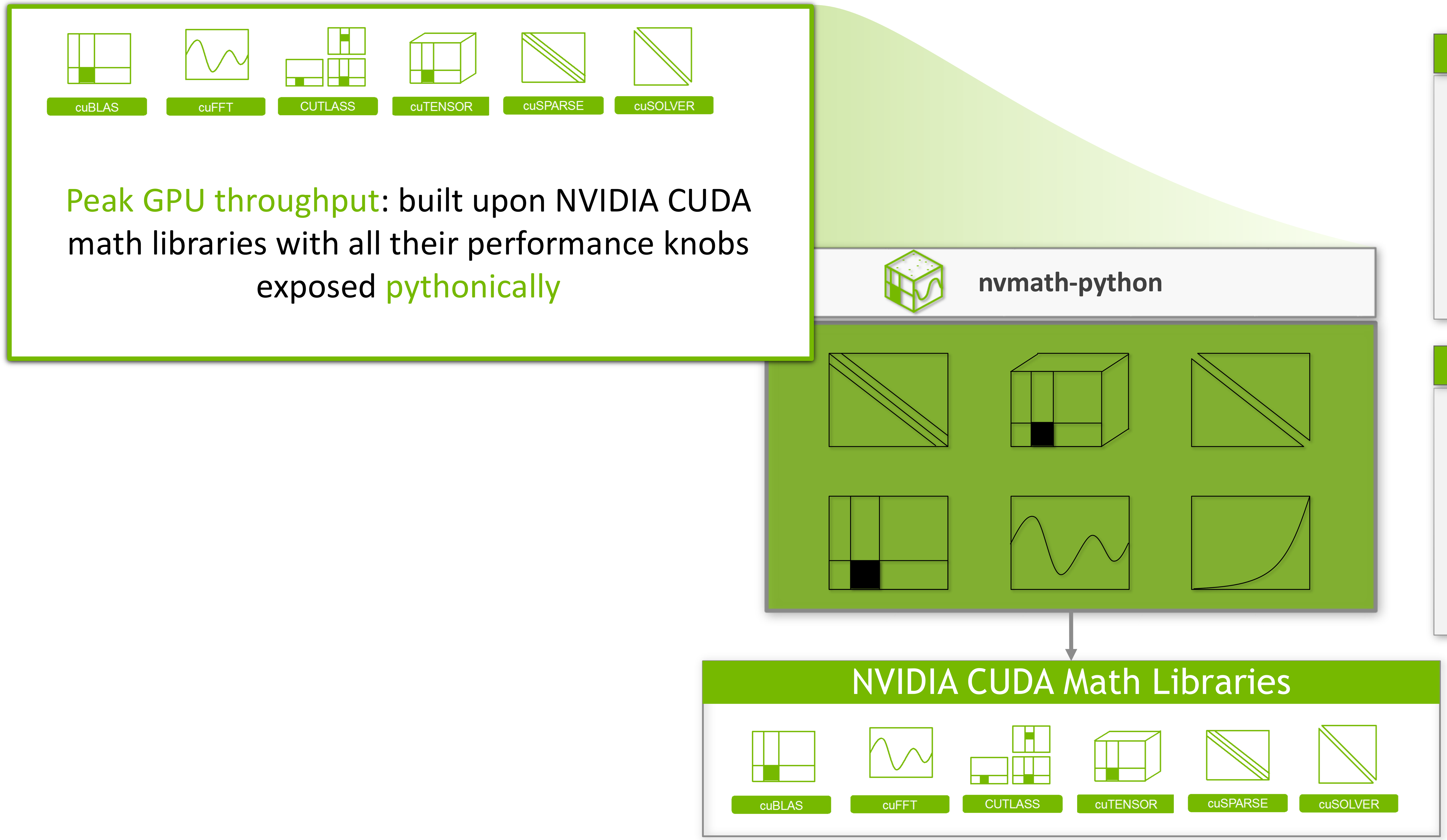
```
import torch
import nvmath

a = torch.rand(m, k)
b = torch.rand(k, n)
bias = 0.25 * torch.rand(m, 1, dtype=torch.bfloat16, device=device_id) - 0.5

relu_bias = nvmath.linalg.MatmulEpilog.RELU_BIAS

result = nvmath.linalg.advanced.matmul(a, b, epilog=relu_bias, epilog_inputs={'bias': bias})
```

Stateful API for greatest control & complex usage scenarios

```
import torch
import nvmath

a = torch.rand(m, k)
b = torch.rand(k, n)
bias = 0.25 * torch.rand(m, 1, dtype=torch.bfloat16, device=device_id) - 0.5

relu_bias = nvmath.linalg.MatmulEpilog.RELU_BIAS

# Use the stateful object as a context manager to automatically release resources.
with nvmath.linalg.advanced.Matmul(a, b, epilog=relu_bias, epilog_inputs={'bias': bias}) as
mm:
    # Planning returns a sequence of configurable algorithms
    mm.plan()

    # Execute the matrix multiplication
    result = mm.execute()

    # Update the operand A in-place.
    a[:] = torch.rand(m, k)

    # Execute the new matrix multiplication.
    result = mm.execute()
```

nvmath-python

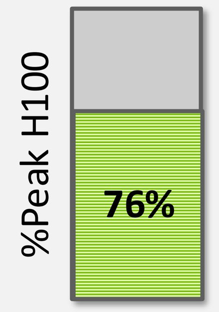
Reimagining math libraries for the Python ecosystem



$$D = \mathrm{ReLU}(A \cdot B + bias)$$

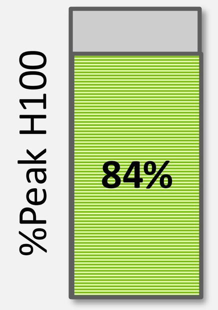
Three unfused kernels with traditional APIs

```
import torch
import nvmath

a = torch.rand(m, k)
b = torch.rand(k, n)
bias = 0.25 * torch.rand(m, 1, dtype=torch.bfloat16, device=device_id) - 0.5

relu = torch.nn.ReLU()
result = relu(torch.matmul(a, b) + bias)
```



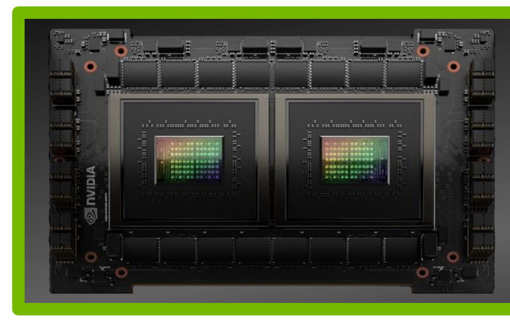
Single fused advanced nvmath-python API

```
import torch
import nvmath

a = torch.rand(m, k)
b = torch.rand(k, n)
bias = 0.25 * torch.rand(m, 1, dtype=torch.bfloat16, device=device_id) - 0.5

relu_bias = nvmath.linalg.MatmulEpilog.RELU_BIAS
result = nvmath.linalg.advanced.matmul(a, b, epilog=relu_bias, epilog_inputs={'bias': bias})
```



nvmath-python

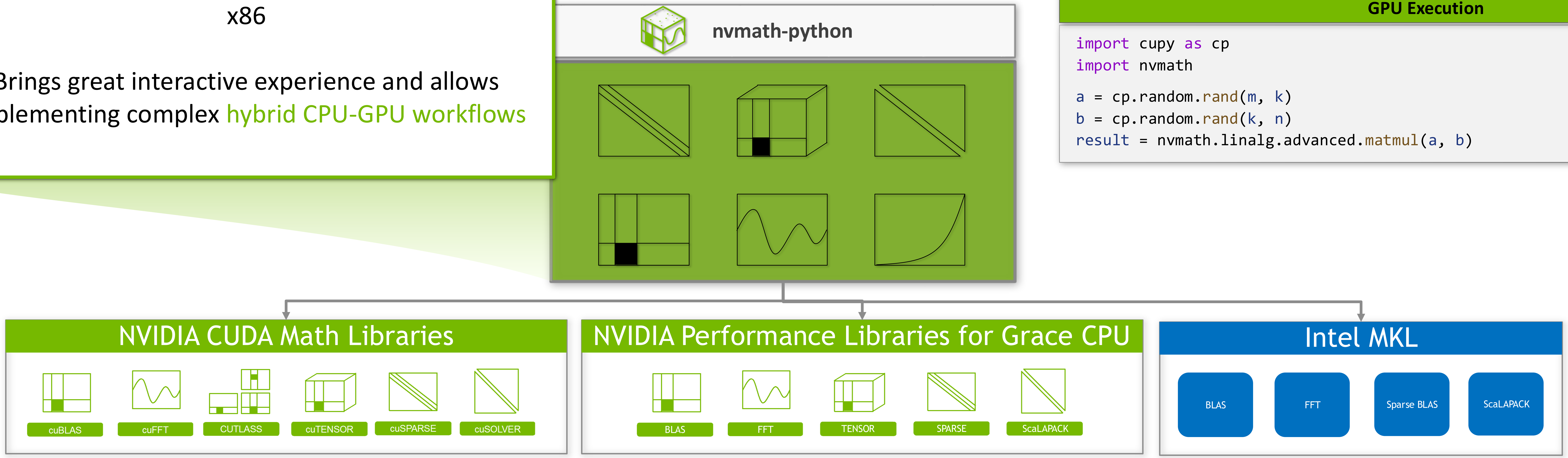
Reimagining math libraries for the Python ecosystem



