ACES: Introduction to CuPy: NumPy & SciPy for GPU

Practical GPU Computing with CuPy

Jian Tao

College of Performance Visualization & Fine Arts,
Texas A&M Institute of Data Science,
Texas A&M University
College Station, TX

Nov 11, 2025

TEXAS A&M
UNIVERSITY

Course Agenda

Part 1
CuPy Overview and Philosophy

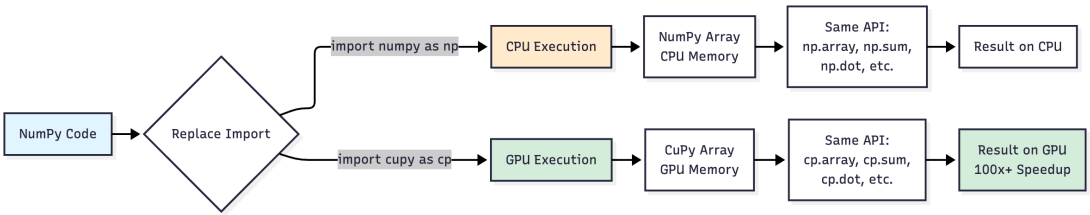
What is CuPy?

CuPy is a NumPy/SciPy-compatible GPU array library accelerating computations via **CUDA** and **ROCm** backends, offering **drop-in replacement APIs** for many array operations. It provides ndarray, sparse matrices, and routines with the same API surface as NumPy and SciPy.

Key goals:

- Complete NumPy/SciPy API coverage for drop-in migration
- Leverage CUDA libraries (cuBLAS, cuFFT, cuSOLVER) for performance
- Enable advanced CUDA features without deep GPU expertise

Core Design Principle



Drop-in paradigm: Replace `numpy` with `cupy` and `scipy` with `cupyx.scipy` to shift compute to GPU with minimal code change when APIs are compatible.

This enables rapid prototyping and gradual migration of existing codebases, maintaining familiar syntax while exploiting GPU parallelism for large-scale data processing.

CuPy

Hello, CuPy – First Program

This mirrors NumPy's syntax but transparently allocates GPU memory and dispatches CUDA kernels for operations.

```
import cupy as cp

# Create array on GPU
x = cp.arange(6).reshape(2, 3).astype('f')
# [[0. 1. 2.]
#  [3. 4. 5.]]

# Reduction along axis 1
y = x.sum(axis=1)
print(y)   # array([ 3., 12.], dtype=float32)
```

CuPy

Part 2
Running CuPy on ACES

ACES Overview

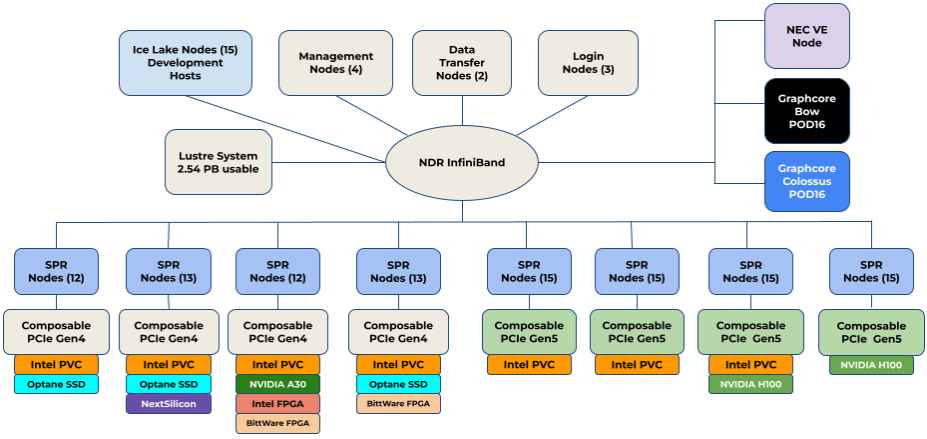
ACES: Accelerating Computing for Emerging Sciences, a **composable** computational prototype developed by Texas A&M University.

ACES
ACCELERATING COMPUTING FOR EMERGING SCIENCES

CuPy

ACES Configuration

Enables mixed-resource workflows optimized for your specific computational needs.



