GPU Python Birds of a Feather
Scipy 2025

Intro: Katrina Riehl

cuda.core, cuda.cccl, numba-cuda: Leo Fang

cuPyNumeric and Legate Sparse: Shriram Jagannathan

nvmath-python: Sergey Maydanov

Developer Tools: Daniel Rodriguez

CUDA Python Overview

cuda.core, cuda.cccl, numba-cuda

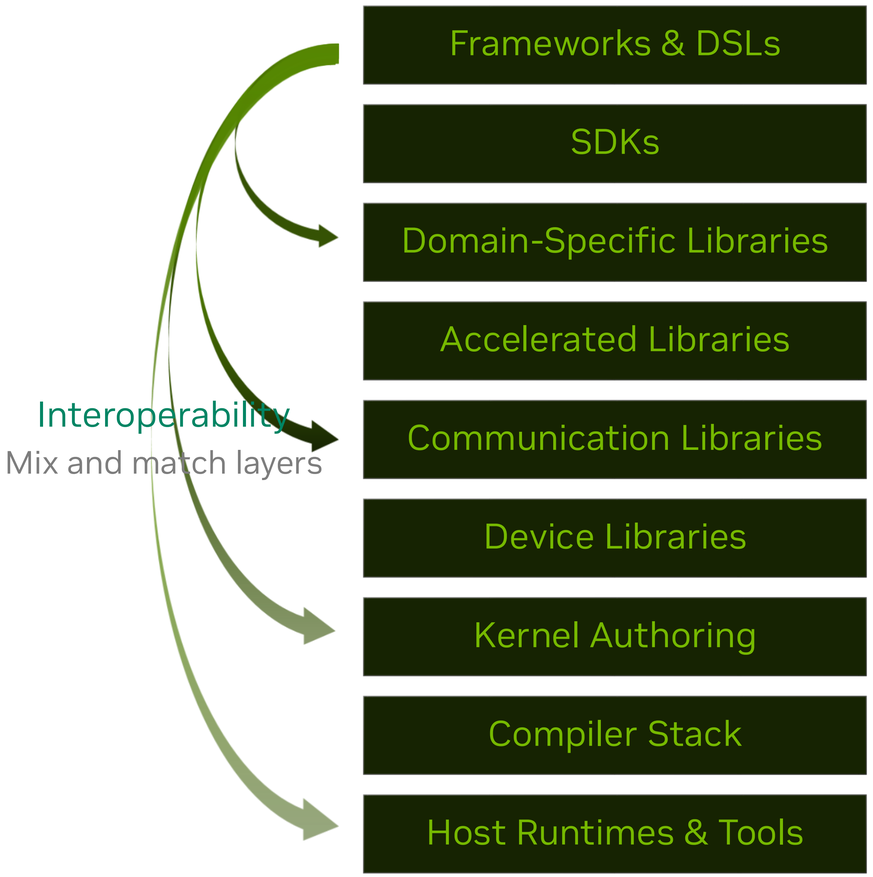
cuPyNumeric and Legate Sparse

nvmath-python

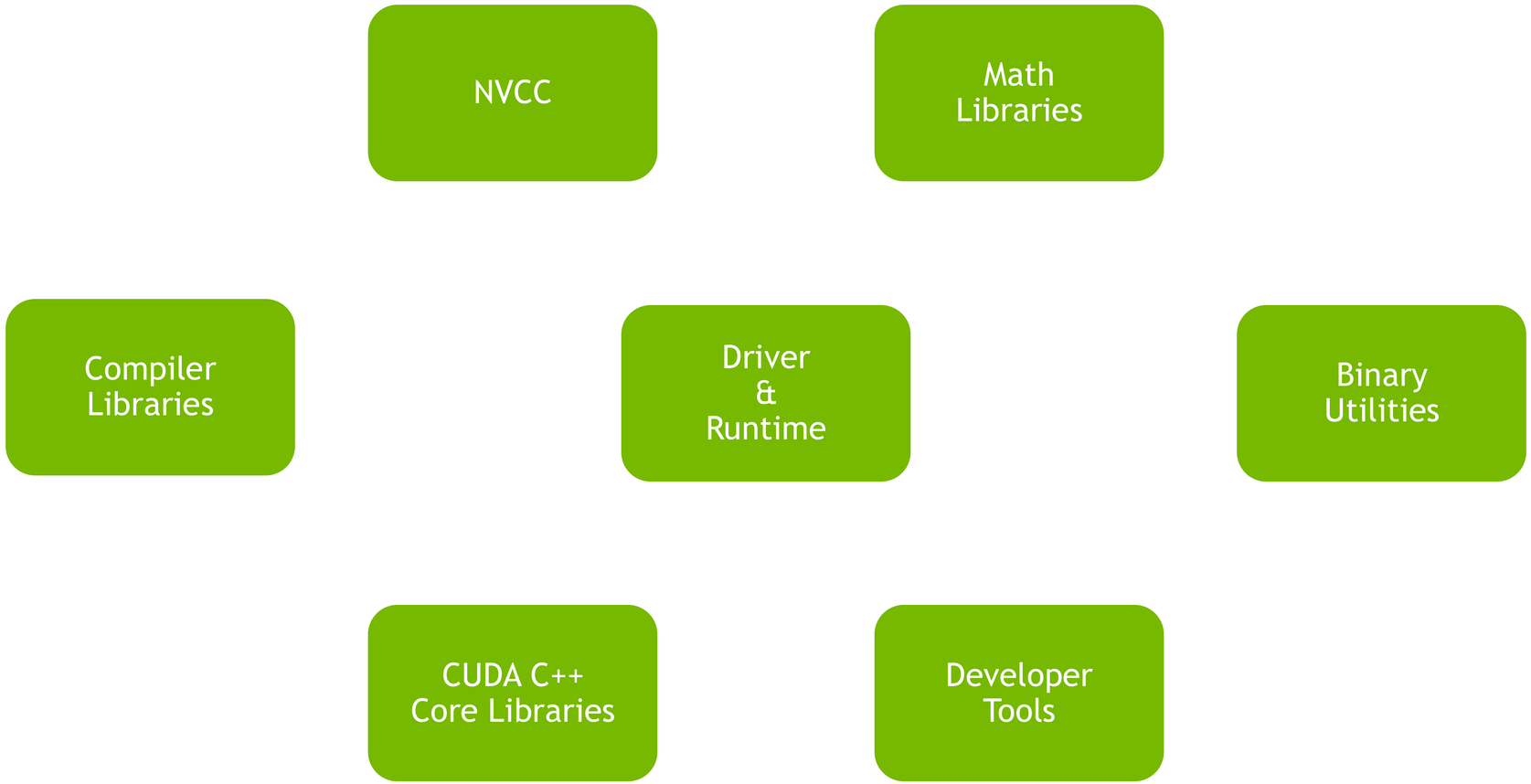
Developer Tools

CUDA Accelerated Python: First-Class Python Support Across The Stack

"First-Class" means CUDA now factors Python into every API and every feature across the platform