Grace CPU

Support for GPU and CPU memory and execution spaces* backed by NVPL on aarch64 and MKL for x86

Brings great interactive experience and allows implementing complex hybrid CPU-GPU workflows



CPU Execution

```python
import numpy as np
import nvmath

a = np.random.rand(m, k)
b = np.random.rand(k, n)
result = nvmath.linalg.advanced.matmul(a, b)
```

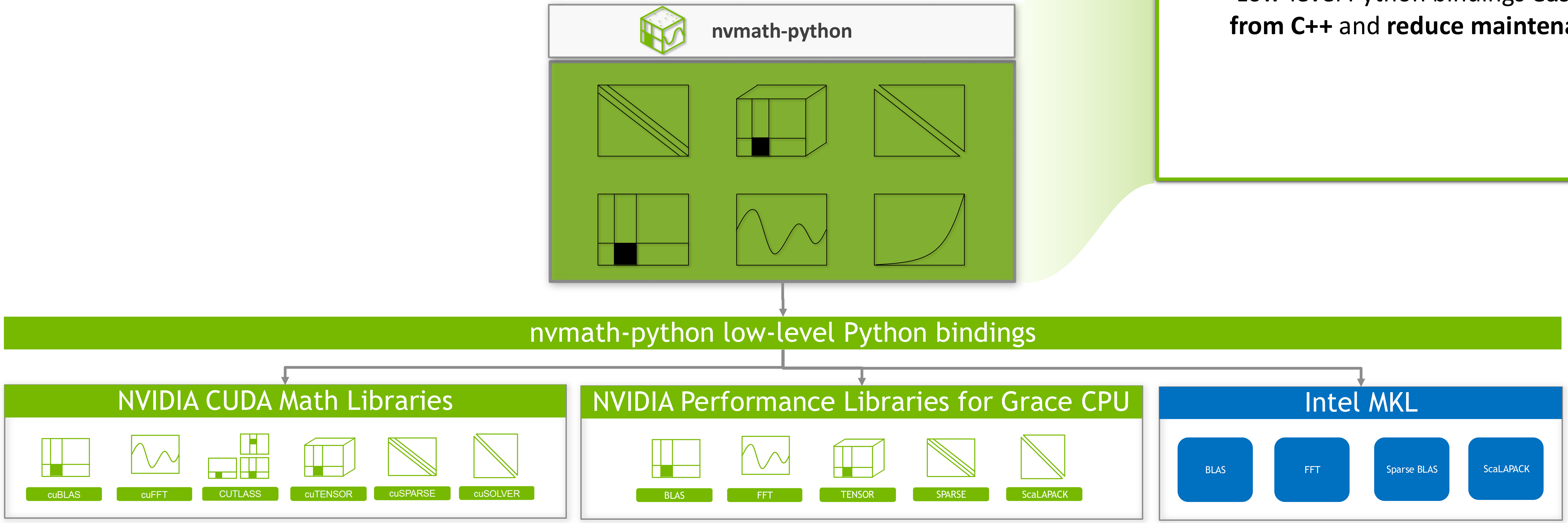
GPU Execution

```python
import cupy as cp
import nvmath

a = cp.random.rand(m, k)
b = cp.random.rand(k, n)
result = nvmath.linalg.advanced.matmul(a, b)
```

* Execution and memory spaces are distinct concepts. The library's default behavior is to execute in the space where the tensor arguments reside. However, users can override default behavior and execute in a different space (at the cost of data transfer across memory spaces).

nvmath-python

Reimagining math libraries for the Python ecosystem

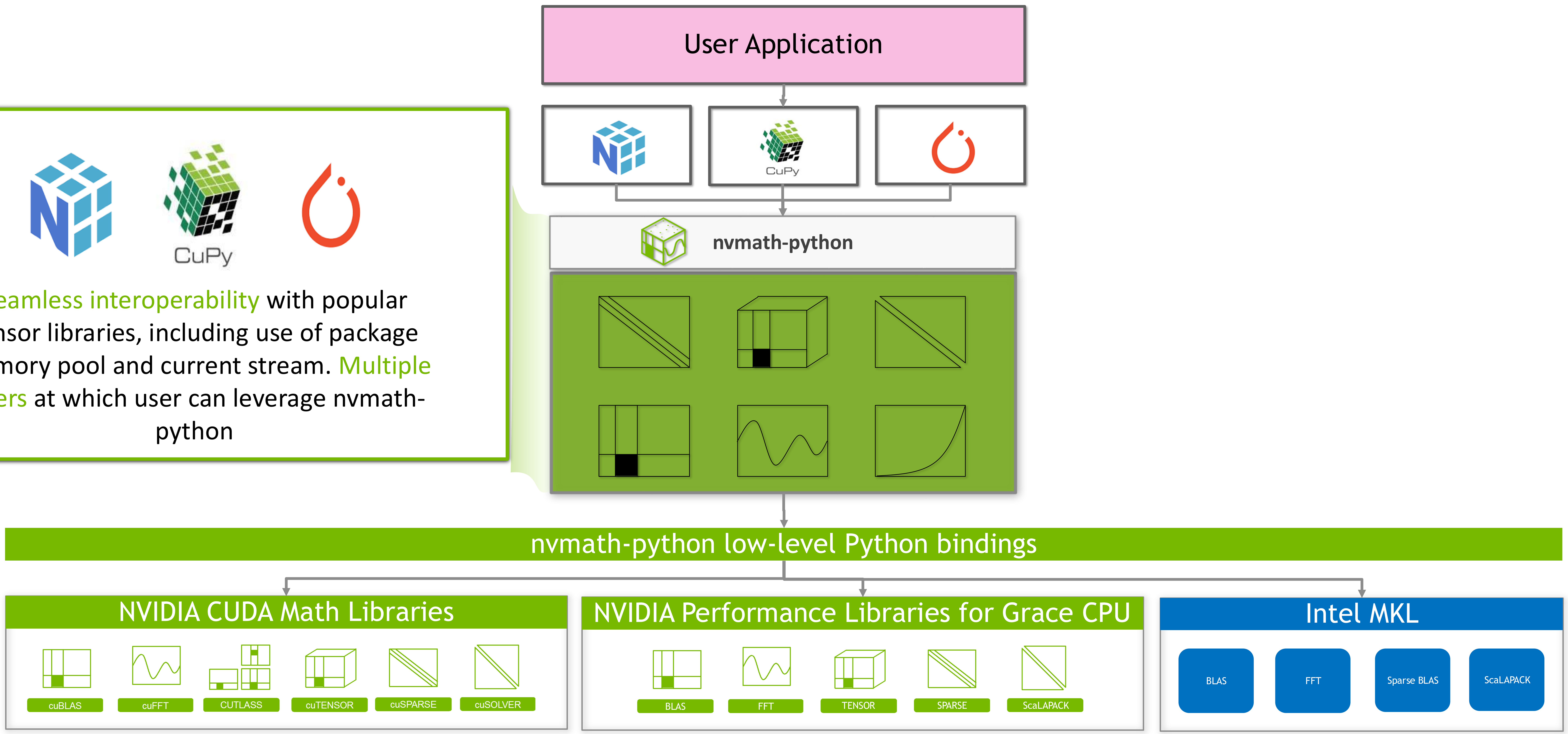


Low-level Python bindings **ease migration from C++** and **reduce maintenance burden**

nvmath-python

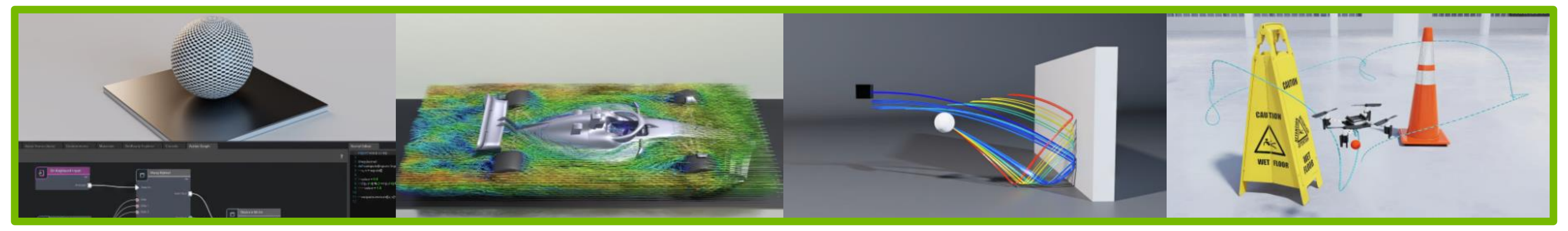
Reimagining math libraries for the Python ecosystem