ACES System Specifications - Compute

Component	Quantity	Description
Sapphire Rapids Nodes	110 nodes	96 cores, dual Xeon 8468 512 GB DDR5 memory NDR 200 Gbps InfiniBand
Ice Lake Nodes	2 nodes	64 cores, dual Xeon 8352Y 512 GB DDR4 memory
PCIe Gen4 Infrastructure	50 nodes	Up to 20 PCIe cards/node
PCIe Gen5 Infrastructure	60 nodes	Up to 16 H100s or 14 PVCs/node
Lustre Storage	2.5 PB	HDR IB connected

ACES Accelerators

Accelerator	Quantity	Purpose
NVIDIA H100	30	HPC, DL Training, AI
NVIDIA A30	4	AI Inference, Compute
Intel PVC GPUs	120	HPC, DL Training, AI
Graphcore IPUs	32	16 Colossus + 16 Bow
Intel PAC D5005 FPGA	2	Stratix 10 GX
BittWare IA-840F FPGA	3	Agilex AGF027
NextSilicon Coprocessor	2	Reconfigurable accelerator
NEC Vector Engine	8	Vector computing
Intel Optane SSD	48	18 TB memory-addressable

Accessing ACES

ACES Portal

URL: portal-aces.hprc.tamu.edu

- Web-based interface powered by Open OnDemand 3.0.0
- Single access point for all HPC resources
- Shell access directly in your browser

ACCESS Users

Available via ACCESS (formerly XSEDE)

- Login with ACCESS credentials
- Select appropriate Identity Provider
- Consent to attribute release

Running CuPy on ACES

Step 1: Login to ACES

- **Via Portal: URL:** portal-aces.hprc.tamu.edu

Step 2: Load Software Environment

```
module purge
module load GCC/12.3.0 OpenMPI/4.1.5 CuPy/13.0.0-CUDA-12.1.1
```

Step 3: Run Sample CuPy Script on GPU Node

```
cp /scratch/training/cupy/CUPY_Examples.tgz $SCRATCH; cd $SCRATCH
tar -zxvf CUPY_Examples.tgz; cd CUPY_Examples
python cupy_linalg_example.py
```

Note: Use **module spider CuPy** to see all available versions of CuPy on ACES.

Using GPUs with Slurm

Sample job script **cupyjob.sh** for running a CuPy script on one H100 GPU.

```
#!/bin/bash
#SBATCH --job-name=cupy_job
#SBATCH --time=00:01:00
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=96
#SBATCH --mem=488G
#SBATCH --partition=gpu
#SBATCH --gres=gpu:h100:1
#SBATCH --output=stdout.%x.%j

module purge
module load GCC/12.3.0 OpenMPI/4.1.5 CuPy/13.0.0-CUDA-12.1.1
python my_cupy_script.py
```

Note: Use **sbatch cupyjob.sh** to submit job to ACES.

Accelerator Access Summary

Accelerator	**Slurm Partition/Access**
NVIDIA H100/A30	`--partition=gpu` `--gres=gpu:h100:N`
Intel PVC GPUs	`--partition=pvc`
BittWare FPGA	`--partition=bittware`
Intel Optane SSD	`--partition=memverge`
NextSilicon	`--partition=nextsilicon`
NEC Vector Engine	`--partition=nec`
Graphcore IPUs	Interactive: `ssh poplar1/poplar2`

Resources and Support

Documentation

- ACES User Guide: `https://hprc.tamu.edu/kb/User-Guides/ACES`
- General HPRC: `https://hprc.tamu.edu`
- YouTube Channel: `https://www.youtube.com/texasamhprc`

Getting Help

- Email: `help@hprc.tamu.edu` (preferred)
- Phone: (979) 845-0219
- Policies: `https://hprc.tamu.edu/policies/`

Part 3
Core CuPy Operations and Computing

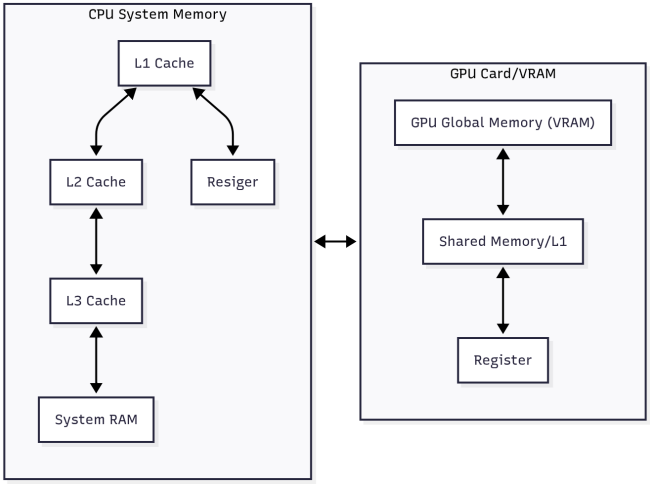
Creating Arrays on GPU

cupy.array mirrors NumPy's constructor, supporting dtype, order (C/Fortran), and copy flags. CuPy keeps array data in memory along with **shape**, **dtype**, and **strides** attributes. Arrays allocate device (GPU) memory and support asynchronous host-to-device (H2D) transfers when used with streams.

Options:

- cp.array(host_data) – copy from host NumPy array
- cp.zeros, cp.ones, cp.empty – initialize directly on GPU
- cp.asarray – zero-copy view if already CuPy array

CPU & GPU Memory



- Memory stores data and programs for fast access.
- **CPU:** Deep cache, flexible random access.
- **GPU:** Shallow cache, high bandwidth, optimized for parallel patterns.
- GPU parallelism increases throughput for big data tasks.

