Accelerating your code with GPUs

Shao-Ching Huang

2025-10-30

Contents

Workshop series:

1. Foundations for sustainable research software (October 9)
2. Scaling your science with parallel computing (October 16)
3. **Accelerating your code with GPUs (October 30)**

Resources:

- Workshop materials [link]
- UCLA Office of Advanced Research Computing [link]
- Hoffman2 Cluster [link]

1 CPU vs. GPU Computing

This document provides essential background on parallel computing, covering both CPU vectorization (SIMD) and GPU computing. Understanding these concepts is crucial for effective Python GPU programming with CUDA, CuPy, Numba, CUDA Python, and JAX.

1.1 Overview

Modern computing relies on parallelism to achieve high performance. Two dominant paradigms exist:

CPU Parallelism (SIMD):

- Single Instruction, Multiple Data
- Vector processing within CPU cores
- 4-16 way parallelism per core
- Latency-oriented design

GPU Parallelism (SIMT):

- Single Instruction, Multiple Threads
- Massive thread-level parallelism
- Thousands of threads executing concurrently
- Throughput-oriented design

Why This Matters:

- Python frameworks leverage both CPU and GPU parallelism
- NumPy uses CPU SIMD (AVX-512) for vectorized operations
- CuPy, Numba, JAX use GPU SIMT for massive parallelism
- Understanding both helps you choose the right tool
- Critical for scientific computing, data analysis, and machine learning

1.2 CPU Vector Processing (SIMD)

CPU vector processing uses SIMD (Single Instruction, Multiple Data) to perform the same operation on multiple data elements simultaneously.

1.2.1 What is SIMD

Core Concept:

- Single instruction operates on multiple data elements in parallel
- Uses special wide vector registers to hold multiple values
- One instruction can add/multiply 4-16 numbers simultaneously
- Available on all modern CPUs (Intel, AMD, ARM)

Example:

```
Traditional (Scalar):
for i in range(8):
    c[i] = a[i] + b[i]  # 8 separate additions

SIMD (Vector):
c[0:8] = a[0:8] + b[0:8]  # Single instruction, 8 additions in parallel
```

1.2.2 CPU SIMD Instruction Sets

Modern CPUs have evolved through multiple generations of SIMD instructions:

Intel/AMD Evolution:

Instruction Set	Year	Register Width	Float32 Parallel Ops
MMX	1997	64-bit	N/A (integer only)
SSE	1999	128-bit (XMM)	4
SSE2	2001	128-bit (XMM)	4
AVX	2011	256-bit (YMM)	8
AVX2	2013	256-bit (YMM)	8 (improved)
AVX-512	2016	512-bit (ZMM)	16

Example SIMD Instructions:

- `vmulps`: Vector multiply packed single-precision (8 multiplications at once with AVX)
- `vaddpd`: Vector add packed double-precision (4 additions at once with AVX)
- `vfmadd`: Vector fused multiply-add (8 operations at once with AVX)

1.2.3 How CPUs Use SIMD

Vector Registers:

- Special wide registers hold multiple values
- AVX: 256-bit YMM registers (8 × float32 or 4 × float64)
- AVX-512: 512-bit ZMM registers (16 × float32 or 8 × float64)
- Data must be aligned in memory for best performance

Programming SIMD:

1. **Automatic (Compiler)**: Compiler auto-vectorizes loops

```
for (int i = 0; i < n; i++) {
    c[i] = a[i] + b[i];  // Compiler may use SIMD
}
```

2. **Explicit (Intrinsics)**: Direct SIMD programming

```
__m256 a_vec = _mm256_load_ps(&a[i]);
__m256 b_vec = _mm256_load_ps(&b[i]);
__m256 c_vec = _mm256_add_ps(a_vec, b_vec);
_mm256_store_ps(&c[i], c_vec);
```

3. **Library (NumPy)**: Libraries use optimized SIMD

```
import numpy as np
c = a + b  # NumPy uses AVX/AVX-512 internally
```

1.2.4 CPU SIMD Performance

Typical Speedups:

- SSE (4-way): ~3-4x over scalar code
- AVX (8-way): ~6-8x over scalar code
- AVX-512 (16-way): ~10-15x over scalar code
- Actual speedup depends on memory bandwidth and algorithm