Seamless interoperability with popular tensor libraries, including use of package memory pool and current stream. Multiple layers at which user can leverage nvmath-python



nvmath-python

Reimagining math libraries for the Python ecosystem



Use nvmath device APIs from within custom kernels written in numba-cuda SIMT or NVIDIA Warp Tile programs

nvmath-python

Host APIs | Host APIs with device callbacks | Device APIs

Device APIs → numba-cuda SIMT, NVIDIA Warp Tile

C++ & cuFFTDx SIMT Convolution

```
template<class FFT, class IFFT>
__launch_bounds__(FFT::max_threads_per_block)
__global__ void convolution_kernel(typename FFT::value_type* signal,
                                   typename FFT::value_type* signal_filter)
{
    using complex_type = typename FFT::value_type;

    // Allocate register memory.
    complex_type signal_rmem[FFT::storage_size];
    extern __shared__ __align__(alignof(float4)) complex_type shared_mem[];

    const unsigned int local_fft_id = threadIdx.y;
    const unsigned int global_fft_id = blockIdx.x *
       FFT::ffts_per_block + local_fft_id;

    // Load data into register memory.
    example::io<FFT>::load(signal, signal_rmem, local_fft_id);

    // Forward FFT (inplace).
    FFT().execute(signal_rmem, shared_mem);

    // Apply the filter in the frequency domain.
    unsigned int index = threadIdx.x;
    for (int i=0; i < FFT::elements_per_thread; ++i) {
        if (index < cufftdx::size_of<FFT>::value) {
            signal_rmem[i] *= signal_filter[global_fft_id + index];
            index += FFT::stride;
        }
    }

    // Inverse FFT (inplace).
    IFFT().execute(signal_rmem, shared_mem);

    // Store convolution result (overwrite input signal).
    example::io<FFT>::store(signal_rmem, signal, local_fft_id);
}
```

45ms

Numba & nvmath SIMT Convolution

```
@cuda.jit(link=FFT.files + IFFT.files)
def convolution_kernel(signal : cp.array,
                       signal_filter: cp.array):

    # Allocate register memory.
    signal_rmem = cuda.local.array(shape=(storage_size,),dtype=value_type)
    shared_mem = cuda.shared.array(shape=(0,), dtype=value_type)

    local_fft_id = cuda.threadIdx.y
    global_fft_id = cuda.blockIdx.x * ffts_per_block + local_fft_id

    # Load data into register memory.
    index = cuda.threadIdx.x
    for i in range(ept):
        signal_rmem[i] = signal[global_fft_id, index]
        index += stride

    # Forward FFT (inplace).
    FFT(signal_rmem, shared_mem)

    # Apply the filter in the frequency domain.
    index = cuda.threadIdx.x
    for i in range(ept):
        signal_rmem[i] *= signal_filter[global_fft_id, index]
        index += stride

    # Inverse FFT (inplace).
    IFFT(signal_rmem, shared_mem)

    # Store convolution result (overwrite input signal).
    index = cuda.threadIdx.x
    for i in range(ept):
        signal[global_fft_id, index] = signal_rmem[i]
        index += stride
```

45ms

Warp & nvmath Tile Convolution

```
@wp.kernel
def convolution_kernel(signal: wp.array2d(dtype=wp.vec2d),
                       signal_filter: wp.array2d(dtype=wp.vec2d)):

    index = wp.tid()

    # Load signal and filter into tiles.
    signal_tile = wp.tile_load(signal, shape=(1, FFT_SIZE), offset=(index,
0))
    signal_filter_tile = wp.tile_load(signal_filter, shape=(1, FFT_SIZE), \
      offset=(index, 0))

    # Forward FFT (inplace) on the tile.
    wp.tile_fft(signal_tile)

    # Apply the filter in the frequency domain.
    convolution = wp.tile_map(complex_multiply_warp, signal_tile, \
      signal_filter_tile)

    # Inverse FFT (inplace) on the tile.
    wp.tile_ifft(convolution)

    # Store convolution tile (overwrite input signal).
    wp.tile_store(signal, convolution, offset=(index, 0))
```

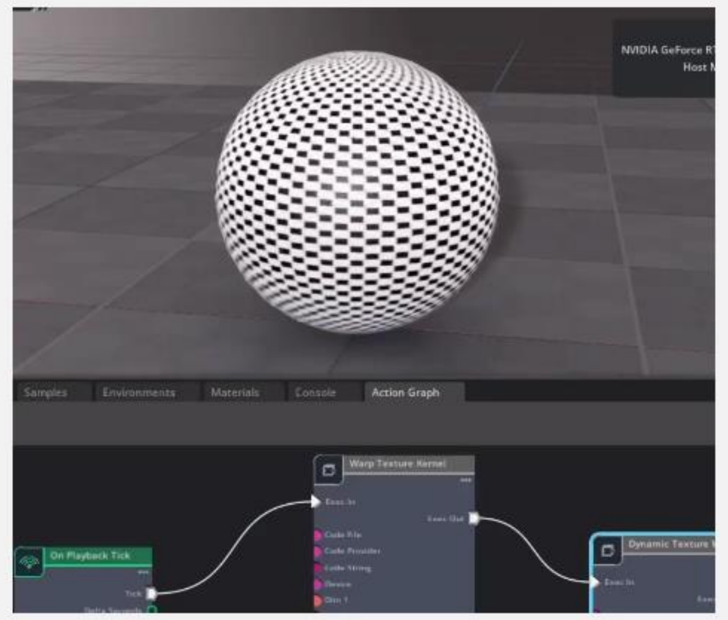
48ms

* Performance measurements on NVIDIA RTX A6000

Warp

NVIDIA Python Framework

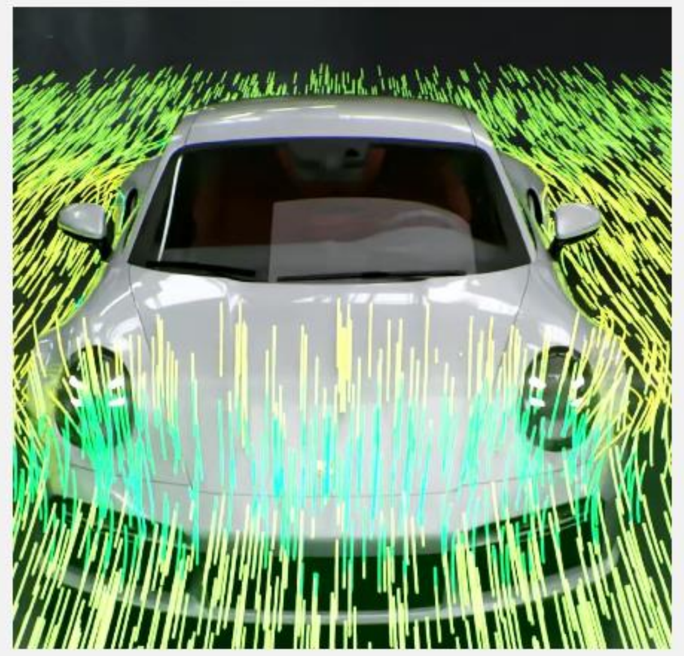
Geometry Processing



Examples:

- Mesh distance/sampling queries (Modulus)
- SDF generation (NanoVDB)
- Point-cloud processing
- GNN construction (Modulus)
- Visualization and analysis of CFD data

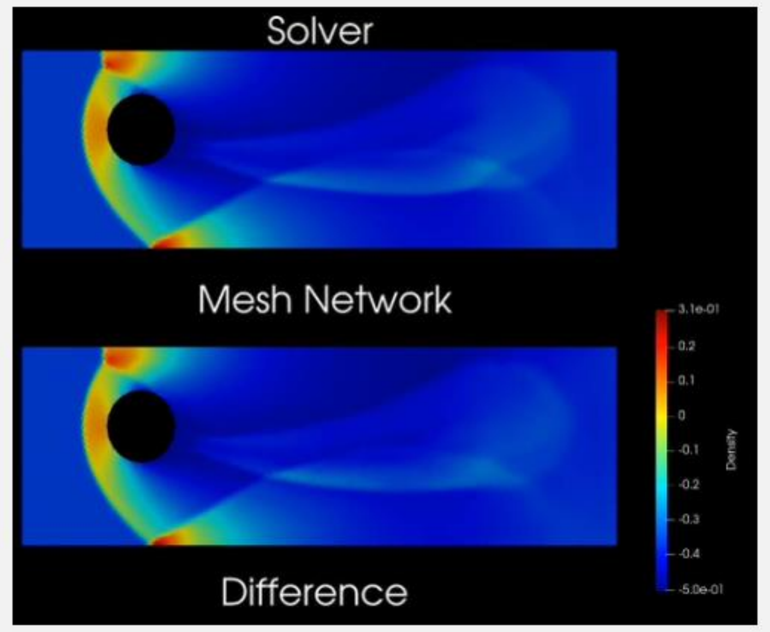
Differentiable Simulation



Examples:

- Efficient lattice-Boltzmann kernels (XLB)
- Accelerated robot policy learning (SHAC)
- Neural reduced-order methods (MIT)
- Physics NeRF (UToT)
- Accelerated FEM methods