CuPy

CPU/GPU Memory Performance

Memory Type	CPU Latency	CPU Bandwidth	GPU Latency	GPU Bandwidth
Register	∼1 cycle	∼500–1000 GB/s	∼1 cycle	∼10 TB/s
L1 Cache	4–5 cycles	∼1 TB/s	20–30 cycles	∼3–4 TB/s
L2 Cache	12–15 cycles	∼500 GB/s	100–200 cycles	∼2 TB/s
L3 / Shared L2	35–70 cycles	∼200 GB/s	200–300 cycles	∼1 TB/s
Main/Global Memory	80–120 ns	∼100 GB/s	400–800 ns	∼800 GB/s

Table: Typical latency and bandwidth in CPU and GPU memory hierarchies as of Nov 2025. Values are approximate and hardware-dependent.

- GPU (device) and CPU (host) transfer data over PCIe, which is much slower than onboard memory.
- Minimizing CPU↔GPU transfers is critical for performance.
- Techniques like pinned (page-locked) memory can speed up transfers.

cupy.array Basics and Options

```python
import numpy as np
import cupy as cp

# Host NumPy array
a = np.arange(8, dtype=np.int32)

# H2D copy to GPU (blocking by default)
x = cp.array(a, copy=True)

# Direct GPU allocation
y = cp.zeros((2, 4), dtype=cp.float32, order='C')
z = cp.ones_like(y, dtype=cp.float64)
```

H2D copies are synchronous unless wrapped in a stream context; manage stream ordering to prevent data races.

Dtypes, Byte Order, and Strides

CuPy supports all NumPy dtypes (int8–64, float16/32/64, complex64/128, structured). When copying big-endian (rare) NumPy arrays, bytes are swapped to little-endian (common) for GPU compatibility.

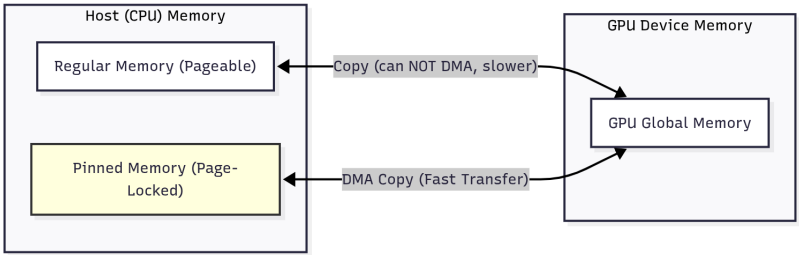
```
y = cp.zeros((2, 4), dtype=cp.float32, order='C')

print(y.strides)  # (16, 4) for row-major float32
```

Memory order:

- `order='C'` – row-major (default)
- `order='F'` – column-major (matches Fortran/MATLAB)

Stride-aware indexing and broadcasting work identically to NumPy; ensure layout matches downstream kernel expectations for peak memory bandwidth.

What is Pinned Memory?



- **Pinned memory** (also called page-locked memory) is host (CPU) memory allocated so that the operating system cannot swap it out to disk.
- Special hardware (like a GPU) can access it directly for high-speed transfers without CPU intervention.

CuPy

Pinned Memory for Faster Transfers

Pinned (page-locked) memory improves H2D/D2H throughput and enables overlap with compute via streams.

```
# Create a pinned memory pool
pinned_pool = cp.cuda.PinnedMemoryPool()
cp.cuda.set_pinned_memory_allocator(pinned_pool.malloc)

# Allocate a pinned memory buffer on the host (CPU)
pinned_mem = cp.cuda.alloc_pinned_memory(400)  # 400 bytes

# Use numpy.frombuffer to create a NumPy array backed by the pinned memory
arr_cpu = np.frombuffer(pinned_mem, dtype=np.float32, count=100)
arr_cpu[:] = np.arange(100, dtype=np.float32)   # Fill with data

# Copy to GPU (using the fast path with pinned memory)
arr_gpu = cp.asarray(arr_cpu)
```

CuPy

Drop-in Replacement Paradigm

NumPy Function	Supported in CuPy?
`fromfunction`	Partially
`vectorize`	No
`piecewise`	No
`matrix`	No
`matmul` (object dtype)	No
`datetime64`, `timedelta64`	No
`fft` (irregular shapes)	Partially
`polyfit`, `polyval`, `histogram2d`	No
`einsum` (complex modes)	Partially
`char` (string ops)	No
`ufuncs` (on lists or NumPy arrays)	No

Table: Examples of NumPy functions not supported or only partially supported by CuPy.

Most NumPy operations work identically except a few (see Table to the left for some examples).