Example CPU Peak Performance:

- Intel Xeon Platinum 8380 (40 cores, AVX-512):
  - Peak FP32: ~5 TFLOPS
  - Peak FP64: ~2.5 TFLOPS
- AMD EPYC 9654 (96 cores, AVX-512):
  - Peak FP32: ~11 TFLOPS
  - Peak FP64: ~5.5 TFLOPS

1.2.5 CPU Design Philosophy: Latency-Oriented

Optimization Goals:

- Minimize latency for single-threaded execution
- Fast execution of individual instruction sequences
- Complex control logic for branch prediction
- Large caches to reduce memory latency

Silicon Budget:

- ~50-60% for cache memory (L1/L2/L3)
- ~20-30% for control logic (branch prediction, out-of-order execution)
- ~10-20% for ALUs (arithmetic logic units)
- Small fraction dedicated to vector units (SIMD)

Strengths:

- Excellent for sequential algorithms
- Good for complex control flow
- Low latency for individual operations
- Versatile general-purpose computing

Limitations:

- Limited parallelism (4-16 way SIMD per core)
- Even with 64 cores, peak throughput is modest
- SIMD complexity increases with wider vectors
- Diminishing returns beyond AVX-512

1.3 GPU Computing: Massive Parallelism

GPUs evolved from graphics accelerators to general-purpose parallel processors capable of executing thousands of threads concurrently.

1.3.1 Brief History

Evolution of GPU Computing:

- **1999**: NVIDIA GeForce 256 - first consumer GPU
- **2006**: NVIDIA CUDA - GPU computing platform launched
- **2007**: AMD Stream SDK - AMD's GPGPU initiative
- **2010**: NVIDIA Fermi - unified compute/graphics architecture
- **2017**: NVIDIA Volta - Tensor Cores for AI
- **2020**: NVIDIA Ampere A100 - dominant data center GPU
- **2020**: AMD MI100 CDNA - dedicated compute architecture
- **2022**: NVIDIA Hopper H100 - transformer engine for LLMs
- **2023**: NVIDIA Grace Hopper (CPU+GPU) — tightly integrated CPU and GPU architecture enabling unified memory and large-scale AI workloads
- **Present**: GPUs essential for AI, scientific computing, data analytics

Industry Landscape:

- NVIDIA: ~80-90% of GPU computing market (CUDA platform)
- AMD: Growing with ROCm open-source platform
- Intel: Entering with Data Center GPU Max series
- Apple: Unified memory architecture with Metal

1.3.2 GPU Architecture Overview

Key Characteristics:

- **Massive Parallelism**: Thousands of cores executing concurrently
- **Throughput-Oriented**: Optimized for processing large amounts of data
- **SIMT Execution**: Single Instruction, Multiple Threads
- **Specialized Memory**: Multiple memory types for different access patterns
- **Heterogeneous**: Works with CPU for maximum performance

Why GPUs Excel:

- 10-1000x speedup over CPU for parallel workloads
- Essential for deep learning and AI
- Ideal for array operations and linear algebra
- High memory bandwidth (1-3 TB/s vs 50-100 GB/s for CPU)
- Cost-effective performance per watt

1.3.3 GPU vs CPU: Architectural Comparison

Design Philosophy:

Aspect	CPU (Latency-Oriented)	GPU (Throughput-Oriented)
Optimization Goal	Minimize latency for single thread	Maximize throughput for many threads
Core Count	4-64 powerful cores	2,000-10,000+ simple cores
Core Complexity	Complex (branch prediction, OoO)	Simple (in-order execution)
Cache Size	Large (megabytes L1/L2/L3)	Small (kilobytes per SM)
Control Logic	Extensive (~30% of silicon)	Minimal (~5% of silicon)
ALU Density	~10-20% of silicon	~70-80% of silicon
Memory Strategy	Large caches hide latency	Parallelism hides latency

Visual Comparison:

```
CPU Die:                          GPU Die:

  Cache    Cache                 Compute Compute

 Control   Cache                 Compute Compute
```

Performance Comparison:

Metric	Typical CPU	Typical GPU
Cores	16-64	2,000-10,000+
Peak FP32	1-10 TFLOPS	20-80 TFLOPS
Peak FP64	0.5-5 TFLOPS	10-20 TFLOPS
Memory Bandwidth	50-100 GB/s	500-3000 GB/s
Power	100-300W	250-700W
TFLOPS/Watt	0.01-0.05	0.05-0.3

1.3.4 When to Use CPU vs GPU

Use CPU for:

- Sequential algorithms with complex control flow
- Small datasets ($< 10{,}000$ elements)
- Tasks requiring low latency (single operation)
- I/O-bound operations
- Irregular memory access patterns
- Code with many branches and conditionals

Use GPU for:

- Data-parallel algorithms (same operation on many elements)
- Large datasets (millions of elements)
- Regular, predictable memory access patterns
- Compute-intensive operations (matrix multiplication, FFT)
- Can tolerate higher latency if throughput is high
- Operations that can be expressed as kernels on arrays

1.4 SIMD vs SIMT: Detailed Comparison

Understanding the differences between CPU SIMD and GPU SIMT helps you leverage both effectively.

1.4.1 Execution Models

CPU SIMD (Single Instruction, Multiple Data):

- Single instruction operates on fixed-size vector register
- 4-16 data elements processed in parallel
- Programmer manages vector operations explicitly (or compiler auto-vectorizes)

- Limited by vector register width

GPU SIMT (Single Instruction, Multiple Threads):

- Single instruction issued to group of threads (warp)
- 32 threads (NVIDIA) or 64 threads (AMD) execute together
- Each thread has its own registers and can follow independent control flow
- Massive scalability: thousands of warps

1.4.2 Comparison Table

Feature	CPU SIMD	GPU SIMT
Parallelism Scale	4-16 way per core	1000s of threads per GPU
Execution Unit	Vector register (256-512 bit)	Warp (32 threads on NVIDIA)
Control Flow	Difficult with divergence	Handles divergence naturally
Memory Model	Shared cache hierarchy	Separate memory spaces (global, shared, registers)
Latency Hiding	Cache reduces memory latency	Thread switching hides latency
Programming Model	Vectorized loops or intrinsics	Kernel functions with thread indexing
Best For	Moderate parallelism within CPU	Massive data parallelism
Peak Performance	~1-10 TFLOPS	~20-80 TFLOPS

1.4.3 Control Flow Divergence

CPU SIMD Divergence:

```
// SIMD code with divergence
for (int i = 0; i < 8; i++) {
    if (data[i] > 0) {
        result[i] = compute_A(data[i]);  // Complex with SIMD
    } else {
        result[i] = compute_B(data[i]);
    }
}
```

- SIMD struggles with divergent paths
- Often requires predication or masking
- Compiler may serialize the operations
- Performance degrades significantly

GPU SIMT Divergence:

```
__global__ void kernel(float* data, float* result, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n) {
        if (data[idx] > 0) {
            result[idx] = compute_A(data[idx]);  // Natural with SIMT
```

```
        } else {
            result[idx] = compute_B(data[idx]);
        }
    }
}
```

- SIMT handles divergence naturally
- Warp executes both paths with different threads active
- Uses per-thread active masks
- Performance impact: 2x slowdown (not serialization)
- Modern GPUs (Volta+) have Independent Thread Scheduling

Warp Divergence Example:

```
Warp with 32 threads:
Threads 0-15: data[i] > 0 is true → execute compute_A()
Threads 16-31: data[i] > 0 is false → execute compute_B()

Execution:
Step 1: Run compute_A() with threads 0-15 active (threads 16-31 masked)
Step 2: Run compute_B() with threads 16-31 active (threads 0-15 masked)
Step 3: Reconverge - all threads continue together
```

1.5 NVIDIA GPU Architecture

NVIDIA GPUs dominate scientific computing. Understanding their architecture is essential for Python GPU programming.