Accelerated Models and Training



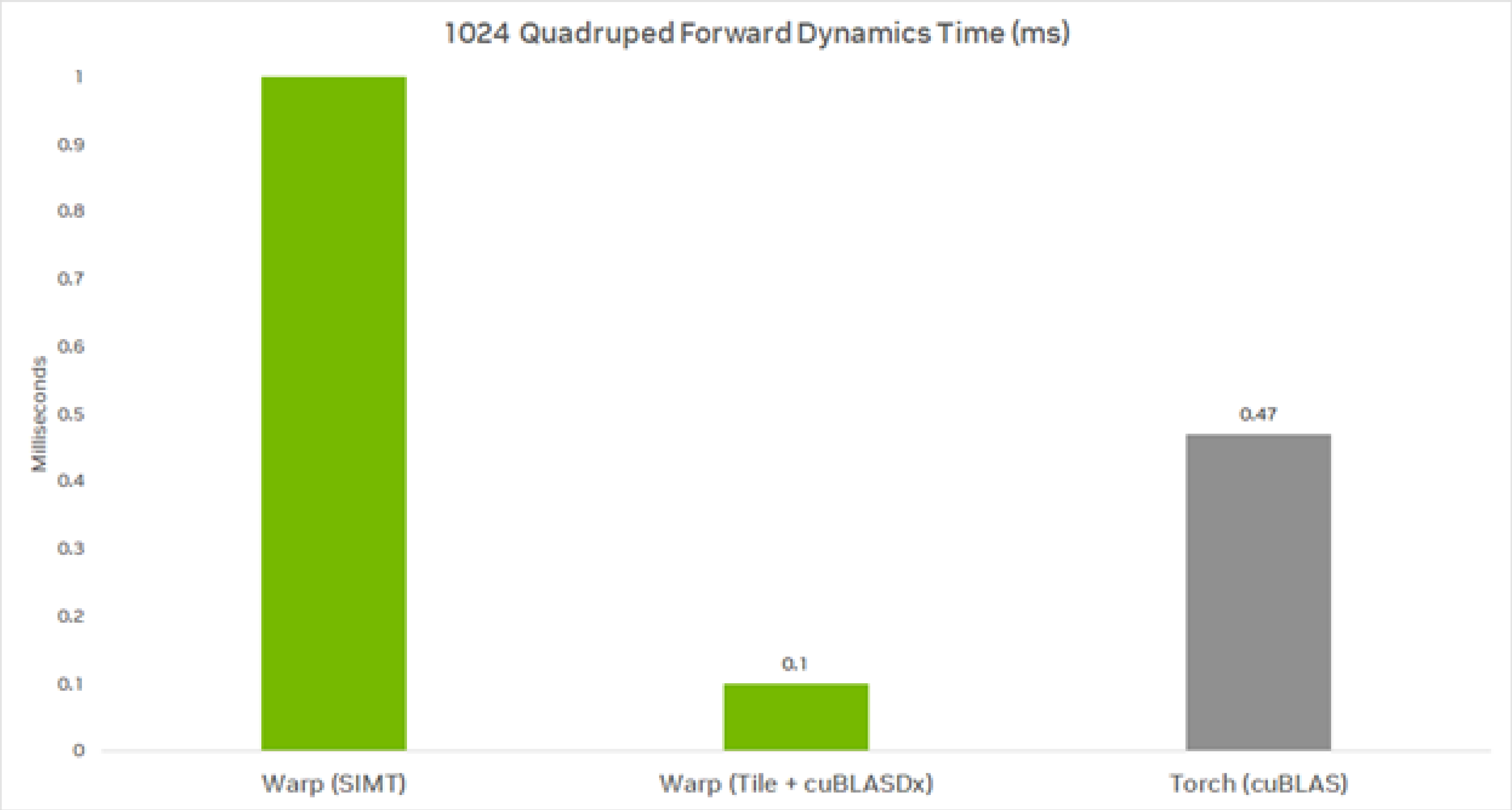
Examples:

- Image processing
- Memory-efficient ConvNet
- Accelerated RFdiffusion
- Accelerated equivariant neural networks
- Accelerated KAN networks

- Open-source **kernel-based** compute framework
  - Often a natural fit for routines found in simulation and geometry processing
- Makes it easy to write GPU simulation code in **Python**
- Thin abstraction layer over CUDA with **JIT compilation** of kernels
- Low barrier to entry, fast **iteration time**
- **Differentiable** for fast training
- Simple **interop** with NumPy, CuPy, PyTorch, JAX, and existing C++/CUDA code
- Built-in **data structures** and algorithms for spatial processing

Tile Based Programming with Warp

Warp and Math Dx Library Integration



Performance for batched robot forward dynamics using Warp’s tile primitives

Blog Post

- **Challenge**: Programming Tensor Core math units can be difficult to integrate with user programs and can lose efficiency which requires careful management of data flow between units.
- **Solution**: Math Device Extension library integration with Warp’s tile programming model, giving Warp developers access to the full power of modern GPU hardware.
  - Device-side math libraries enable seamless fusion of Tensor Core-accelerated GEMM, FFT and other tile operations within a single kernel.
    - Memory I/O reduction
    - Kernel launch overhead reduction
    - Arithmetic Intensity maximization

cuPyNumeric

Easy MGMN Distributed Accelerated Computing

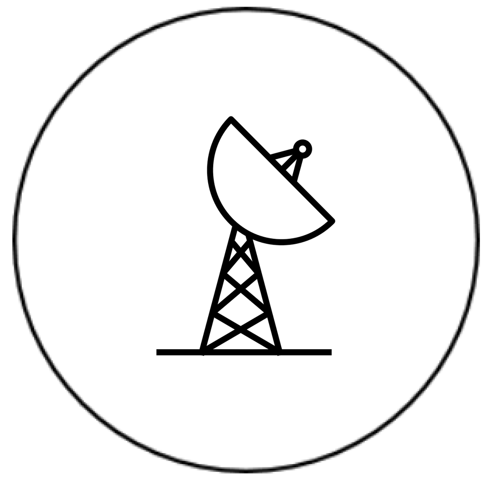
```python
#import numpy as np
import cupynumeric as np
size = 100000 #100kx100k
A = np.random.randn(size,size)
B = np.random.randn(size,size)
C = A @ B
```

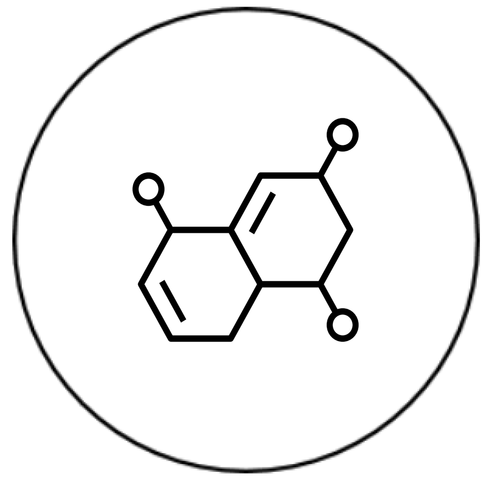
- Simple python code w/ NumPy
- No partitioning code
- No MPI code

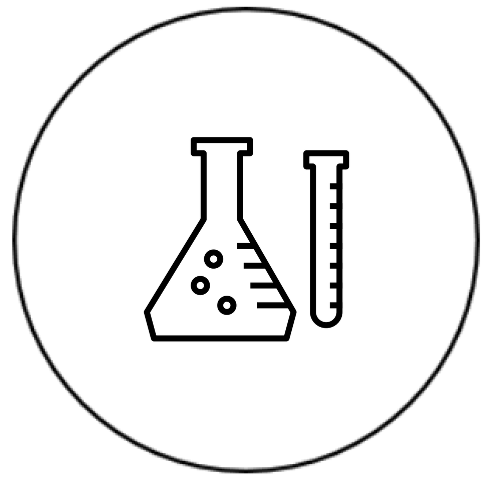
Scales to multi-GPU multi-Node at runtime

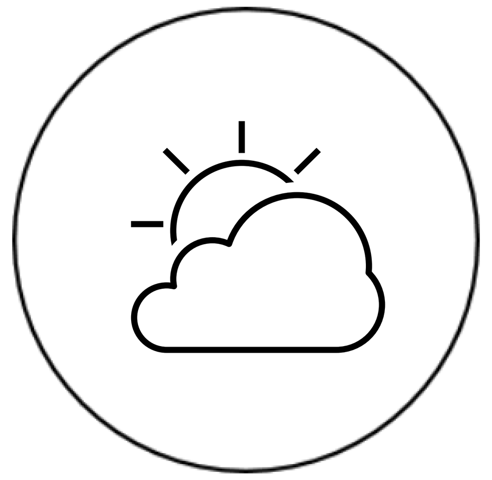
```
(legate) bod@dgx-1:~$ legate --nodes 64 --gpus 8 matmul.py
```