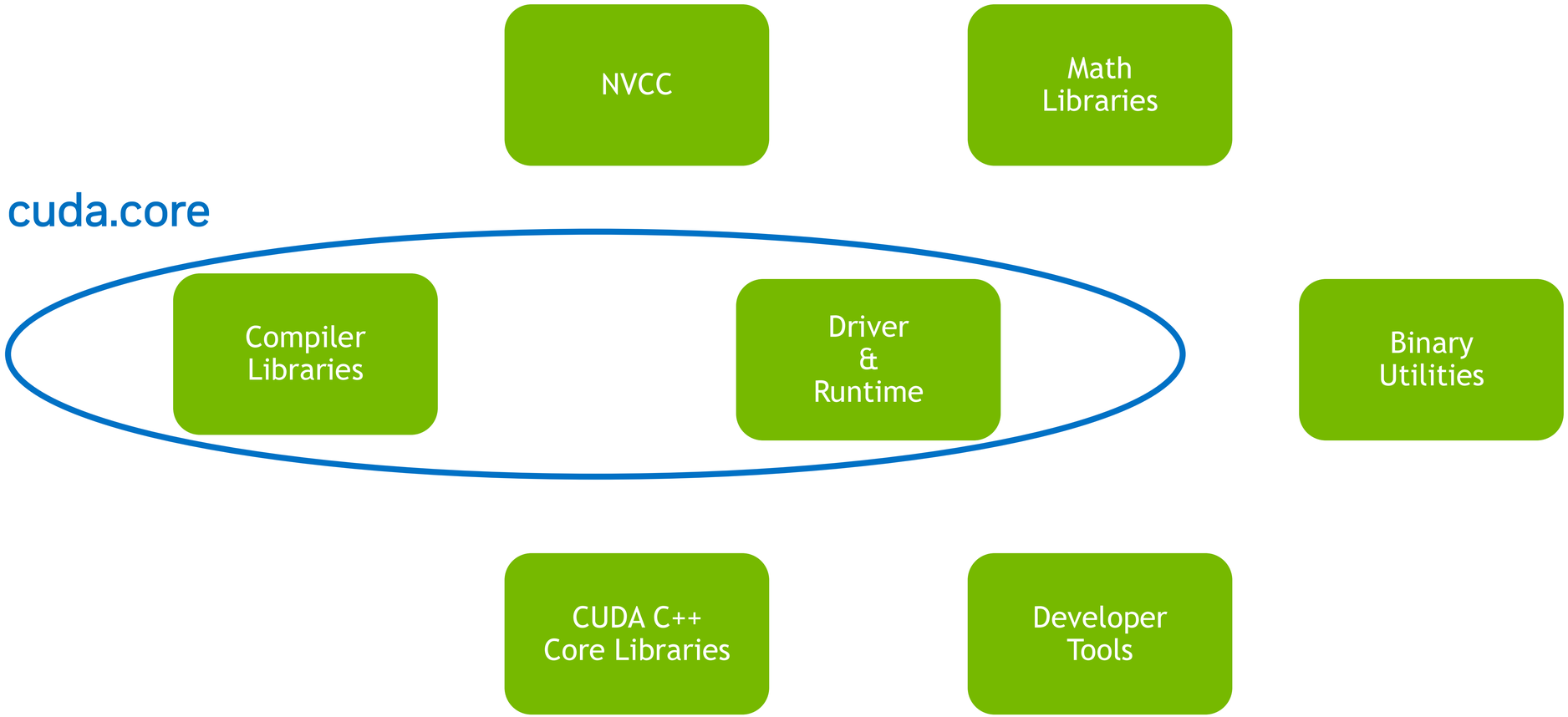


CUDA Toolkit Has a Ton of Things



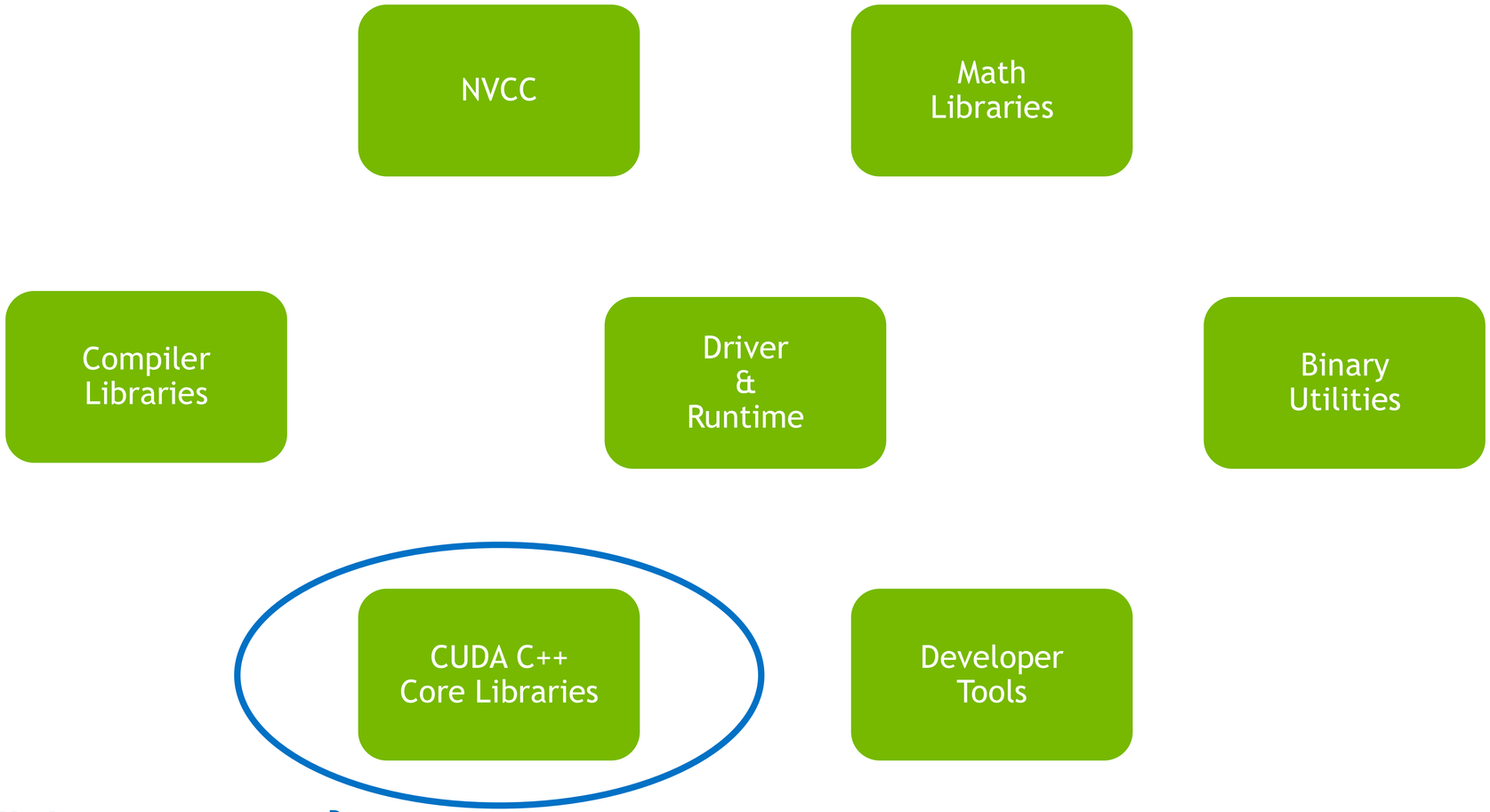
CUDA Toolkit Has a Ton of Things

These are the minimal components that you need to run your GPU code



CUDA Toolkit Has a Ton of Things

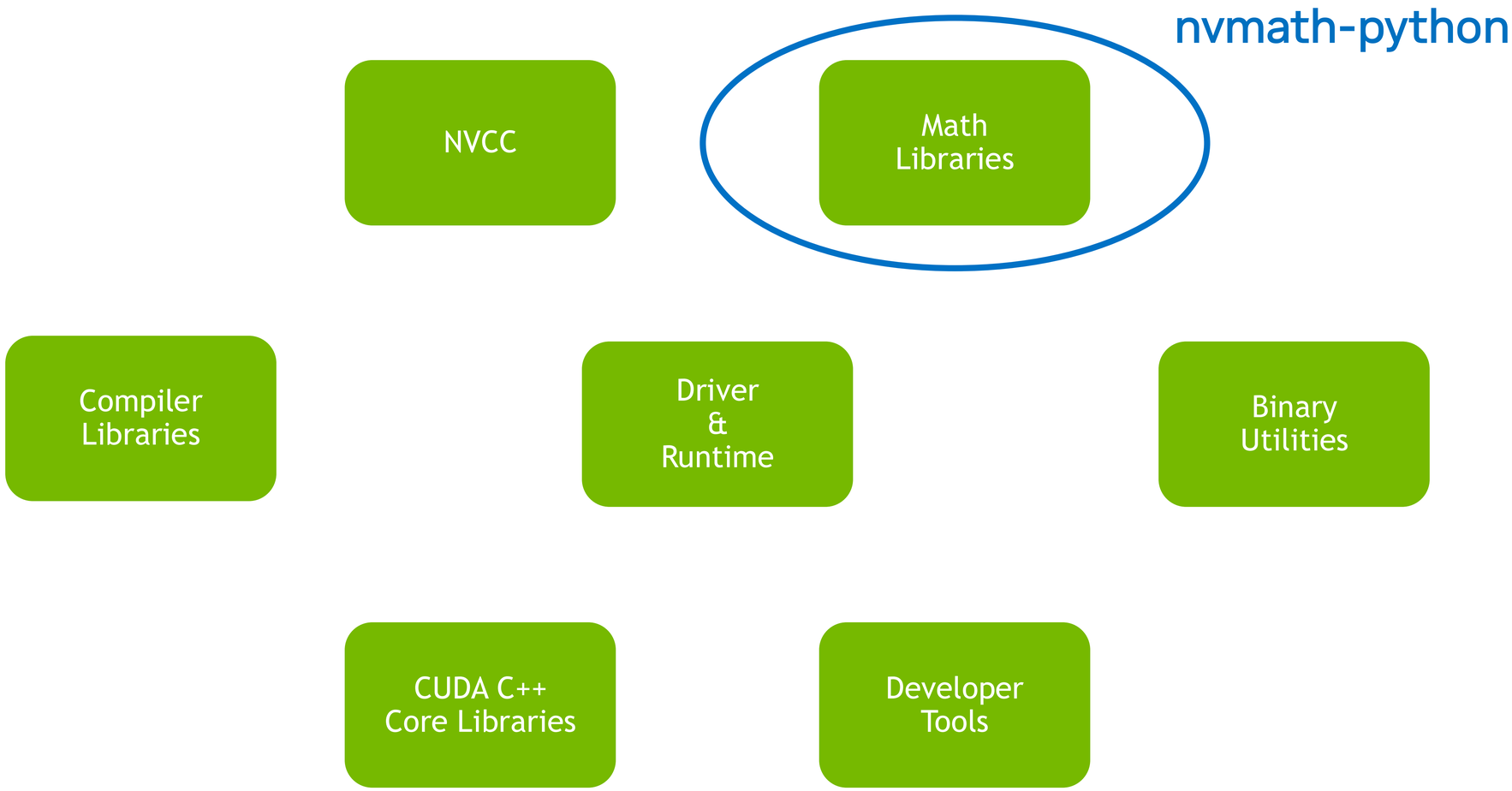
This gets you access to GPU-accelerated parallel algorithms