Automatic dispatch: CuPy leverages cuBLAS for GEMM, cuFFT for FFT, and cuSOLVER for decompositions, providing performance on par with native CUDA code for standard operations.

Key Behavioral Differences

Intentional divergences from NumPy:

- Out-of-bounds integer indexing may **wrap** instead of raising `IndexError`. Some CuPy functions (like cp.put) let you specify mode='raise' for NumPy-like error handling if needed.
- Certain dtype promotion rules differ subtly
- Unsupported: some advanced NumPy features (e.g., `np.vectorize`, certain string ops)

Always test ported code for correctness; consult the differences guide for edge cases before migration; review CuPy's user guide if your application uses advanced NumPy features.

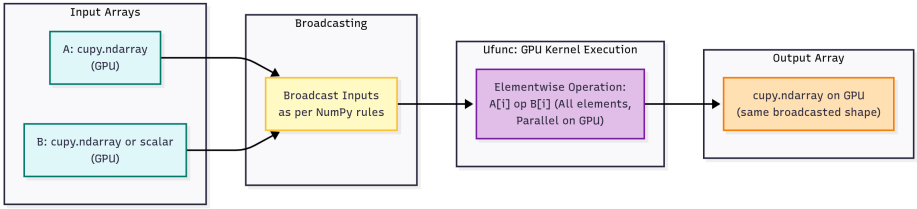
Migrating Existing NumPy Code

```python
# Before: NumPy on CPU
import numpy as np
x = np.random.rand(1000, 1000)
y = (x - x.mean(axis=0)) / x.std(axis=0)
u, s, vt = np.linalg.svd(y, full_matrices=False)

# After: CuPy on GPU (identical syntax)
import cupy as cp
x = cp.random.rand(1000, 1000)
y = (x - x.mean(axis=0)) / x.std(axis=0)
u, s, vt = cp.linalg.svd(y, full_matrices=False)
# Result stays on GPU; use .get() only when needed
```

Avoid round-tripping arrays to host in hot loops; keep pipeline GPU-resident for maximum speedup.

Ufunc Support in CuPy

CuPy implements NumPy's universal functions (ufuncs) with full type signature support, out parameters, and methods like `accumulate`, `reduce`, `outer`.



Ufunc methods:

- `cp.add.reduce(x)` – sum all elements
- `cp.add.accumulate(x)` – prefix sum (scan)
- `cp.multiply.outer(x, y)` – Cartesian product

Each method dispatches optimized GPU kernels, exploiting parallelism for large arrays.

Using Built-in Ufuncs

```python
import cupy as cp

x = cp.linspace(-3, 3, 7, dtype=cp.float32)

# Elementwise unary ufunc
y = cp.tanh(x)

# Reduction via ufunc method
total = cp.add.reduce(x)

# Prefix sum (inclusive scan)
cumulative = cp.add.accumulate(x)
print(cumulative)
# array([-3., -5., -6., -6., -5., -3.,  0.], dtype=float32)
```

Ufuncs execute on GPU and support broadcasting; combine with `out=` to reuse buffers and reduce allocations.

CuPy

cupy.ufunc Class and Metadata

The `cupy.ufunc` object exposes attributes:

- `name` – kernel function name
- `nin`, `nout` – number of input/output arrays
- `types` – list of supported dtype signatures

Use introspection to verify supported dtypes before dispatching custom ufuncs or when debugging type mismatches.

Custom Elementwise Ufunc with ElementwiseKernel

```
import cupy as cp

# Define custom ufunc: y = x^2
square = cp.ElementwiseKernel(
    'float32 x',        # input
    'float32 y',        # output
    'y = x * x;',       # operation (C syntax)
    'square_f32'        # kernel name
)

data = cp.arange(8, dtype=cp.float32)
result = square(data)
print(result)
# array([  0.,   1.,   4.,   9.,  16.,  25.,  36.,  49.], dtype=float32)
```

ElementwiseKernel JIT-compiles CUDA code at first call; subsequent invocations reuse cached kernel.

CuPy

Reductions and Scans at Scale

```
x = cp.arange(10_000_000, dtype=cp.float32)

# Global reductions (single-pass where possible)
s = x.sum()
mn = x.min()
mx = x.max()

# Prefix sum (inclusive scan via Thrust)
prefix = cp.cumsum(x)

# Argmax/argmin for index of extrema
idx = cp.argmax(x)
```

Large reductions exploit GPU parallelism; CuPy dispatches optimized kernels from Thrust/CUB libraries for peak throughput.

Part 4

Scientific Computing with GPU-Accelerated Libraries

GPU Random Number Generation

```python
import cupy as cp

# Modern API (NumPy 1.17+ style)
rng = cp.random.default_rng(seed=123)
x = rng.normal(loc=0, scale=1, size=(1_000_000,))

# Compute statistics on GPU
mean = x.mean()
std = x.std()

# Legacy API still supported
y = cp.random.rand(1000, 1000)
```

Random generation uses cuRAND; all operations stay on device. Avoid `.get()` until final results needed on host.