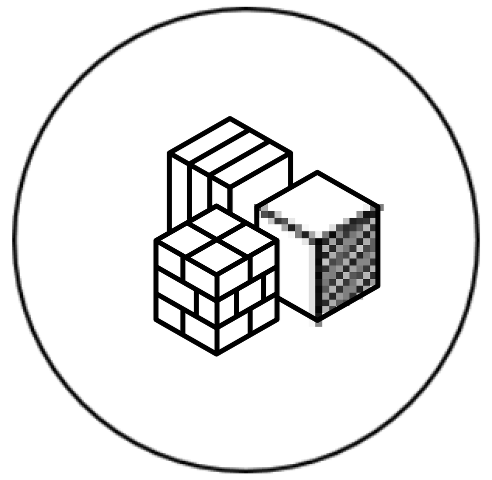
- Designed for **domain scientists and researchers**
- Write research code **simply** in **Python w/ NumPy**
- **Zero-code-change scaling** from single CPU core to a multi-GPU multi-node supercomputer w/ thousands of GPUs
- **Use HPC without code HPC** or waiting for other people to rewrite w/ MPI
- Aspiring drop-in replacement library for NumPy and **expanding to SciPy, and other scientific libraries**

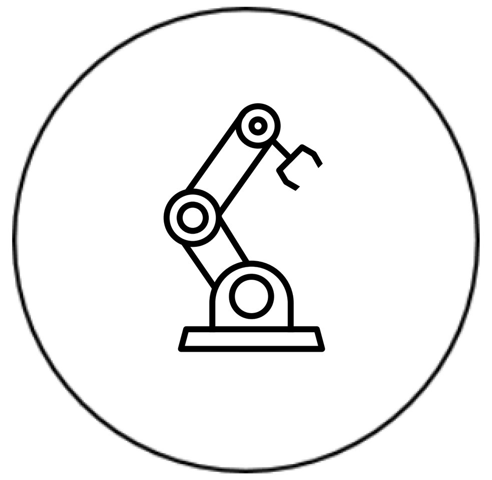

Astronomy

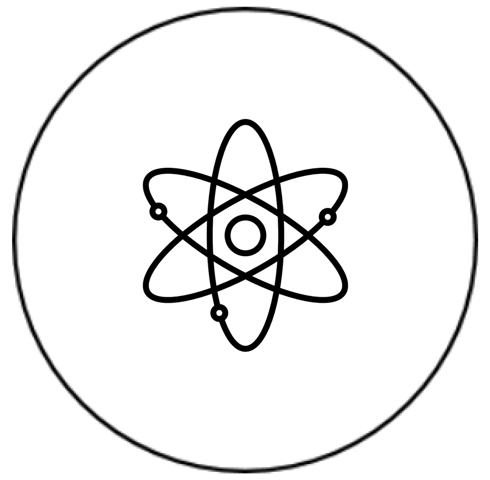

Biology

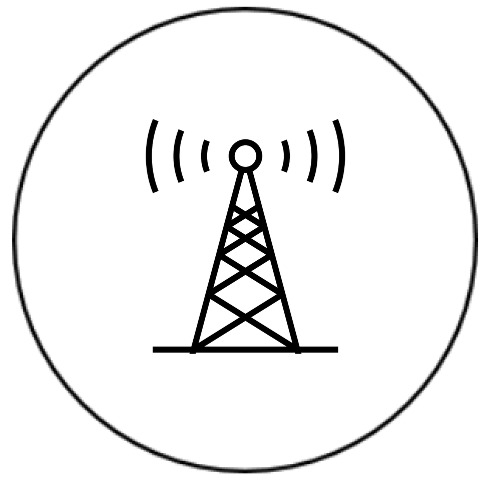

Chemistry

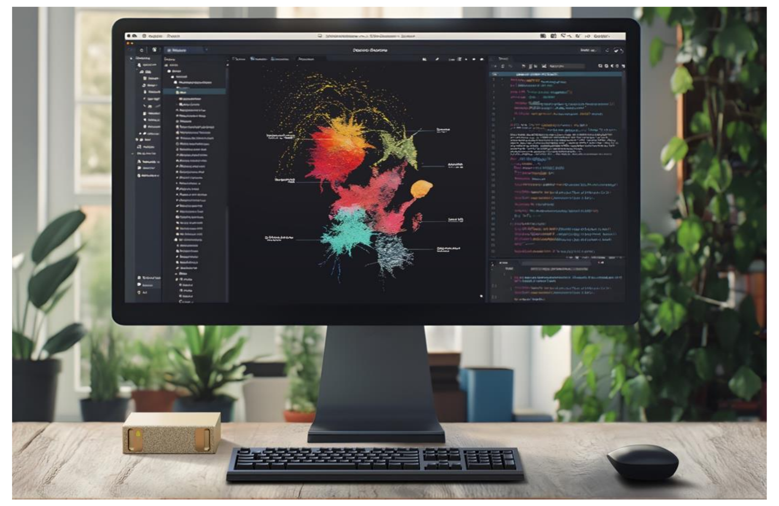

Climate

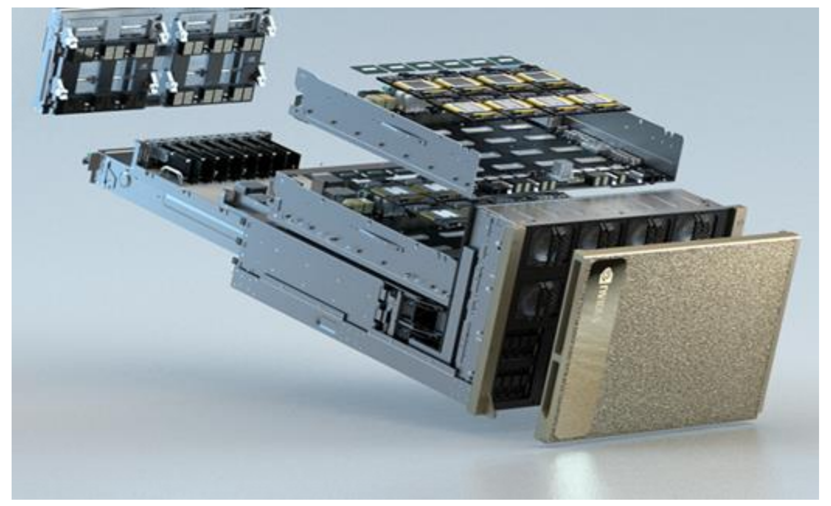