cuda.cccl.{parallel,cooperative}

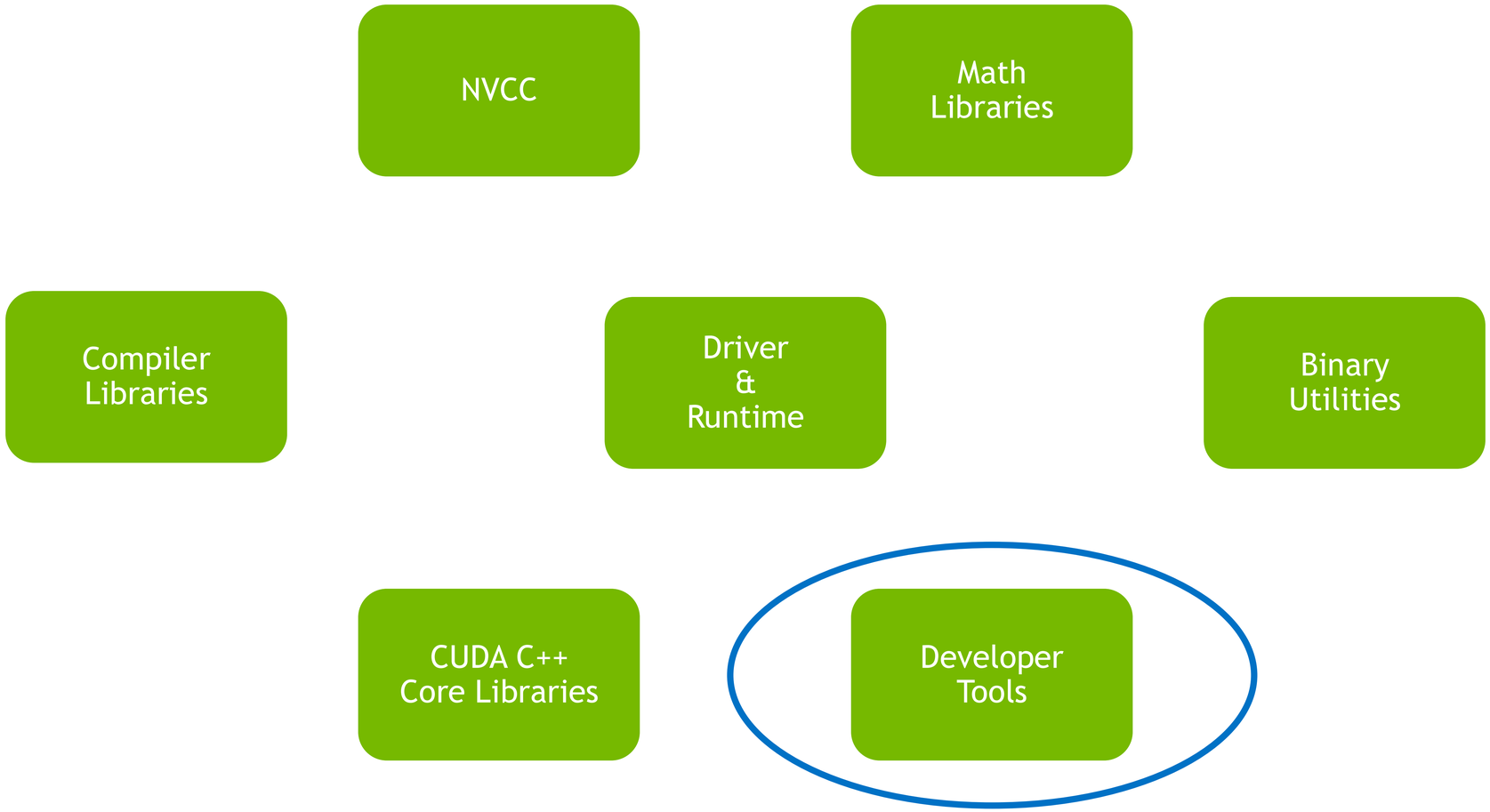
CUDA Toolkit Has a Ton of Things

This offers you highly-optimized math primitives



CUDA Toolkit Has a Ton of Things

This lets you debug and profile your GPU programs



cuda.{core,cccl} & numba-cuda

Leo Fang

cuda.core

https://github.com/NVIDIA/cuda-python

cuda.core provides pythonic access to core CUDA features

- **cuda.core.{Device, Stream, Event, Program, Linker, ObjectCode, Kernel, launch, Buffer, MemoryResource, Graph, GraphBuilder}**
- Pythonic entry points of CUDA runtime
- Pythonic exposure of runtime compilation toolchain & kernel launch
- Pythonic exposure of memory management inspired by RAPIDS Memory Manager (RMM)
- (NEW!) Pythonic interface to build and interact with CUDA Graphs

Status

- cuda.core v0.3.1: released July 2, support CUDA 11 & 12

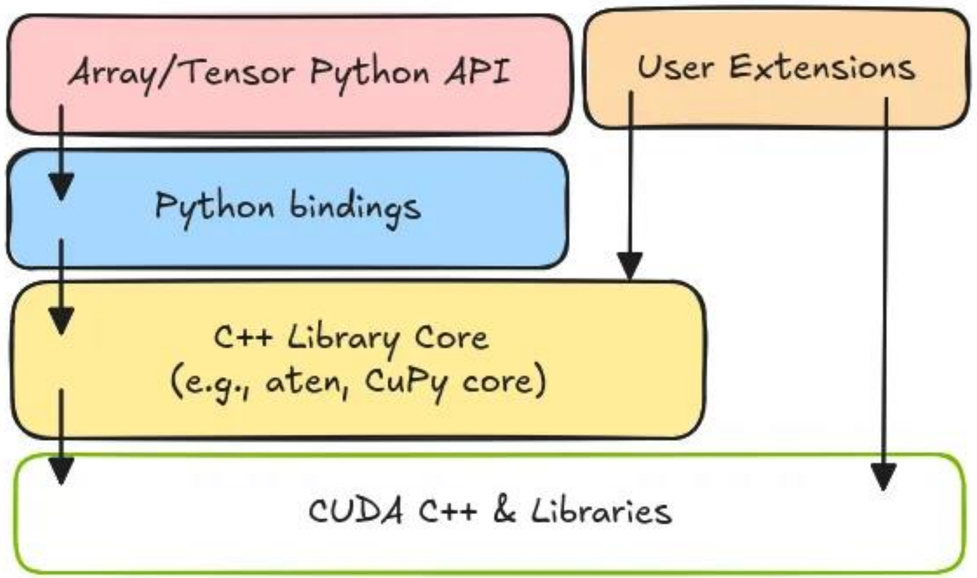
What's next?

- Expand CUDA Graphs support for memory allocation nodes and memory copy nodes
- Add support for NVVM IR as an input to Program abstraction
- Improve performance in accessing cuda.core objects
- Memory Resources initial support
- Expand CUDA Graphs support for host callback nodes
- CUDA IPC initial support
- CUDA Green Contexts initial support
- CUDA Compute-Graphics Interop initial support

```
from cuda.core.experimental import (Device, GraphBuilder,
    LaunchConfig, Program, ProgramOptions, launch,)


dev = Device(0)
dev.set_current()

code = r"""extern "C" __global__ void empty_kernel() {}"""
arch = "".join(f"{i}" for i in dev.compute_capability)
prog = Program(code, code_type="c++",
  options=ProgramOptions(std="c++17", arch=f"sm_{arch}"))
mod = prog.compile("cubin")
empty_kernel = mod.get_kernel("empty_kernel")

s = dev.create_stream()
gb = dev.create_graph_builder().begin_building()
launch(gb, LaunchConfig(grid=1, block=1), empty_kernel)
left, right = gb.split(2)
launch(left, LaunchConfig(grid=1, block=1), empty_kernel)
launch(right, LaunchConfig(grid=1, block=1), empty_kernel)
gb = GraphBuilder.join(left, right)
launch(gb, LaunchConfig(grid=1, block=1), empty_kernel)
graph = gb.end_building().complete()
graph.upload(s)

graph.launch(s)
s.sync()
```

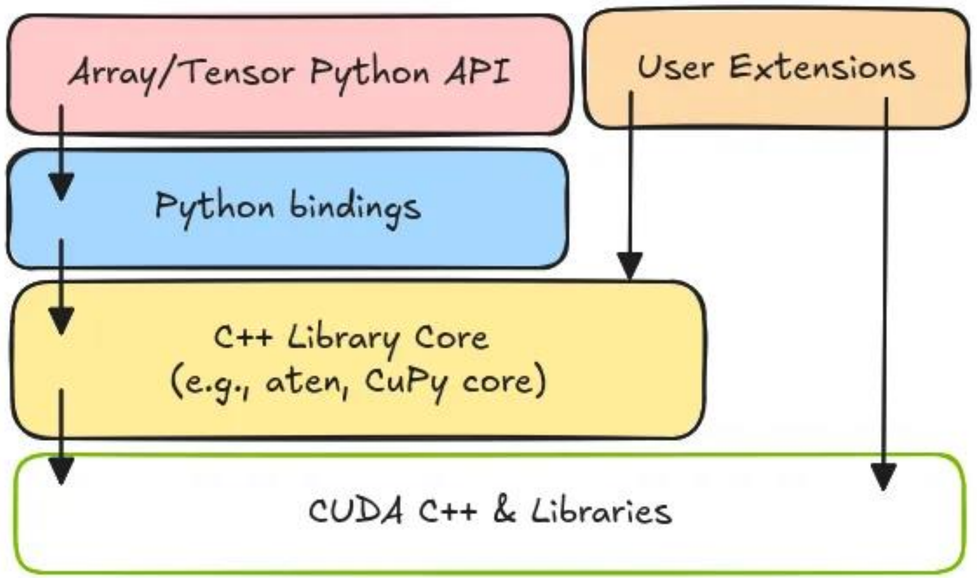
cuda.cccl

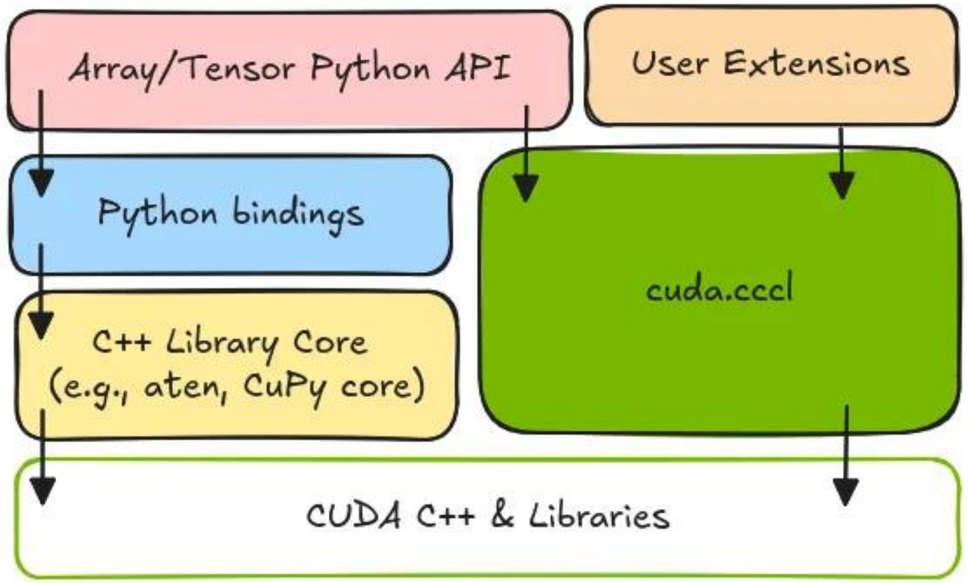
https://nvidia.github.io/cccl/python.html