Linear Algebra Dispatch

CuPy's `linalg` module leverages:

- **cuBLAS** – matrix multiplication (GEMM), vector operations (BLAS Level 1/2/3)
- **cuSOLVER** – LU, QR, SVD, Cholesky decompositions
- **Tensor cores** – automatic use of low-precision acceleration (FP16/TF32) on modern GPUs

These libraries deliver near-peak FLOPS for standard dense algebra workloads.

Matrix Operations and Decompositions

```python
import cupy as cp

A = cp.random.rand(2048, 2048, dtype=cp.float32)
B = cp.random.rand(2048, 2048, dtype=cp.float32)

# Matrix multiplication via @ operator (calls cuBLAS GEMM)
C = A @ B

# Singular value decomposition (thin SVD)
u, s, vt = cp.linalg.svd(A, full_matrices=False)

# Eigendecomposition, matrix inversion, etc.
eigvals = cp.linalg.eigvalsh(A @ A.T)  # symmetric
```

Use `float32` or mixed precision where acceptable to maximize throughput on tensor cores.

Batched and Vectorized Operations

```python
import cupy as cp
# Batch of small matrices (100 matrices, 64x64 each)
X = cp.random.rand(100, 64, 64, dtype=cp.float32)

# Batched inversion (loops internally if no batch API)
X_inv = cp.linalg.inv(X)

# For large batch counts, consider manual loop with streams
from cupy.cuda import Stream
s = Stream()
with s:
    for i in range(100):
        X_inv[i] = cp.linalg.inv(X[i])
s.synchronize()
```

Batch operations may loop under the hood; use streams to overlap compute when beneficial.

Sparse Matrix Support

`cupyx.scipy.sparse` provides GPU-accelerated sparse formats:

- CSR (Compressed Sparse Row) – optimized for row slicing, mat-vec
- COO (Coordinate) – flexible for construction
- CSC (Compressed Sparse Column) – optimized for column slicing

Sparse operations dispatch to cuSPARSE library; efficient for large, sparse linear systems and graph algorithms.

CSR (Compressed Sparse Row)

Original matrix A:

$$A = \begin{pmatrix} 0 & 0 & 3 \\ 4 & 0 & 0 \\ 0 & 5 & 6 \end{pmatrix}$$

CSR is best for fast row access and mat-vec operations.

- Optimized for row slicing and fast matrix-vector multiplication.
- **Data**: [3, 4, 5, 6]
- **Indices (column)**: [2, 0, 1, 2]
- **Indptr (row pointers)**: [0, 1, 2, 4]

COO (Coordinate Format)

Original matrix A:

$$A = \begin{pmatrix} 0 & 0 & 3 \\ 4 & 0 & 0 \\ 0 & 5 & 6 \end{pmatrix}$$

COO is best for easy construction.

- Flexible for construction and easy for incremental updates.
- **Row indices**: [0, 1, 2, 2]
- **Column indices**: [2, 0, 1, 2]
- **Data**: [3, 4, 5, 6]

CSC (Compressed Sparse Column)

Original matrix A:

$$A = \begin{pmatrix} 0 & 0 & 3 \\ 4 & 0 & 0 \\ 0 & 5 & 6 \end{pmatrix}$$

CSC is best for fast column access.

- Optimized for column slicing and efficient column operations.
- **Data**: [4, 5, 3, 6]
- **Row indices**: [1, 2, 0, 2]
- **Colptr (column pointers)**: [0, 1, 3, 4]

Sparse Matrix-Vector Multiply

```python
from cupyx.scipy.sparse import csr_matrix
import cupy as cp

# Construct CSR matrix (3×3, 5 non-zeros)
A_dense = cp.array([[0, 0, 3], [4, 0, 0], [0, 5, 6]], dtype=cp.float32)
A = csr_matrix(A_dense)

# Dense vector
b = cp.arange(3, dtype=cp.float32)

# Sparse mat-vec product (calls cuSPARSE)
x = A @ b
print(x)  # array([6., 0., 17.], dtype=float32)
```

Sparse mat-vec runs on GPU; scales to millions of rows/cols with low density.

Signal Processing on GPU

`cupyx.scipy.signal` accelerates:

- Convolution (direct and FFT-based)
- Filtering (FIR, IIR design and application)
- Spectral analysis (periodograms, spectrograms)

Useful for image processing, audio DSP, and time-series analysis pipelines requiring high throughput.