Materials


Engineering


Physics


Signal Processing

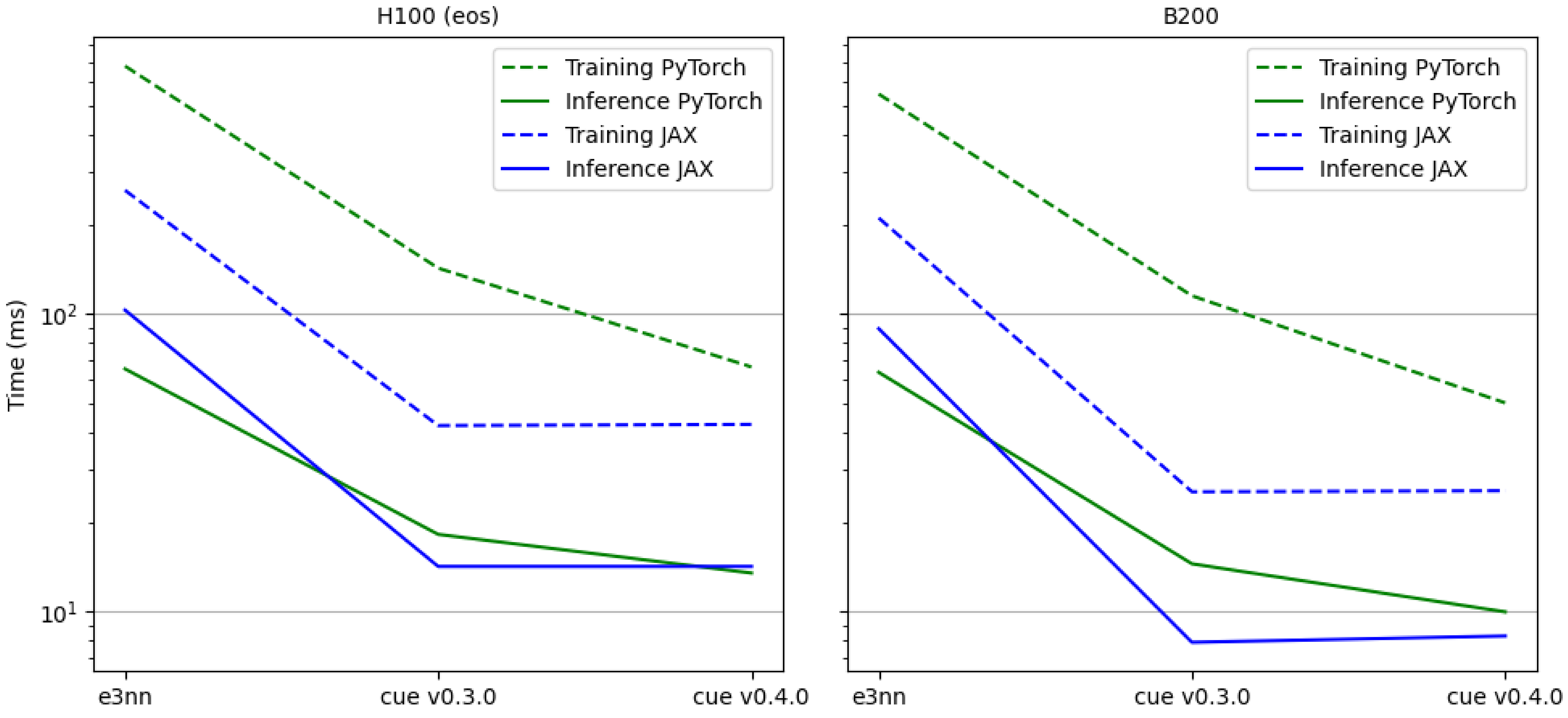




AI for HPC

cuEquivariance

AI for Science – Accelerating Equivariant Neural Networks



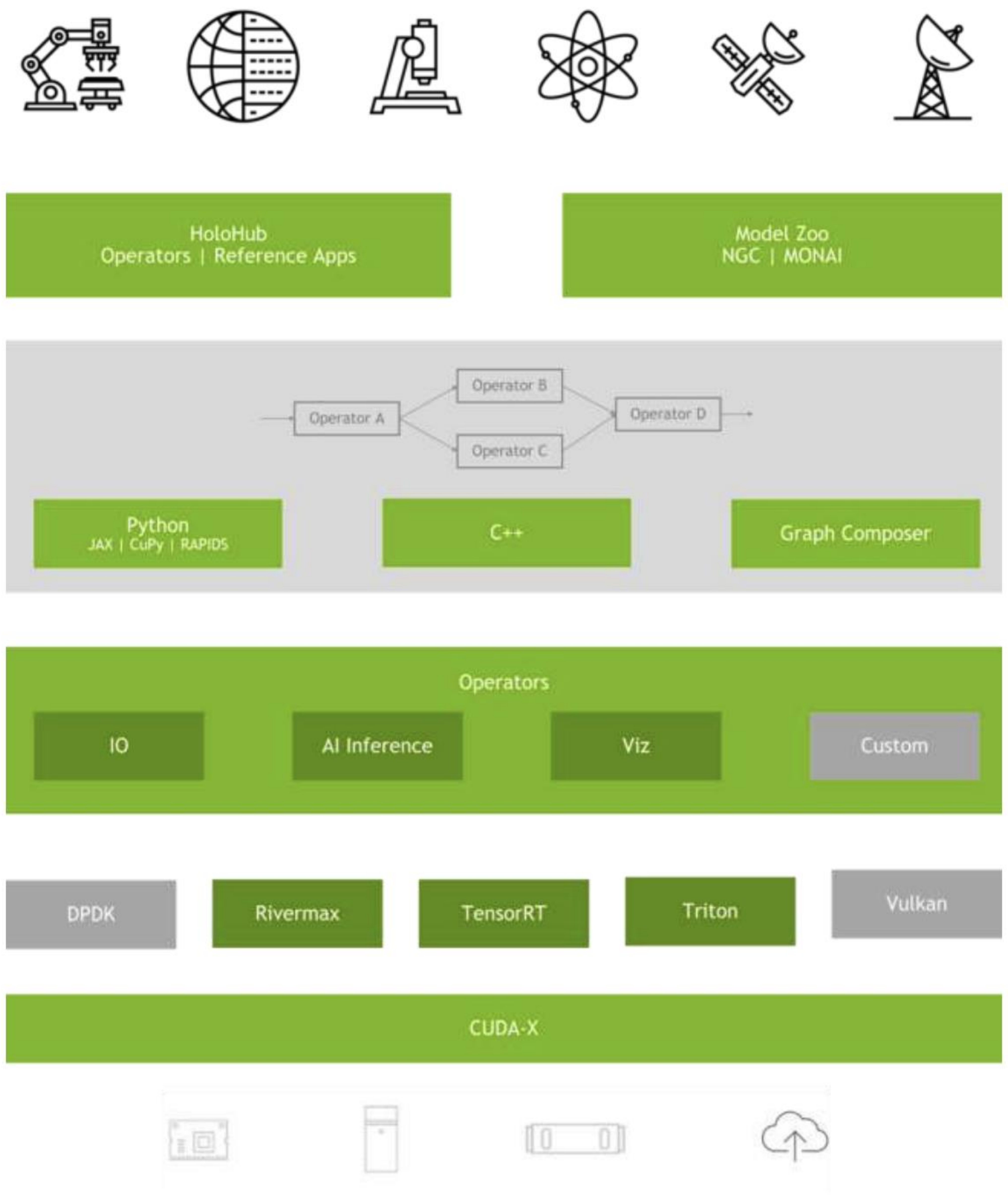
MACE-MP-L Model with 3,000 atoms and 160,000 edges

Float 32, 3,508,368 Parameters

- CUDA-accelerated building blocks for equivariant neural networks
- Acceleration across popular AI for science models
  - DiffDock
  - MACE → **directly integrated for speedups**
  - NequiP
  - Allegro
- Support for training and inference

NVIDIA Holoscan

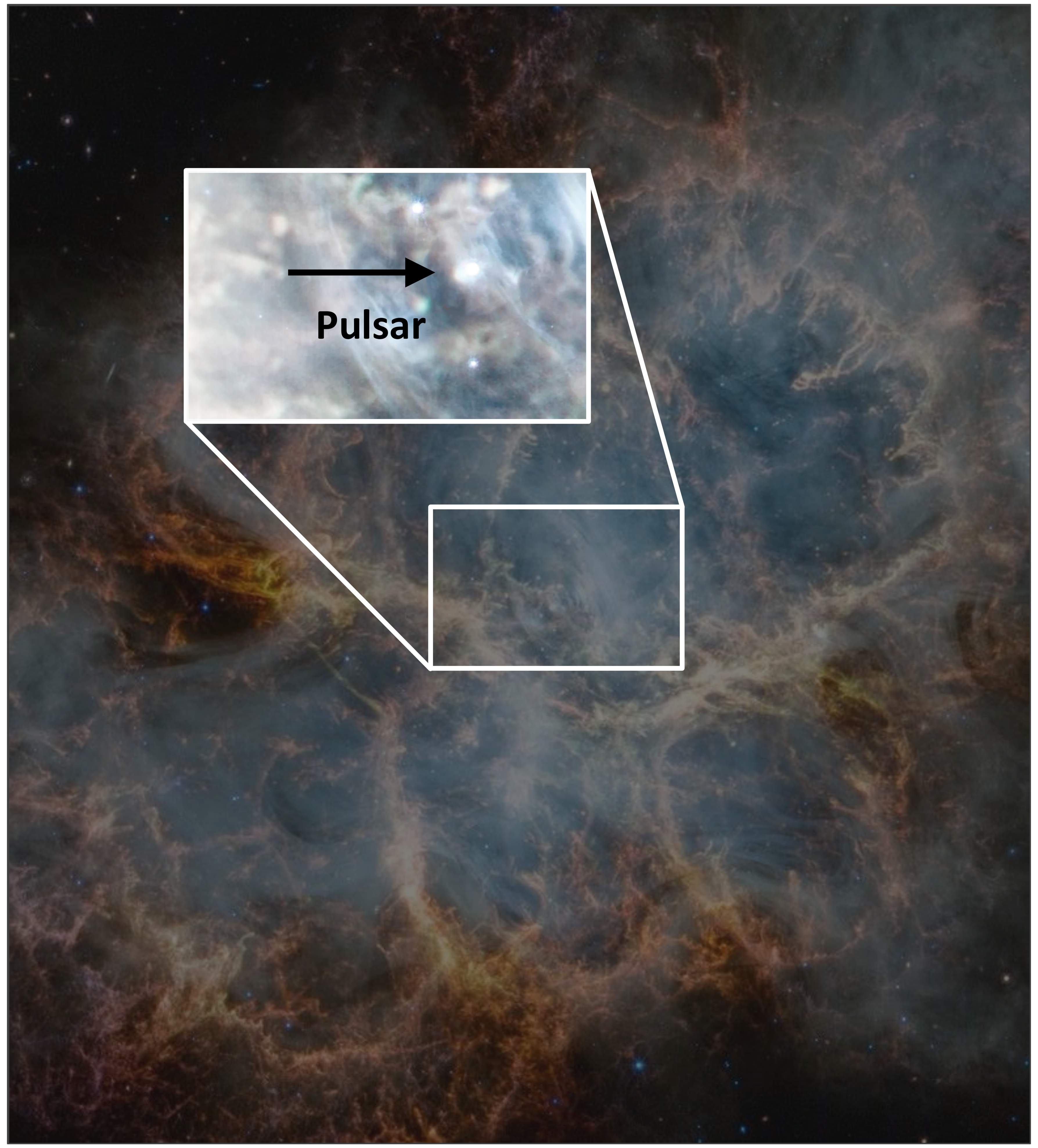
SDK for AI-Powered Real-time Processing of Streaming Data