cuda.cccl

https://nvidia.github.io/cccl/python.html

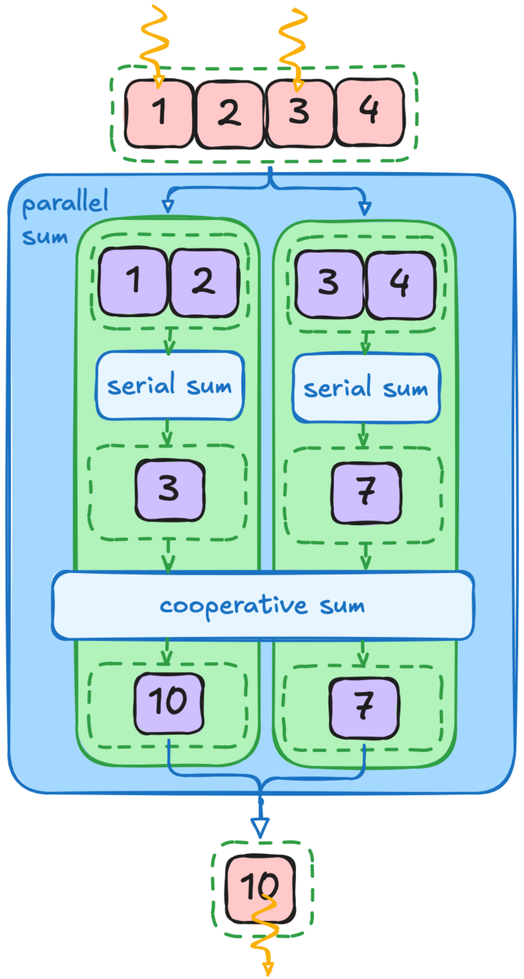


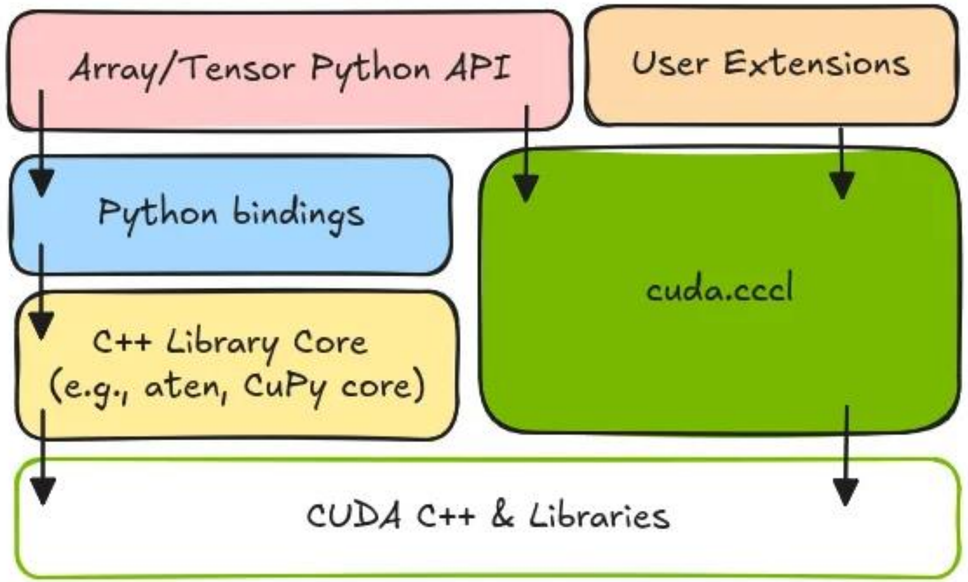


Just out:
https://developer.nvidia.com/blog/delivering-the-missing-building-blocks-for-nvidia-cuda-kernel-fusion-in-python/

cuda.cccl

https://nvidia.github.io/cccl/python.html





cuda.cccl.parallel

- Invoked in host Python code
- Works across entire GPU
- Fundamental parallel algorithms in Python
- GPU acceleration without need to drop into C++
- Built on Thrust parallel library

cuda.cccl.cooperative

- Invoked in Python kernels
- Works within warps or blocks
- Foundational algorithms for numba-cuda kernel authors
- Built on CUB cooperative device library

cuda.cccl

https://nvidia.github.io/cccl/python.html

Provide pythonic access to fundamental parallel algorithms

- Leverage investment in CCCL C++ code and expose to Python
- Fills gaps for Python accelerated library developers
  - Intentionally lower-level than Python array/tensor libraries
  - Gives the same performance and control of C++ without leaving Python
  - JIT compiled ➔ small binaries!

Status

- **`cuda.cccl`** pip packages published for linux-64
- 7x **`cuda.cccl.parallel`** host APIs
  - `scan`, `reduce_into`, `merge_sort`, `radix_sort`, `transform`, `segmented_reduce`, `unique_by_key`
- 9x **`cuda.cccl.cooperative`** device APIs (for numba-cuda kernels)
  - `scan`, `reduce`, `merge_sort`, `load/store`, `radix_sort`
- Performance parity with C++

What's next?

- **`cuda.cccl`** pip packages for linux-aarch64 & win-64
- **`cuda.cccl`** conda packages for linux-64, linux-aarch64, win-64
- **`cuda.cccl.parallel`**: `histogram`, `segmented_sort`, `reduce_by_key`
- **`cuda.cccl.cooperative`**: pythonic interface
- Initial design and prototype of **`cuda.cccl.std`** device APIs

```python
import cuda.parallel.experimental as parallel

# define some simple Python functions that we'll use later
def add(x, y): return x + y


def transform(x):
    return -x if x % 2 == 0 else x


# create a counting iterator to represent 1, 2, 3, 4, ... N
counts = parallel.CountingIterator(np.int32(1))

# create a transform iterator to represent 1, -2, 3, -4, ... N
seq = parallel.TransformIterator(counts, transform)

# create a reducer object for computing the sum of the sequence
out = cp.empty(1, cp.int32)  # holds the result
reducer = parallel.reduce_into(seq, out, add, initial_value)

# compute the amount of temporary storage needed for the
# reduction, and allocate a tensor of that size
tmp_storage_size = reducer(None, seq, out, size, initial_value)
tmp_storage = cp.empty(temp_storage_size, cp.uint8)

# compute the sum, passing in the required temporary storage
reducer(tmp_storage, seq, out, num_items, initial_value)
print(out)  # out contains the result
```

cuPyNumeric and Legate Sparse

Shriram Jagannathan

Libraries in Legate Ecosystem

Composable libraries built on a common runtime stack

Foundational Legate Libraries

NumPy SciPy scikit learn pandas

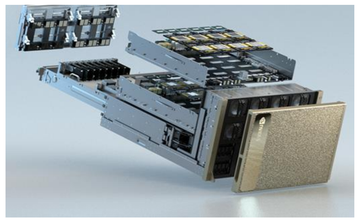
cuPyNumeric | Legate Sparse | Legate Jax | Legate ML | Legate Dataframe | Legate IO | . . .

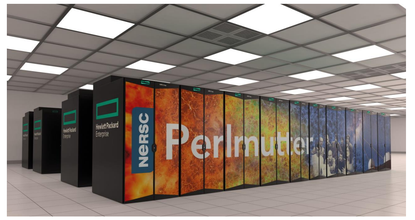
Legate Runtime Stack

Legate - Productivity & Composability Layer

Legion - Implicit Parallelism Layer

Realm - Runtime for Scalable and Portable Execution

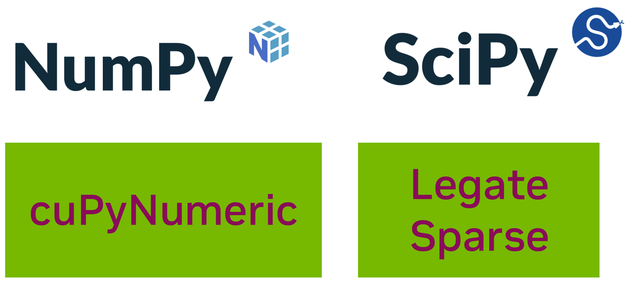





Motivation

Transparent acceleration with familiar APIs

- Multi-GPU simulations necessary as problem sizes become large.
- NumPy and SciPy: widely used single threaded libraries
- CuPy provides impressive acceleration for single-GPU use-cases
- Currently, multi-GPU simulations will need, e.g., mpi4py + CuPy
- Need for a library that provides transparent acceleration with familiar API
- No explicit data partitioning, data movement, data synchronization
- Parallelism taken care by the library/runtime
- Help scientists and researchers focus on science domain



An Example

Run on Multiple GPUs/Nodes with Zero Code Change

```
import cupynumeric as np
import legate_sparse as sparse

N = 10000000 # 10 Million

A = np.random.rand(N)
B = np.random.rand(N)
C = sparse.diags(…, shape=(N,N))
X = np.ones(N)

out = A @ B # dense mat-mat multiply
out = C @ x # sparse mat-vec multiply
```

Run on multiple GPUs

```
legate --gpus 2 ./main.py
```

Run on multiple OMP threads

```
legate --omps 1 --ompthreads 4 ./main.py
```

Run on multiple nodes

```
legate --nodes 2 --gpus 8 ./main.py
```

Concurrent Executions on Different Processor Types

Explicitly map computations to resources

- Query machine to get available resources (e.g., CPUs/GPUs)
- Target computations to specific resources
  - Create a matrix and parallelize using OMP threads
  - Solve multiple times on the GPU
- Control number of workers explicitly
- Concurrent execution on multiple resources also possible

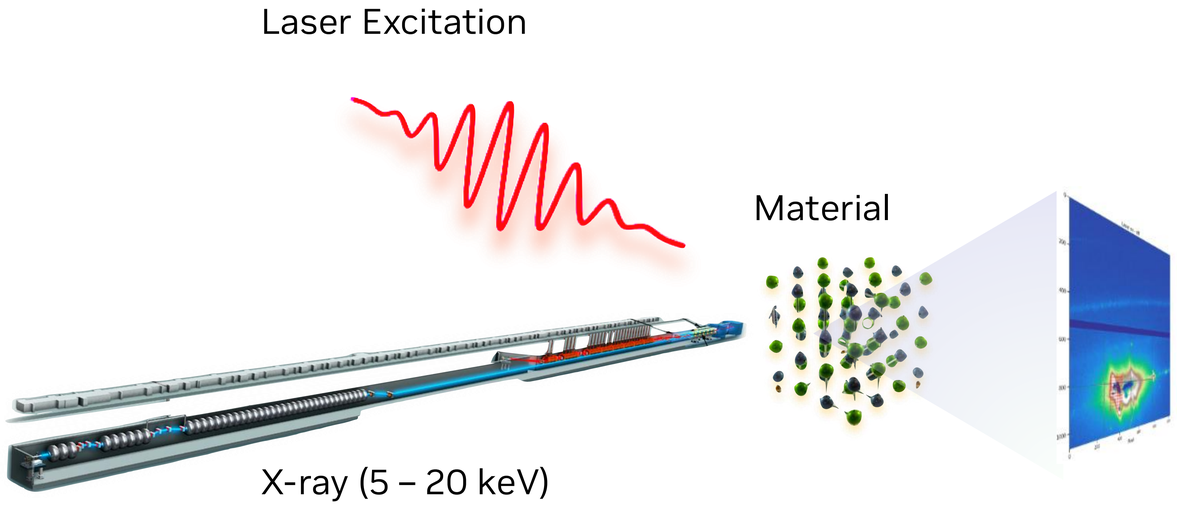
Runs on **eight** OMP threads

Runs on **two** GPUs

```
from legate.core import TaskTarget, get_machine

all_devices = get_machine()

build_procs = all_devices.only(TaskTarget.OMP)
solve_procs = all_devices.only(TaskTarget.GPU)

with build_procs:
   A, b = build()

with solve_procs:
   x = solve(A, b)
```

```
legate --omps 1 --ompthreads 8 --gpus 2 ./main.py
```

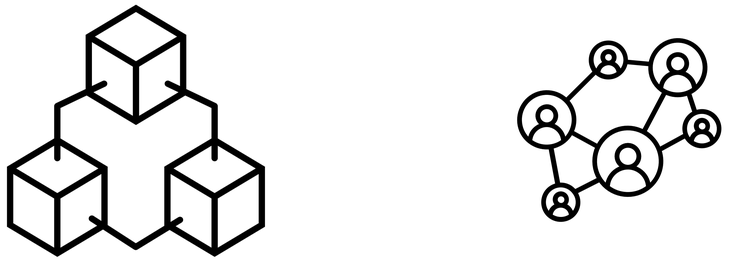
Early Adopters



SLAC National Accelerator Laboratory

Data Analytics from hours to minutes

Money Laundering Detection by
Transaction Path Tracking



National Payments Corporation of India

Computation from hours to seconds

Give it a try and let us know!