FFT Acceleration via cuFFT

CuPy's `cp.fft` module mirrors `numpy.fft` and dispatches to cuFFT for 1D/2D/nD transforms:

- `cp.fft.fft, rfft, ifft` – complex and real FFTs
- `cp.fft.fft2, fftn` – multidimensional transforms
- `cp.fft.fftshift, ifftshift` – frequency-domain reordering

Large FFTs exploit GPU memory bandwidth; plan caching reduces overhead on repeated calls.

CuPy FFT Example

Compute 1D FFT on GPU with CuPy.

```
import cupy as cp

# Create a 1D array on the GPU
a = cp.array([0, 1, 0, 0], dtype=cp.float32)

# Compute its FFT
fft_a = cp.fft.fft(a)

# Print the result (still on GPU)
print(fft_a)

# Optionally, bring to CPU for display:
import numpy as np
print(cp.asnumpy(fft_a))
```

CuPy

cupyx.scipy – SciPy on GPU

`cupyx.scipy` brings a subset of SciPy functionality to GPU:

- `cupyx.scipy.signal` – convolution, filtering, spectral analysis
- `cupyx.scipy.sparse` – sparse matrix formats (CSR, COO, CSC)
- `cupyx.scipy.linalg` – extended linear algebra beyond NumPy
- `cupyx.scipy.fft` – FFT wrappers (overlaps with `cp.fft`)

Coverage is partial; check documentation for supported functions before porting SciPy-heavy codebases.

CuPy Signal Processing Example

Median filtering with CuPy.

```
import cupy as cp
from cupyx.scipy.signal import medfilt

# Create a 1D noisy signal on the GPU
x = cp.linspace(0, 10, 100)
signal = cp.sin(x) + 0.5 * cp.random.randn(100)  # Noisy sinusoid

# Apply median filter
filtered = medfilt(signal, kernel_size=5)

print(filtered)
```

GPU 2D Convolution Example

```python
import cupy as cp
from cupyx.scipy.signal import convolve2d

# Large image (4K×4K)
img = cp.random.rand(4096, 4096).astype(cp.float32)

# Gaussian-like kernel (31×31 Hanning window)
kernel = cp.outer(cp.hanning(31), cp.hanning(31))
kernel /= kernel.sum()  # normalize

# 2D convolution on GPU
out = convolve2d(img, kernel, mode='same', boundary='wrap')

print(out.shape, out.dtype)
# (4096, 4096) float32
```

Convolution mirrors SciPy's API but runs on GPU, leveraging parallel threads for spatial operations.

CuPy

Part 5

Advanced GPU Programming and Optimization

NCCL: NVIDIA Collective Communication Library

What is NCCL?

- **NCCL**: NVIDIA Collective Communication Library
- Provides efficient multi-GPU communication for operations like AllReduce, Broadcast, Gather, Scatter
- Optimized for CUDA GPUs and used for large-scale deep learning
- CuPy uses NCCL for distributed array operations on multiple GPUs

How Does CuPy Use NCCL?

- CuPy supports NCCL via the `cupy.cuda.nccl` module
- Enables collective operations (sum, broadcast, etc.) between arrays on different GPUs
- Simple Pythonic API for multi-GPU communication and synchronization

NCCL in CuPy

Key NCCL Operations in CuPy

- **AllReduce**: Sum (or other op) across all participating GPUs, result is distributed back
- **Broadcast**: Copy data from one GPU to all others
- **Reduce**: Aggregate data onto a single GPU
- **AllGather**: Concatenate data from all GPUs and distribute the result

When to Use NCCL with CuPy?

- Multi-GPU deep learning for model/data parallelism
- Large-scale data processing
- Scientific computing that requires fast GPU-to-GPU communication

Example: NCCL AllReduce in CuPy

```
import cupy as cp
from cupy.cuda import nccl

# Assume at least two GPUs are available
size = 2
rank = 0  # This would be different for each process
comm_id = nccl.get_unique_id()
comm = nccl.NcclCommunicator(size, comm_id, rank)

cp.cuda.Device(0).use()
data = cp.array([1, 2, 3], dtype=cp.float32)
comm.allReduce(data.data.ptr, data.data.ptr, data.size, nccl.NCCL_FLOAT,
               nccl.NCCL_SUM, cp.cuda.Stream.null.ptr)
print(cp.asnumpy(data))
```

CuPy

Memory Pools for Efficient Allocation

```python
import cupy as cp
mp = cp.get_default_memory_pool()
pp = cp.get_default_pinned_memory_pool()

x = cp.empty((1024, 1024), dtype=cp.float32)

# Inspect pool usage
print(f"Used: {mp.used_bytes() / 1e6:.2f} MB")
print(f"Total allocated: {mp.total_bytes() / 1e6:.2f} MB")

# Free unused blocks back to OS
mp.free_all_blocks()
```

Memory pools cache allocations to amortize cudaMalloc/cudaFree overhead; monitor usage with `nvidia-smi` to avoid OOM.