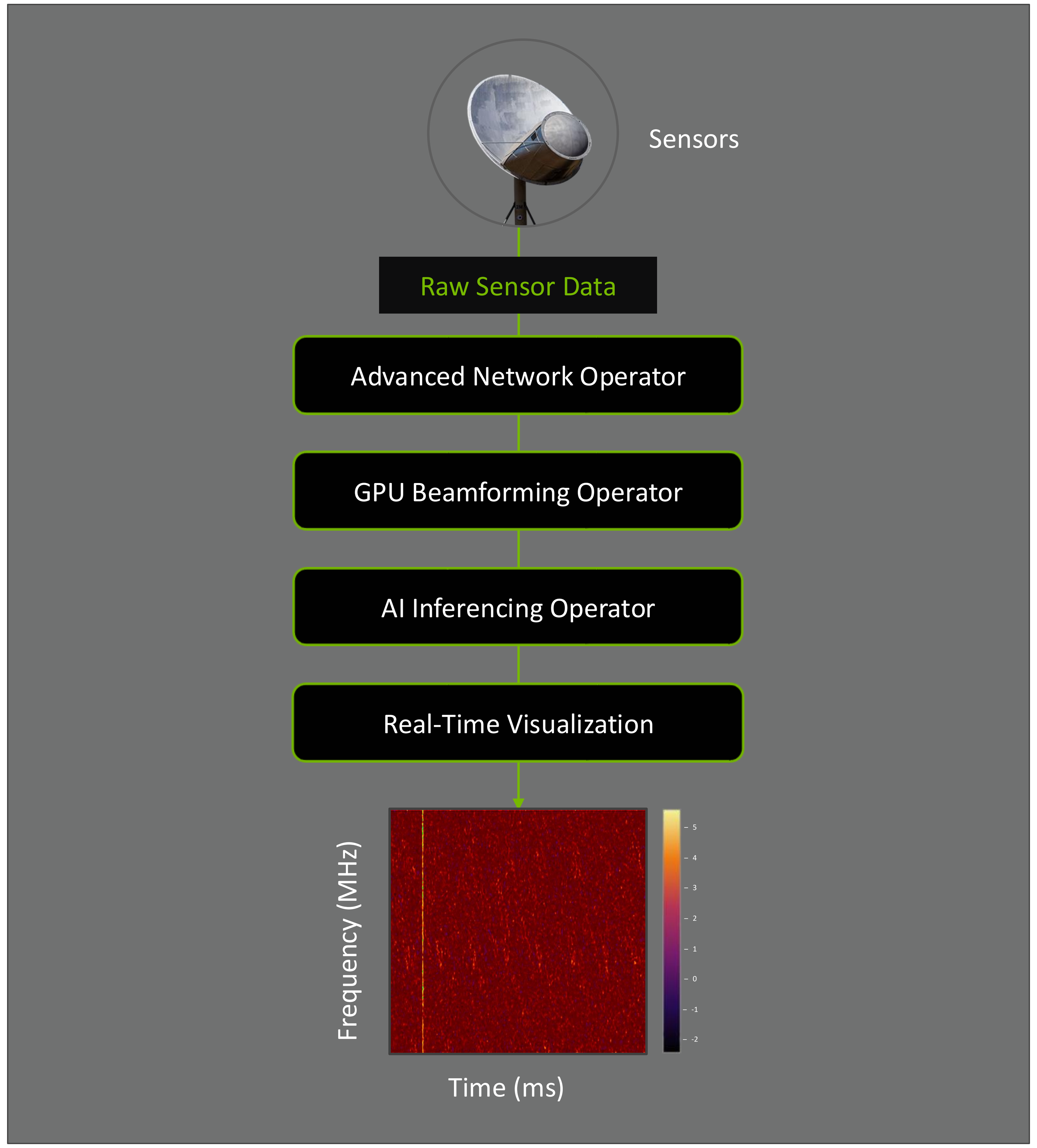
- Simplifies sensor I/O to GPU
- Simplifies the performant deployment of an AI model int a streaming pipeline
- Provides customizable, reusable, and flexible components to build and deploy GPU-accelerated algorithms
- AI inference with pluggable backends such as ONNX, Torchscript, and TensorRT

First Real-Time AI Detection of a Pulsar Using Raw Streaming Sensor Data

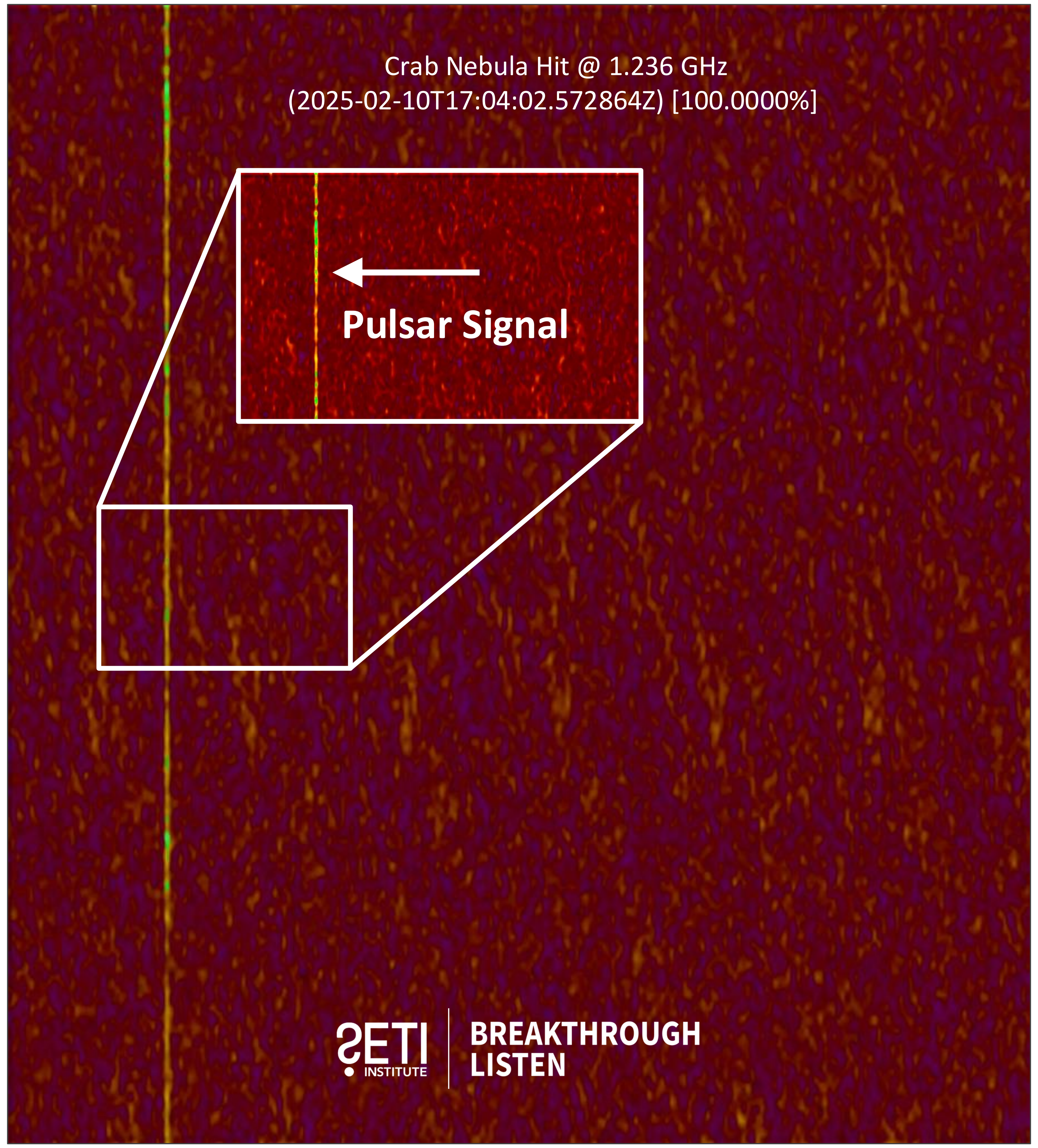
NVIDIA Holoscan enables real-time AI-powered sensor workloads at the edge