Legate Sparse

(https://github.com/nv-legate/legate-sparse)

Your feedback to

legate@nvidia.com

is highly appreciated.

Nvmath-python: Celebrating the 1st anniversary @ SciPy

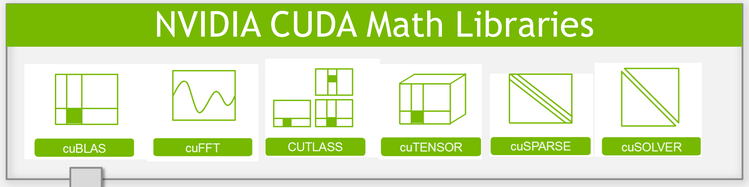
Sergey Maydanov

SciPy 2024

Beta 1

- FFT and specialized matmul APIs for host and device kernel programming with numba-cuda
- Interoperability with NumPy, CuPy, PyTorch tensors

Yet another math library? Yes



cuBLAS
cuBLASLt
cuBLASDx
cuBLASXt

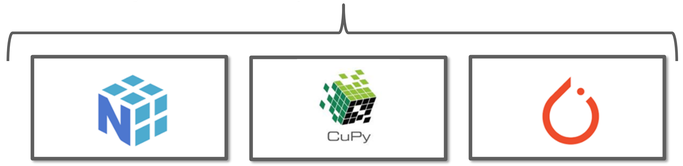
Every hammer has its purpose

ϟHammer Stock photos by Vecteezy

One of key ideas behind: Do not be constrained to NumPy-like API hammer, use appropriate hammers as needed

- Problem size: Tiny problem in shared memory? Medium size problem that fits in a single GPU memory? Large problem that requires multiple GPUs?
- Want to amortize planning & autotuning costs over multiple computations?
- Want to reach extreme efficiency by fusing multiple kernels into a single one?
- …

Complementary to & interoperable with core tensor libraries



Stateless APIs for convenience, stateful for peak performance

Stateless API

```
import torch
import nvmath

a = torch.rand(m, k)
b = torch.rand(k, n)

result = nvmath.linalg.advanced.matmul(a, b)
```

Stateful API

```
import torch
import nvmath

a = torch.rand(m, k)
b = torch.rand(k, n)

with nvmath.linalg.advanced.Matmul(a, b) as mm:
    mm.plan()
    result = mm.execute()
```

Pythonic programming model for custom kernel writers

numba-cuda

```
@cuda.jit(link=FFT.files)
def custom_kernel(signal : cp.array):
    …
    # Forward FFT (inplace).
    FFT(signal_rmem, shared_mem)
    …
```

SciPy 2024

Beta 1

- FFT and specialized matmul A
  for host and device kernel
  programming with numba-cu
- Interoperability with NumPy, C
  PyTorch tensors

CPU execution space

- Applications targeting both CPU and GPU
- Applications targeting hybrid execution
- …

Custom FFT callbacks

- Write custom GPU callback code in Python, not as a CUDA string

Tile programming with Warp

- Go beyond Numba, allow writing math-intensive code with simpler programming model of Warp compiler

SC24 Atlanta, hpc GA, creates.

Beta 2 and Beta 2.1

- Added CPU execution space based on FFT libraries conforming to FFTW3 API (NVPL for aarch64, MKL for x86)
- Added support for prolog and epilog callbacks for FFT, written in numba-cuda
- Warp tile programming using math library primitives

Pythonic programming models for custom kernel writers

numba-cuda

```
@cuda.jit(link=FFT.files)
def custom_kernel(signal : cp.array):
    …
    # Forward FFT (inplace).
    FFT(signal_rmem, shared_mem)
    …
```

NVIDIA Warp

```
@wp.kernel
def custom_kernel(signal: wp.array2d(dtype=wp.vec2d)):
    …
    # Forward FFT (inplace) on the tile.
    wp.tile_fft(signal_tile)
    …
```

NVIDIA CUDA Math Libraries

cuBLAS | cuFFT | CUTLASS | cuTENSOR | cuSPARSE | cuSOLVER

NVIDIA Performance Libraries for Grace CPU

BLAS | FFT | TENSOR | SPARSE | ScaLAPACK

Intel MKL for x86

BLAS | FFT | Sparse BLAS | ScaLAPACK

Peak GPU performance from native math libraries

Consistent CPU experience and productivity

NVIDIA

SciPy 2024

Beta 1

- FFT and specialized matmul API for host and device kernel programming with numba
- Interoperability with NumPy, PyTorch tensors

GTC 2025

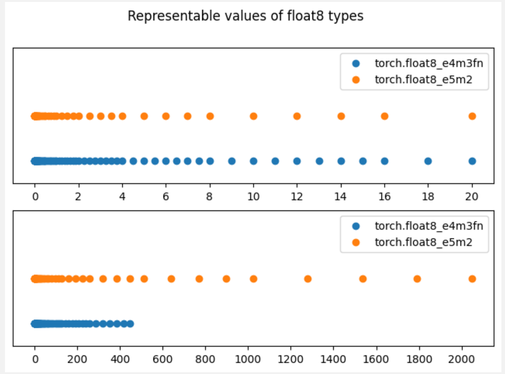
Beta 3

- Support for narrow floating-point types, FP8 and MXFP8, in nvmath-python specialized matmul

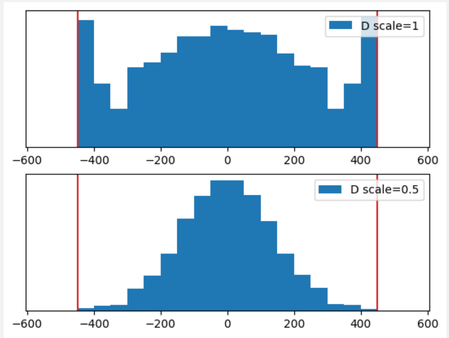
Beta 2.1

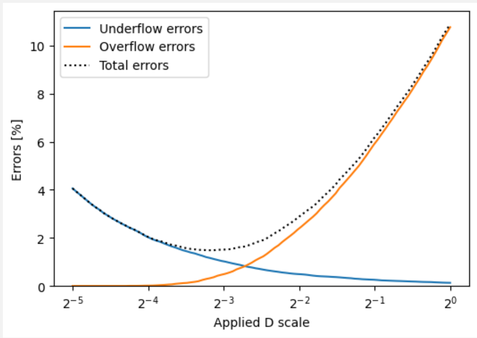
- e based on FFT
- W3 API (NVPL for
- nd epilog callbacks
- cuda
- ng math library

Still floating-point but at tiny range

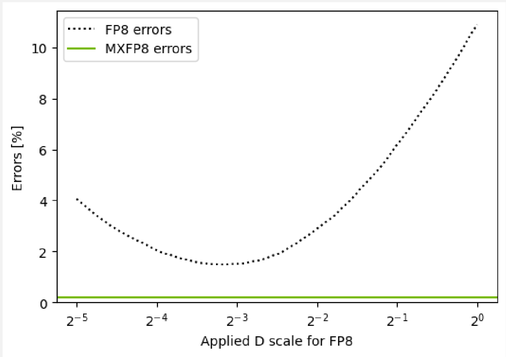


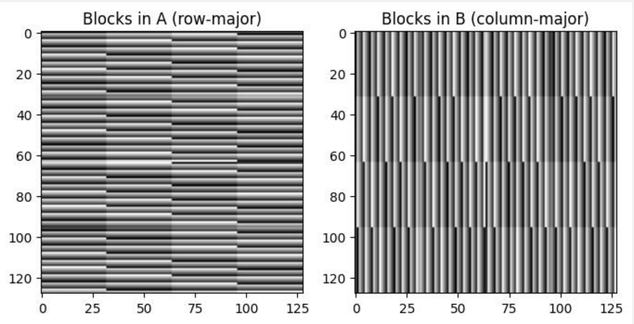
Must be scaled to avoid saturation on the left (underflows) and on the right (overflows)





Using block-scaling with MXFP8 (vs. tensor-wide scaling with FP8) allows to increase dynamic range and minimize overflows and underflows yet maintaining peak FP8 performance





SciPy 2024

Beta 1

- FFT and specialized matmul APIs for host and device kernel programming with numba-cuda
- Interoperability with NumPy, CuPy, PyTorch tensors

Beta 4

- New distributed FFT APIs to run at multi-node/multi-GPU systems
- New device matmul tensor APIs for writing high-performance kernels with numba-cuda

GTC 2025

Beta 3

- Support for narrow floating-point types, FP8 and MXFP8, in nvmath-python specialized matmul

Multi-GPU/-Node execution

- Extreme scalability to thousands of GPUs
- Helper functions for data partitioning
- …

Exposed advanced kernel writing capabilities

- Thanks to NVIDIA MathDx and CuTe, developers now can easily write high-performance numerical kernels in C++.
- With Beta 4 release, nvmath-python delivers these capabilities in pythonic way, so that developers can write such kernels using numba-cuda

Multi-GPU FFT API with NumPy ndarrays

```
import numpy as np
import cuda.core.experimental as ccx                    (1)
from mpi4py import MPI
import nvmath.distributed


comm = MPI.COMM_WORLD
rank, nranks = comm.Get_rank(), comm.Get_size()
device_id = rank % ccx.system.num_devices
nvmath.distributed.initialize(device_id, comm)

shape = 64, 256 // nranks, 128                          (2)

a = np.random.rand(*shape) + 1j * np.random.rand(*shape)
b = nvmath.distributed.fft.fft(a, distribution=nvmath.distributed.fft.Slab.Y)   (3)
```

Going big is easy as 1, 2, 3…

1. Add a few extra imports and initialization code
2. Provide partitioned operands
3. Use distributed FFT API and invoke script using mpiexec

Here is complete code that you can copy-n-paste and to run on thousands of GPUs!

Note: Like single-GPU APIs, distributed APIs provide both stateless (functional) and stateful (class-object) forms.