Streams for Concurrent Kernel Execution

```python
import cupy as cp
s1 = cp.cuda.Stream(non_blocking=True)
s2 = cp.cuda.Stream(non_blocking=True)

with s1:
    x = cp.random.rand(1_000_000, dtype=cp.float32)
    y = cp.sin(x)

with s2:
    a = cp.random.rand(1_000_000, dtype=cp.float32)
    b = cp.cos(a)

s1.synchronize()
s2.synchronize()
```

Non-blocking streams enable concurrent kernel launches and overlapping H2D/D2H transfers when hardware permits.

Events for Device-side Timing

```python
import cupy as cp
start = cp.cuda.Event()
end = cp.cuda.Event()

start.record()
# Kernel or computation here
z = x @ y.reshape(-1, 1)
end.record()
end.synchronize()

elapsed_ms = cp.cuda.get_elapsed_time(start, end)
print(f"Kernel took {elapsed_ms:.3f} ms")
```

CUDA events measure GPU execution time accurately, avoiding host-side scheduling noise.

Transfer Patterns and Data Residency

Best practices:

- Keep data GPU-resident across entire pipeline
- Batch H2D/D2H transfers; avoid per-iteration copies
- Use pinned memory for large transfers
- Overlap transfers with compute via streams

Minimizing PCIe traffic is critical; profile with Nsight Systems to identify bottlenecks.

Device Enumeration and Context

```python
import cupy as cp

# Query device count
n_devices = cp.cuda.runtime.getDeviceCount()
print(f"Available GPUs: {n_devices}")

# Get current device object
dev = cp.cuda.Device()
print(dev.id, dev.compute_capability)

# Inspect device attributes
attrs = dev.attributes
print(f"Max threads/block: {attrs['MaxThreadsPerBlock']}")
print(f"Multiprocessors: {attrs['MultiProcessorCount']}")
```

Use `Device` and runtime APIs to inspect hardware capabilities and guide kernel launch configurations.

When GPUs Accelerate Workloads

Ideal use cases:

- Large arrays ($> 10^6$ elements)
- Compute-intensive kernels (high arithmetic intensity)
- Parallelizable operations (minimal data dependencies)

Avoid:

- Small arrays with frequent H2D/D2H copies
- Tight Python loops calling single-element ops
- Algorithms with heavy branching or irregular memory access

End-to-End Timing with Synchronization

```python
import time
import cupy as cp

# Ensure all prior kernels complete
cp.cuda.Stream.null.synchronize()

t0 = time.perf_counter()

# Compute workload
x = cp.random.rand(4096, 4096, dtype=cp.float32)
y = cp.linalg.svd(x, full_matrices=False)

# Wait for GPU to finish
cp.cuda.Stream.null.synchronize()

elapsed = time.perf_counter() - t0
print(f"Elapsed: {elapsed:.3f} s")
```

Always synchronize before/after timing; kernel launches are asynchronous by default.

cupyx.profiler.benchmark Utility

```python
from cupyx.profiler import benchmark

def my_func(a, b):
    return 3 * cp.sin(-a) * b

a = 0.5 - cp.random.rand(10000)
b = cp.random.rand(10000)

# Run n_repeat times; report CPU/GPU time
result = benchmark(my_func, (a, b), n_repeat=1000, n_warmup=10)

print(result)
```

benchmark automates timing with warm-up, synchronization, and statistics collection.

CuPy

Kernel Fusion and Launch Overhead

Problem: Small kernels suffer from launch latency ($\sim$5–10 μs per launch).

Solutions:

- Fuse multiple elementwise ops into single `ElementwiseKernel`
- Batch inputs to process multiple arrays per launch
- Use vectorized CuPy expressions that auto-fuse where supported

Fusion reduces memory traffic and kernel count; critical for bandwidth-bound workloads.

Bandwidth vs Compute Bound Analysis

Bandwidth-bound: Memory transfer time dominates compute.

- Optimize: reduce data movement, fuse kernels, increase arithmetic intensity

Compute-bound: ALU utilization limits throughput.

- Optimize: use lower precision (FP16/TF32), exploit tensor cores

Profile with Nsight Compute to measure achieved bandwidth and FLOPS vs theoretical peak.

Streams and Concurrency for Overlap

Multiple streams enable:

- Concurrent kernel execution (independent data)
- Overlapping H2D transfers with compute
- Overlapping D2H copies with next iteration's compute

Requires careful synchronization (events/barriers) to avoid data races; test with CUDA memcheck.

Part 6

Case Study: 2D Convolution Pipeline

Convolution Case Study Overview

Goal: Accelerate 2D image convolution on 4K×4K images using GPU.

Steps:

1. Allocate data directly on GPU
2. Apply convolution via `cupyx.scipy.signal`
3. Measure timing with CUDA events
4. Compare to CPU SciPy baseline for validation

This workflow demonstrates end-to-end GPU pipeline with minimal host interaction.