Pulsar in the Crab Nebula



Holoscan From Beamformer to AI Model



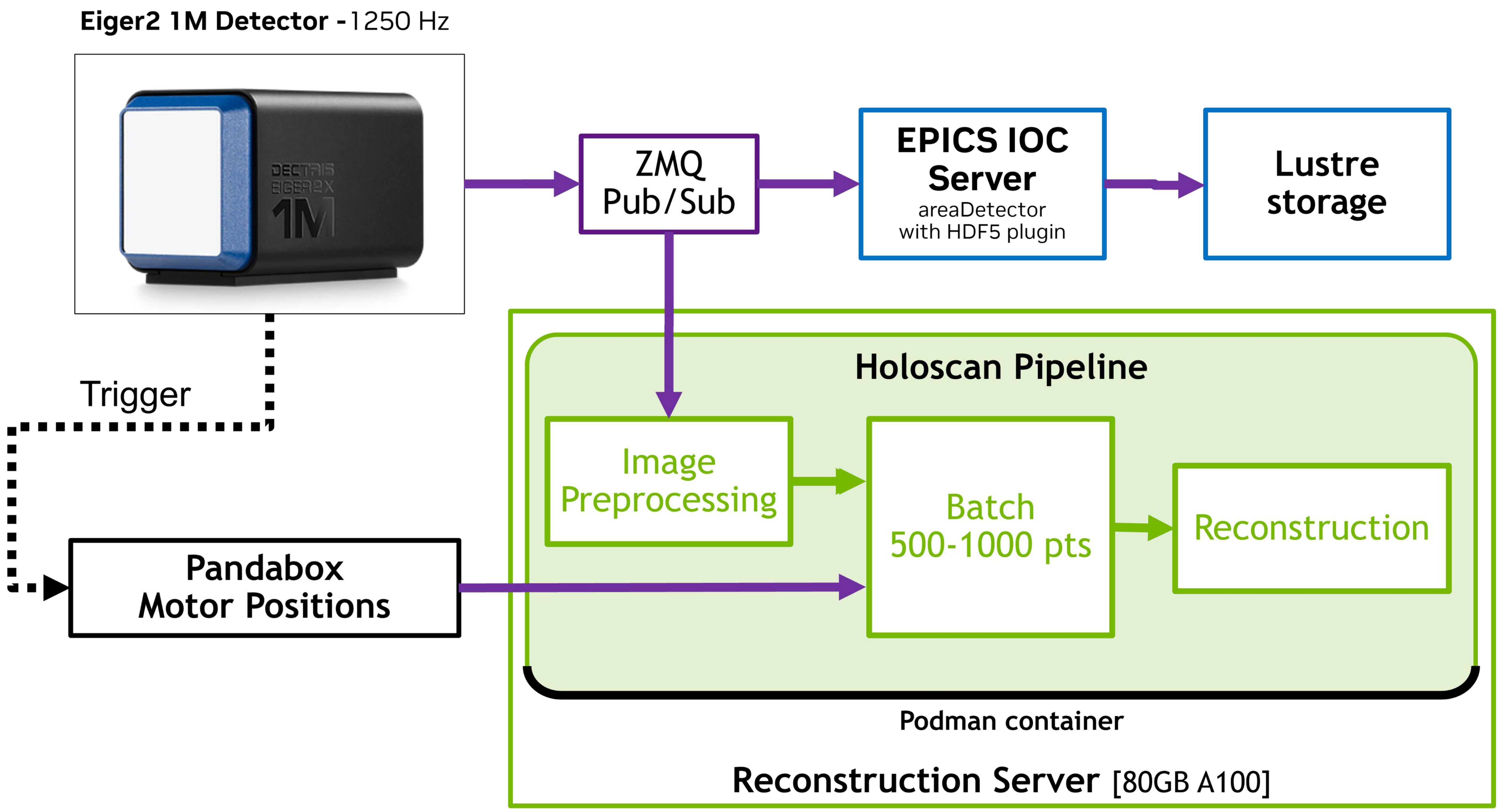
Signal Detection Beyond Noise

NSLS-II HXN Real Time Ptychography Reconstruction

Brookhaven National Laboratory

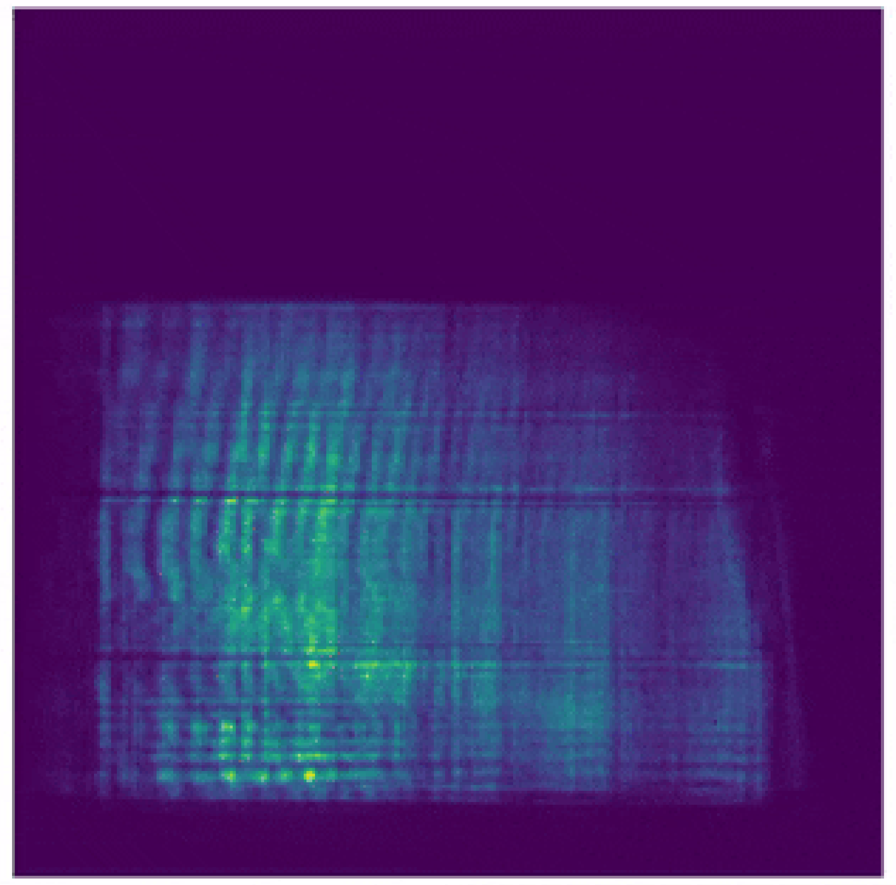
- Current beamline scan takes about 8 seconds, with reconstruction taking 1-2 minutes
    - Waits for full dataset and disk storage prior to reconstruction
- **Impact of Edge HPC for HXN**
    - Real time reconstruction provides instantaneous user feedback on experiment results
    - Ability to handle higher data rates yields higher resolution experiments and improved science products
    - Laying foundation for AI-driven agents and autonomous experimentation – **discovery of new materials at unprecedented rates**
- **Future Work**
    - Deploy to NSLS-II HXN Beamline
    - AI based adaptive scan pattern

HXN Holoscan POC Architecture

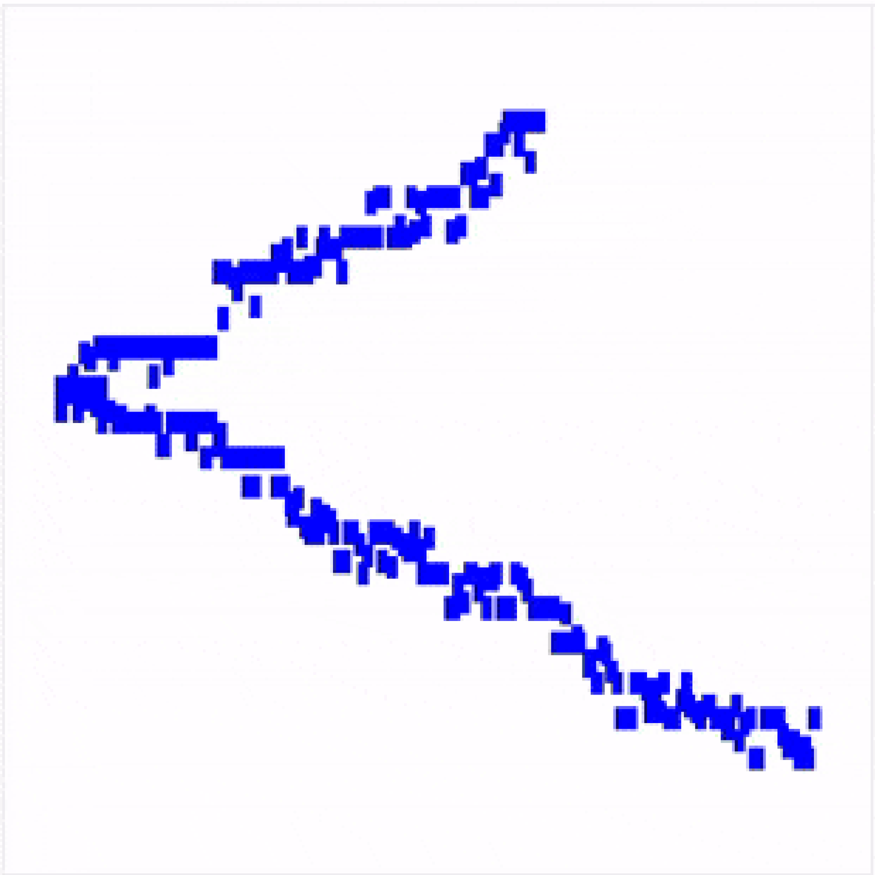


Simulated HXN Holoscan Pipeline

Eiger2 Output



Scan Pattern



Reconstructed Output

NVIDIA