Check these examples out on nvmath-python GitHub:
https://github.com/NVIDIA/nvmath-python/tree/main/examples/distributed

Expanding linear algebra support to sparse domain

- Based on the state-of-the-art direct sparse solver, cuDSS
- Interoperates with existing sparse tensor libraries from SciPy, CuPy, PyTorch

Stateful API

```
import cupy as cp
import cupyx.scipy.sparse as sp
import nvmath

n = 8  # number of equations
a = sp.random(n, n, density=0.5, format="csr", dtype="float64")
a += sp.diags([2.0] * n, format="csr", dtype="float64")
b = cp.ones((n,))

with nvmath.sparse.advanced.DirectSolver(a, b) as solver:
    plan_info = solver.plan()
    fac_info = solver.factorize()
    x = solver.solve()
```

Stateless API

```
import cupy as cp
import cupyx.scipy.sparse as sp
import nvmath

n = 8  # number of equations
a = sp.random(n, n, density=0.5, format="csr", dtype="float64")
a += sp.diags([2.0] * n, format="csr", dtype="float64")
b = cp.ones((n,))

x = nvmath.sparse.advanced.direct_solver(a, b)
```

Demonstrating the novel Universal Sparse Tensor (UST) library

- Sparse computing performance greatly depends on data structure, format, and performed operations
- Throughout decades several popular named formats emerged, sparse tensor libraries provide support for key operations for these formats.
- However, sparse ecosystem is still very sparse and fragmented
- UST expands sparse ecosystem by offering DSL for user-defined sparse formats. It also implements core tensor operations and dispatches those to respective library calls (cuBLAS, cuDSS, cuSparse) or JIT/LTO to cover “holes”

SC24

Beta 2 and Beta 2.1

- Added CPU execution space based on FFT libraries conforming to FFTW3 API (NVPL for aarch64, MKL for x86)
- Added support for prolog and epilog callbacks for FFT, written in numba-cuda
- Warp tile programming using math library primitives

Pls to run at
systems
nsor APIs for
nce kernels

SciPy 2025

Just released!

Beta 5

- Added support for direct sparse solvers, cuDSS
- Interoperability with SciPy, CuPy, and PyTorch sparse tensors
- Demo for interoperability with Universal Sparse Tensors
- Complete migration to cuda.core

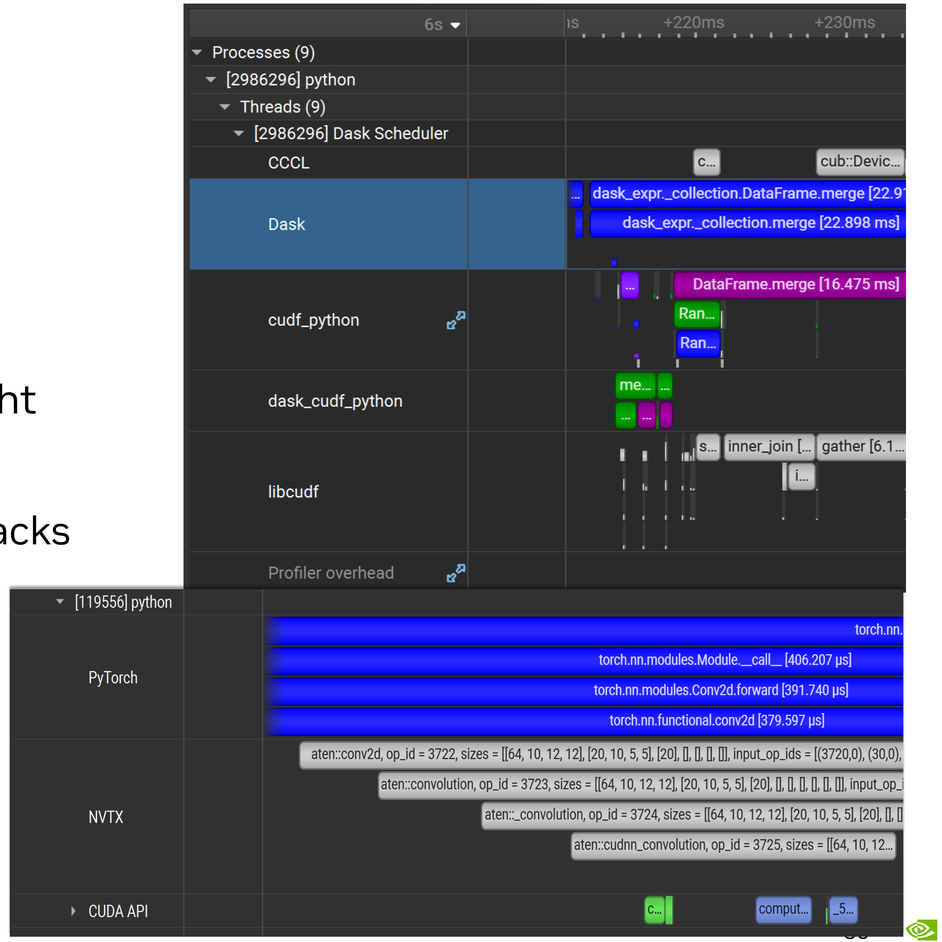
Developer Tools

Daniel Rodriguez

Enabling Python

- Developer tools tutorials with examples covering Nsight Systems, Nsight Compute, Compute Sanitizer
  - Accelerated Computing Hub > Accelerated Python User Guide > Chapter 10: Developer Tools

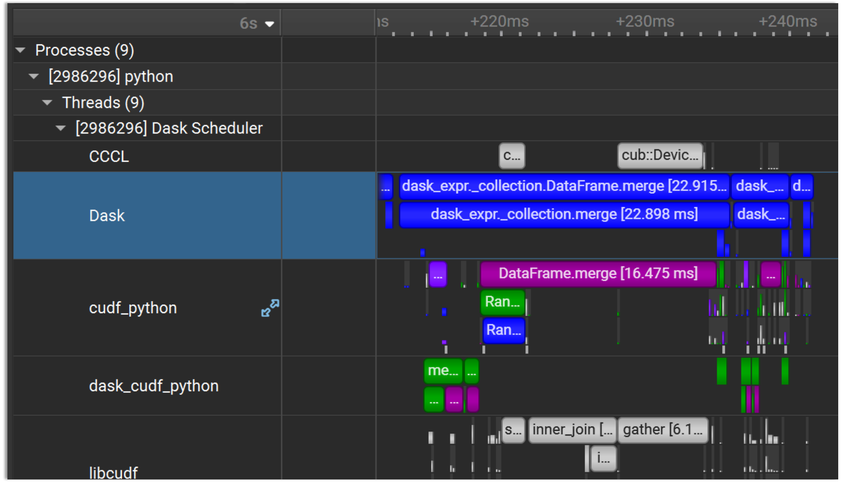
- Python Profiling
  - **Workload and function trace in Nsight Systems**
    - **PyTorch (including Layers), RAPIDS, Dask**
    - Python NVTX module to annotate .py files
    - User-defined annotation injection config files
  - Call Stacks with CUDA API backtrace in Compute Sanitizer, Nsight Systems (includes sampling), and Nsight Compute
    - Nsight Compute Source correlation with native & Python call-stacks
    - Nsight Systems Call stack Views: top-down, bottom-up, flat
- JupyterLab extension: Nsight Systems & Nsight Compute
  - Profile individual cells
  - View reports in Nsight tool GUI inside JupyterLab

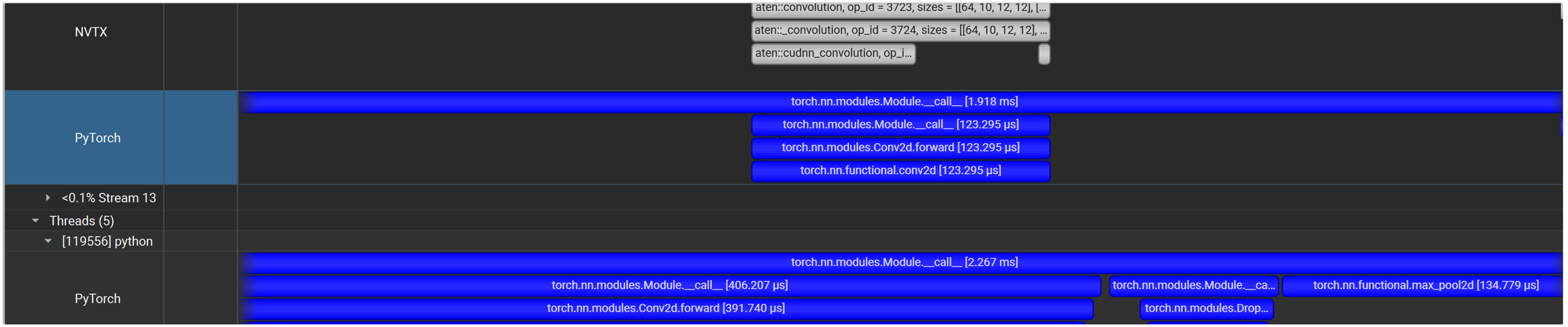


Nsight Systems

Python Profiling Updates

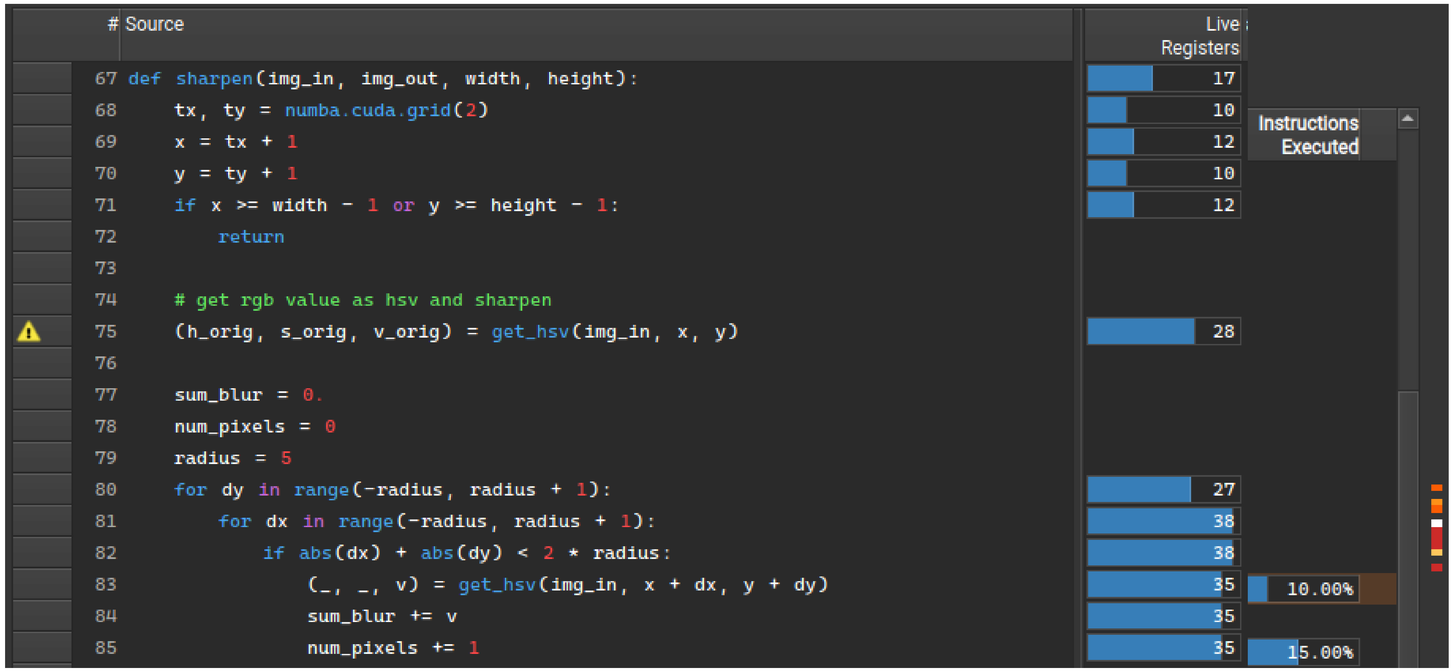
- Python Call Stacks Samples and CUDA API Backtrace
  - Top-down and Bottom-up tree views for easy analysis
- Global Interpreter Lock (GIL) trace
  - Common performance limiter in Python
- See annotated code ranges built into in popular frameworks and libraries such as:
  - PyTorch(with Layers trace), RAPIDS, DASK, Spark, CV-CUDA, and more…
  - Custom trace of Python functions without needing source code changes