Preparing Data on GPU

```python
import cupy as cp

# Create sparse input (Dirac comb) on GPU
diracs = cp.zeros((4096, 4096), dtype=cp.float32)
diracs[::512, ::512] = 1.0

# 2D Gaussian-like kernel (31×31 Hanning window)
gauss = cp.outer(cp.hanning(31), cp.hanning(31))
gauss /= gauss.sum()  # normalize to unit sum
```

Allocate arrays directly on device; avoid copying from host to maintain GPU residency.

CuPy

Running GPU Convolution with Timing

```
from cupyx.scipy import signal

t0 = cp.cuda.Event()
t1 = cp.cuda.Event()

t0.record()
blur = signal.convolve2d(diracs, gauss, mode='same', boundary='wrap')
t1.record()
t1.synchronize()

elapsed_ms = cp.cuda.get_elapsed_time(t0, t1)
print(f"GPU convolution: {elapsed_ms:.2f} ms")
```

CUDA events measure kernel execution time independently of Python GIL or host scheduling.

Part 7
Appendix

Hardware and Software Requirements for CuPy

Prerequisites:

- NVIDIA GPU with CUDA Compute Capability ≥ 3.5 or AMD GPU with ROCm support
- CUDA Toolkit (11.x/12.x) or ROCm runtime
- Python 3.8+ and pip for wheel installation

Ensure drivers match toolkit versions; consult CuPy's installation matrix for exact compatibility before selecting wheels.

Installation Quick Start

Ensure the installed CuPy package matches your CUDA or ROCm version—mismatches can lead to import failures or runtime errors.

```
# For CUDA 11.2 ~ 11.x
pip install cupy-cuda11x
# For CUDA 12.x
pip install cupy-cuda12x
# For CUDA 13.x
pip install cupy-cuda13x
# For AMD ROCm 4.3
pip install cupy-rocm-4-3
# For AMD ROCm 5.0
pip install cupy-rocm-5-0
# Verify installation
python -c "import cupy; print(cupy.__version__)"
```

CuPy

ElementwiseKernel for Fused Operations

```
scale_bias = cp.ElementwiseKernel(
    'float32 x, float32 s, float32 b',  # inputs
    'float32 y',                         # output
    'y = s * x + b;',                    # C++ operation
    'scale_bias'                         # name
)

data = cp.arange(8, dtype=cp.float32)
result = scale_bias(data, 0.5, 1.0)
# result = 0.5 * data + 1.0
print(result)
# array([1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5], dtype=float32)
```

Fuses scalar arithmetic into single kernel launch; reduces memory traffic vs chaining separate ufuncs.

RawKernel for Full CUDA Control

```
code = r'''
extern "C" __global__
void saxpy(const float* x, const float* y, float* z,
           float a, int n) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < n) z[i] = a * x[i] + y[i];
}'''
saxpy = cp.RawKernel(code, 'saxpy')
n = 1 << 20  # 1M elements
x = cp.random.rand(n, dtype=cp.float32)
y = cp.random.rand(n, dtype=cp.float32)
z = cp.empty(n, dtype=cp.float32)
# Launch with grid/block dimensions
blk = 256
grd = (n + blk - 1) // blk
saxpy((grd,), (blk,), (x, y, z, cp.float32(2.0), n))
```

RawKernel JIT-compiles CUDA C++; cached per device for reuse.

RawKernel with Shared Memory Reduction

```
code = r'''
extern "C" __global__
void block_sum(const float* x, float* out, int n) {
    extern __shared__ float sdata[];
    int tid = threadIdx.x;
    int i = blockIdx.x * blockDim.x + tid;
    sdata[tid] = (i < n) ? x[i] : 0.0f;
    __syncthreads();
    for (int s = blockDim.x / 2; s > 0; s >>= 1) {
        if (tid < s) sdata[tid] += sdata[tid + s];
        __syncthreads();
    }
    if (tid == 0) out[blockIdx.x] = sdata[0];
}'''
kernel = cp.RawKernel(code, 'block_sum')
# Launch with shared memory size (3rd arg to __call__)
```

Shared memory tiling reduces global memory accesses; critical for bandwidth-bound reductions.

RawModule for Multiple Kernels

```
code = r'''
extern "C" __global__ void kernel_a(float* x, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) x[i] *= 2.0f;
}
extern "C" __global__ void kernel_b(float* x, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) x[i] += 1.0f;
}'''
mod = cp.RawModule(code=code)
ka = mod.get_function('kernel_a')
kb = mod.get_function('kernel_b')

x = cp.ones(1024, dtype=cp.float32)
ka((4,), (256,), (x, 1024))  # x *= 2
kb((4,), (256,), (x, 1024))  # x += 1
print(x[:5])  # [3. 3. 3. 3. 3.]
```

RawModule compiles once; retrieve multiple kernels from the same source.