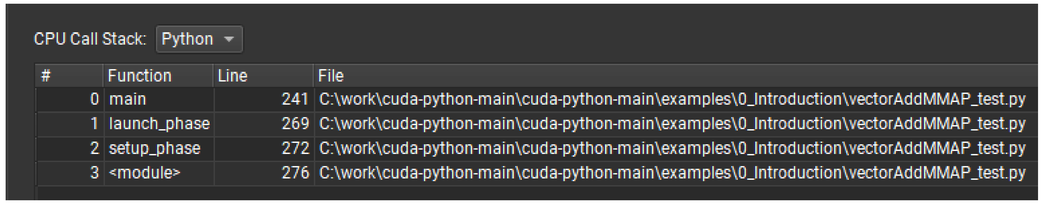


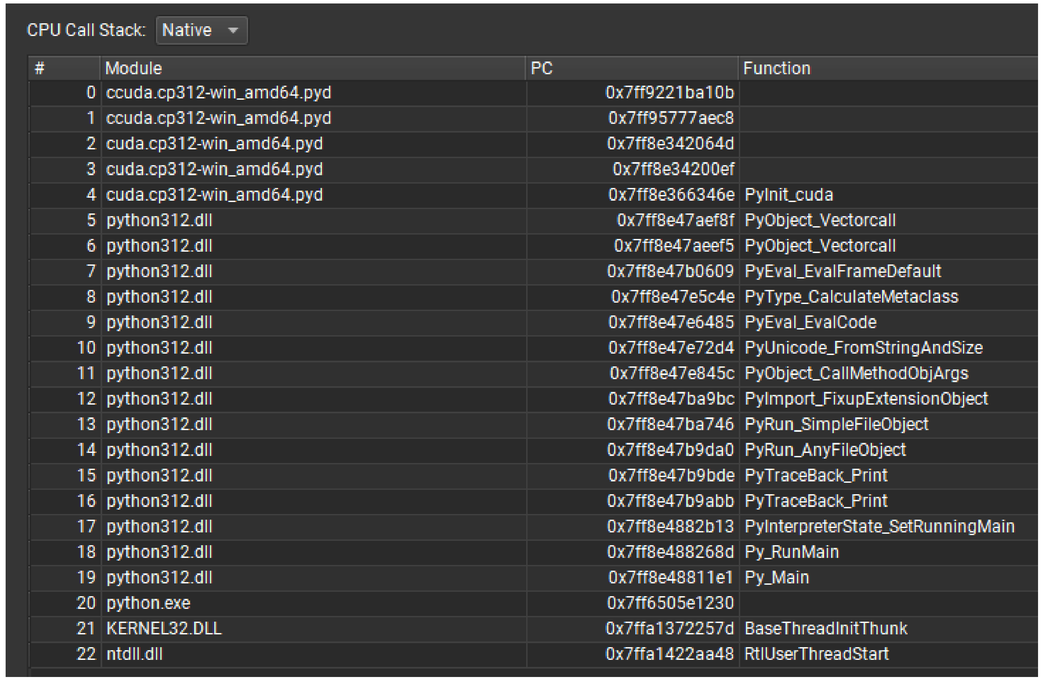


Nsight Compute

- Source resolution and syntax highlighting
- Call stacks for native and Python code





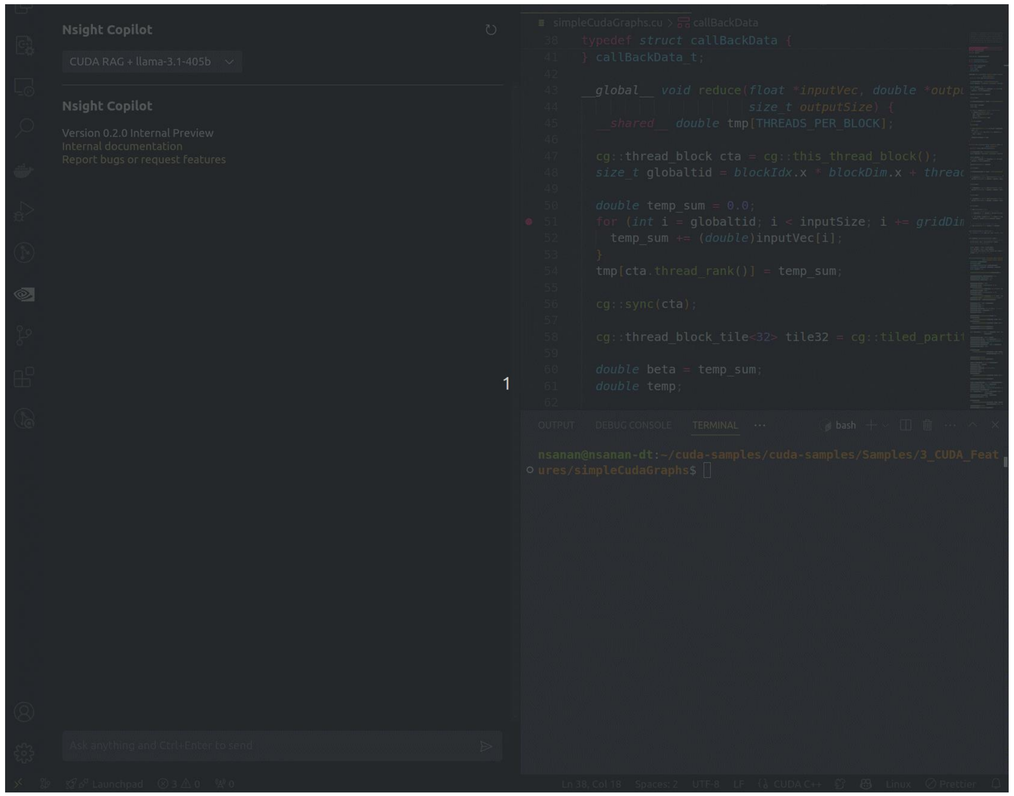


Nsight Copilot

AI Integrations in Nsight products

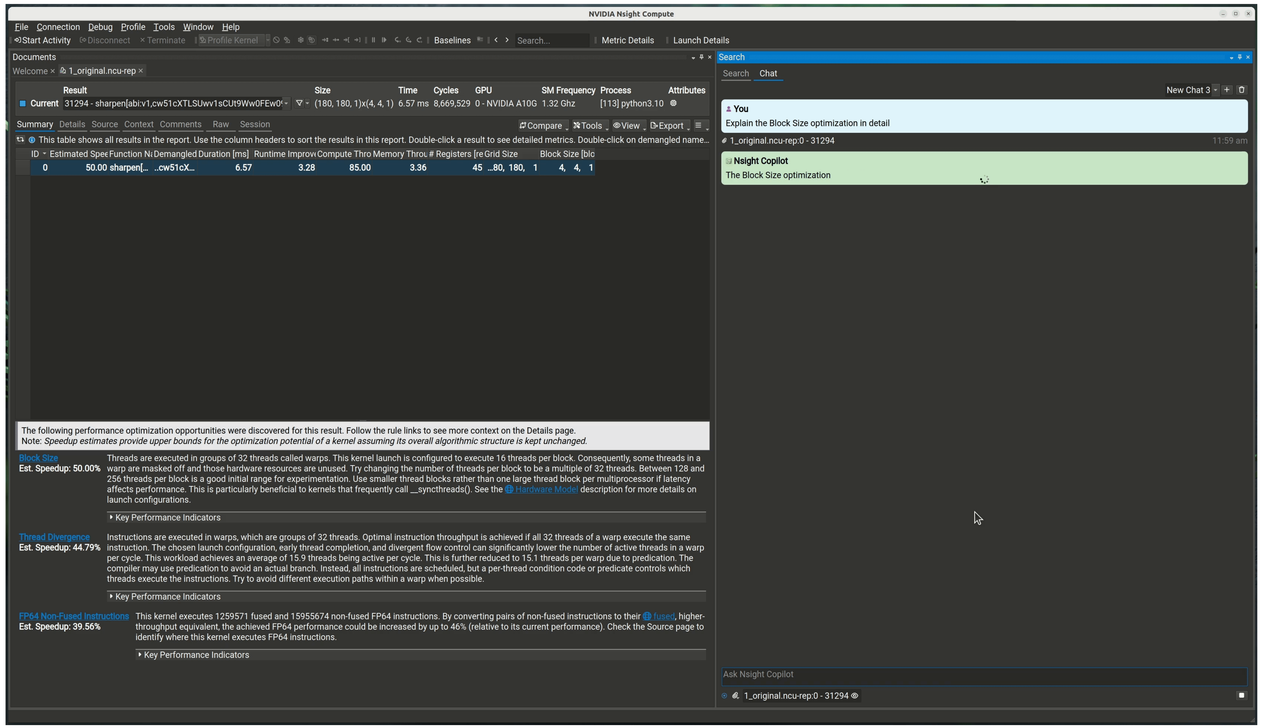
Nsight Copilot in Visual Studio Code

- Helps users write and understand CUDA code and concepts
- Autocomplete and code generation



Nsight Copilot in Nsight Compute

- Helps users with questions about Nsight Compute usage, profile result data and performance optimizations
- Interactive chat with conceptual CUDA knowledge of collected data



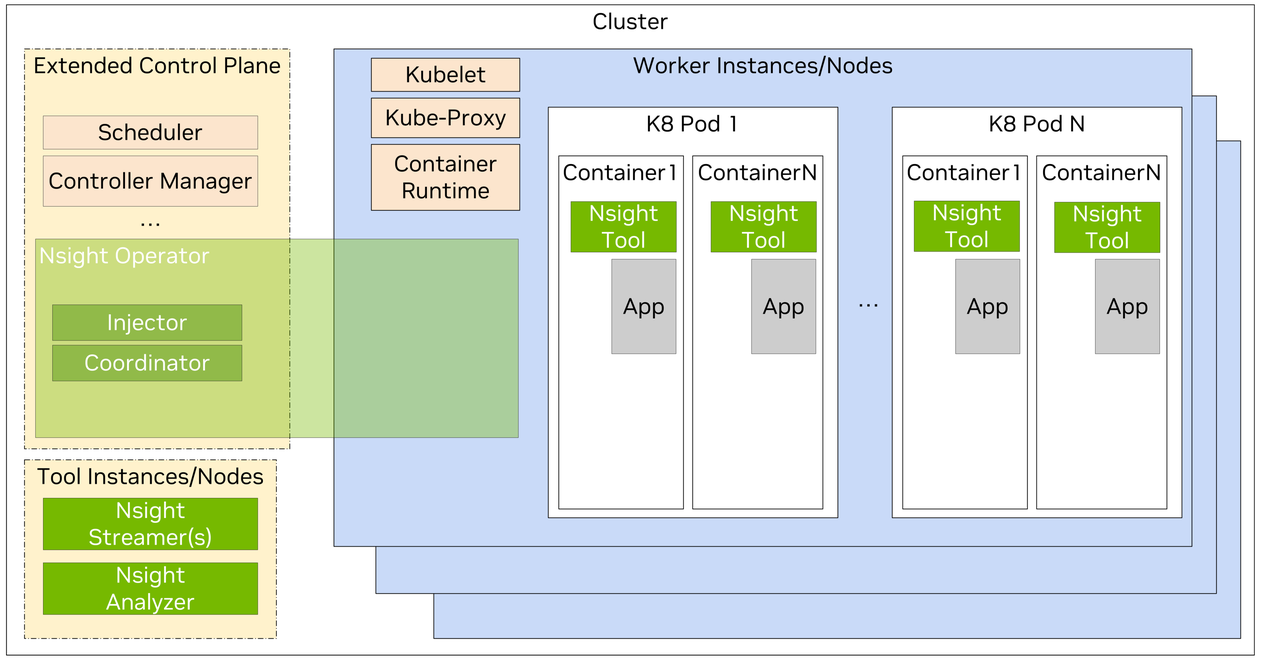
Introducing Nsight Cloud

Goals

- Improve ease of use of Nsight Tools
  - Reduce need for custom debug builds/containers
  - Simplify analysis and presentation of data collected over “n”(~1000s) nodes
- Improve scale-out for Cloud users (Public/CSP, Private, On-premises), Data Centers, Research Labs
  - Multi-node communication
  - Job schedulers / Task orchestrators

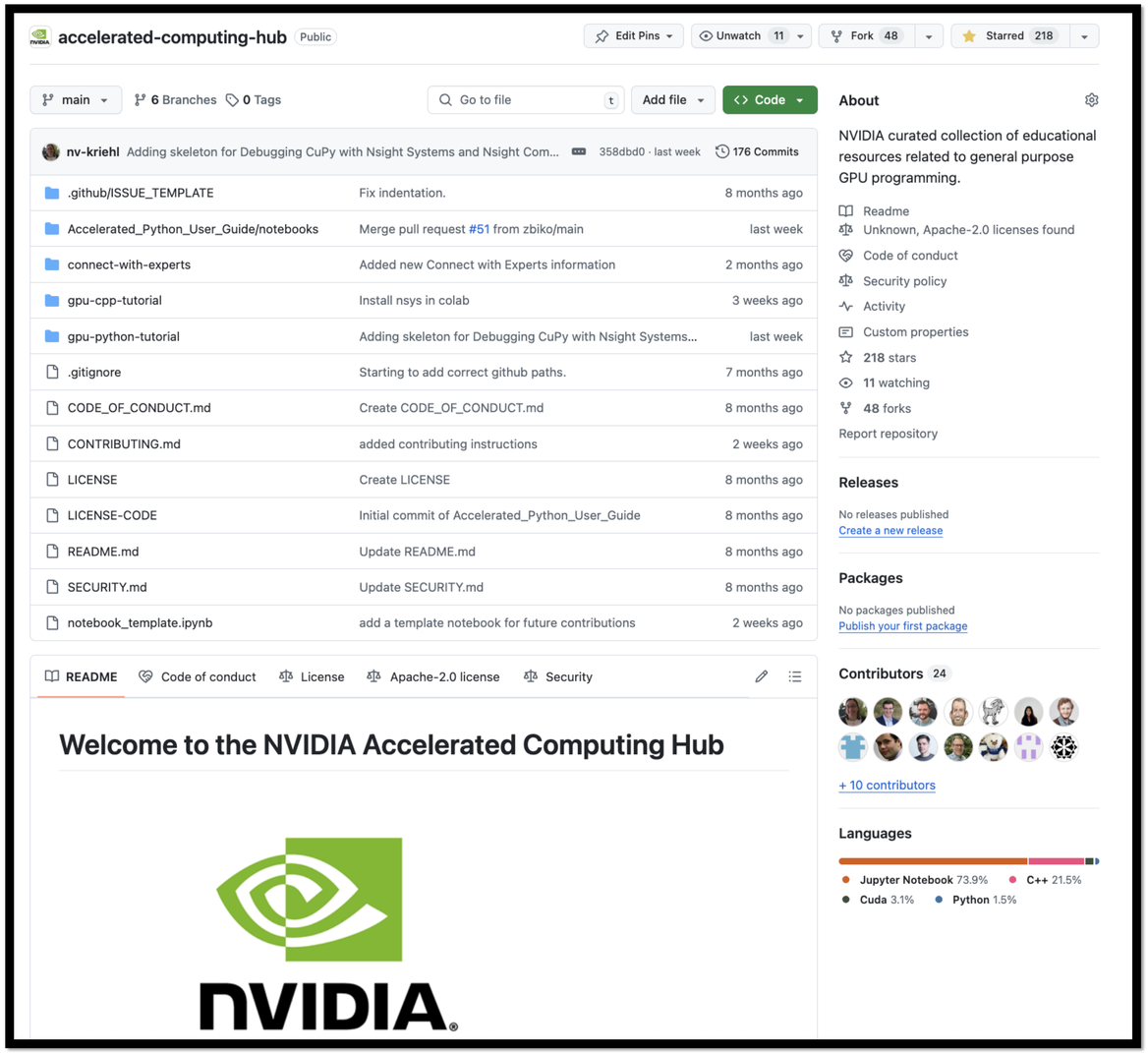
Components

- **Nsight Operator** - provision, enable and manage profile
  - **Injector** - contextual profiling agent added to task or container, helps avoid the need modify and create dev/debug instances
  - Available on ngc.nvidia.com for Nsight Systems today
- **Nsight Streamer** - view/visualize reports from Nsight tools
  - Lightweight, reduces need to move large amounts of data
  - Secure, sensitive profile and system data stays on prem
  - Available on ngc.nvidia.com for Nsight Systems, Nsight Deep Learning Designer, Nsight Compute and Nsight Graphics today



Accelerated Computing Hub

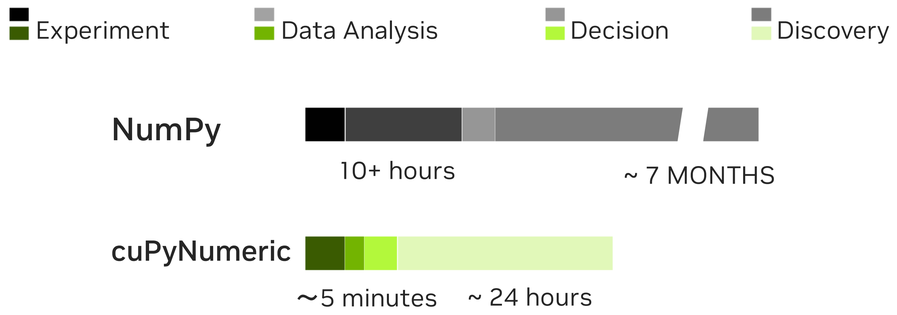
Go To: https://github.com/NVIDIA/accelerated-computing-hub



BACKUP

Backup: Early Adopters

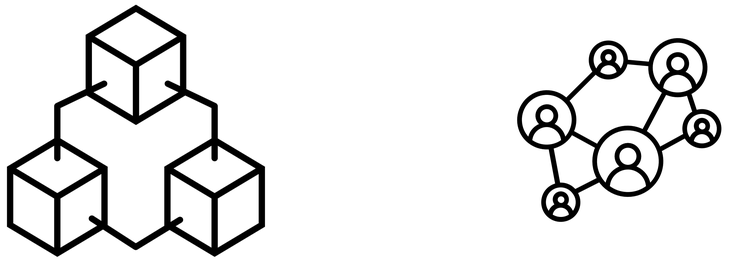
LCLS Data Discovery in Experiment



SLAC National Accelerator Laboratory

Data Analytics from 10+ hours to ~5 minutes

Money Laundering Detection by Transaction Path Tracking



National Payments Corporation of India

Computation from hours to seconds

CUPTI Updates

- Low overhead kernel tracing using Hardware Events System (HES) on Blackwell GPU
- Periodic metric sampling
  - Low-overhead GPU performance metric sampling at the device level
  - Each sample includes the metric value and a timestamp
  - Turing+
- A new set of Host and Target Profiling APIs
  - A consistent and uniform set of APIs – naming, types, error codes, error handling macros etc.
  - New APIs simplify API usage compared to current Perfworks APIs.
  - Unlocks NVLINK, C2C and PCIe metrics
- Python APIs
  - A subset of CUPTI C Activity and Callback APIs
  - Supported Python versions 3.9, 3.10, 3.11 or 3.12
- Symbol file for improved debuggability

```
void Profiler::Setup()
{
    // Enable CUPTI activities
    cuptiActivityEnable(CUPTI_ACTIVITY_KIND_DRIVER);
    cuptiActivityEnable(CUPTI_ACTIVITY_KIND_RUNTIME);
    cuptiActivityEnable(CUPTI_ACTIVITY_KIND_CONCURRENT_KERNEL);
    cuptiActivityEnable(CUPTI_ACTIVITY_KIND_MEMCPY);
    cuptiActivityEnable(CUPTI_ACTIVITY_KIND_MEMSET);

    // Register callbacks for buffer requested and completed
    cuptiActivityRegisterCallbacks(bufferRequested, bufferCompleted);
}

void Profiler::Flush()
{
    // Flush activity buffers
    cuptiActivityFlushAll();
}
```