1.5.1 Streaming Multiprocessors (SMs)

Core Building Block:

- GPU consists of multiple Streaming Multiprocessors (SMs)
- Each SM contains many CUDA cores (Streaming Processors)
- SMs execute thread blocks independently
- Number of SMs varies by GPU model

Example GPU Configurations:

GPU Model	Architecture	SMs	CUDA Cores	FP32 TFLOPS	Memory
RTX 3090	Ampere	82	10,496	35.6	24 GB
A100 (40GB)	Ampere	108	6,912	19.5	40 GB
H100	Hopper	132	16,896	51.0	80 GB
RTX 4090	Ada Lovelace	128	16,384	82.6	24 GB

SM Components:

- CUDA Cores: Execute arithmetic operations
- Special Function Units (SFUs): Fast transcendentals (sin, cos, exp)

- Tensor Cores: Matrix multiply-accumulate (AI/ML acceleration)
- Shared Memory: Fast on-chip memory (48-164 KB per SM)
- Registers: Per-thread storage (~64K registers per SM)
- Warp Scheduler: Selects warps for execution

1.5.2 Thread Hierarchy: Grids, Blocks, Warps

Three-Level Organization:

1. Thread (Finest Granularity):

- Single execution context running kernel code
- Has unique thread ID within its block
- Operates on one data element
- Has private registers and local memory

2. Block (Thread Block):

- Group of threads (typically 128-1024 threads)
- All threads in a block execute on the same SM
- Threads can cooperate via shared memory
- Can synchronize using barriers (`__syncthreads()`)
- Block has unique block ID within the grid

3. Grid:

- Collection of blocks executing the same kernel
- Can contain thousands of blocks
- Blocks execute independently (no synchronization across blocks)
- Grid spans entire problem domain

4. Warp (Execution Unit):

- Group of 32 threads that execute together
- Fundamental unit of execution on NVIDIA GPUs
- All threads in a warp execute same instruction simultaneously
- Hardware schedules entire warps, not individual threads

Visual Hierarchy:

```
Grid (launched by CPU)
     Block 0 (executes on SM)
          Warp 0 (threads 0-31)
          Warp 1 (threads 32-63)
          Warp 2 (threads 64-95)
          Warp 3 (threads 96-127)
     Block 1 (executes on SM)
          Warp 0
          ...
     Block N-1
```

1.5.3 Latency Hiding Through Parallelism

The Problem:

- Global memory access takes 400-800 cycles
- Arithmetic operations are much faster (1-10 cycles)
- If GPU waited for memory, cores would be idle

The Solution:

- SM has many more warps resident than it can execute simultaneously
- When a warp stalls (waiting for memory), SM instantly switches to another ready warp
- Zero-overhead scheduling: no context switch cost
- Keeps compute units busy while memory operations complete

Example:

```
SM with 64 CUDA cores, can execute 2 warps simultaneously:
- 32 resident warps total (1024 threads)
- Currently executing: Warp 0 and Warp 1
- Warp 0 stalls on memory load
- SM immediately switches to Warp 2 (no overhead)
- Warp 0's memory load completes in background
- Warp 0 becomes ready again, will execute later
```

Occupancy:

- Percentage of maximum resident warps actually achieved
- Higher occupancy = better latency hiding
- Limited by register usage and shared memory per thread/block
- Target: 50-100% occupancy for good performance

1.6 GPU Memory Hierarchy

GPUs have multiple memory types optimized for different access patterns.

1.6.1 Memory Types

1. Global Memory (Device Memory):

- Largest capacity (8-80 GB)
- Accessible by all threads and host CPU
- Slowest (~400-800 cycles latency)
- Highest bandwidth (500-3000 GB/s)
- Off-chip GDDR6/HBM2/HBM3
- This is the "24 GB" or "80 GB" in GPU specs

2. Shared Memory:

- Small capacity (~48-164 KB per SM)
- Accessible by threads in same block only
- Fast (~1-2 cycles latency)
- On-chip memory (part of SM)

- Programmer-managed
- Used for thread cooperation and data reuse

3. Registers:

- Tiny capacity (~64K 32-bit registers per SM)
- Private to each thread
- Fastest (<1 cycle)
- On-chip, part of SM
- Compiler-managed
- Limited supply affects occupancy

4. L1/L2 Cache:

- L1: 128 KB per SM (Ampere/Hopper)
- L2: 6-50 MB shared across GPU
- Automatic caching of global memory
- Hardware-managed
- Reduces global memory latency

5. Constant Memory:

- Small capacity (64 KB)
- Read-only from kernels
- Cached and broadcast efficiently
- Good for parameters used by all threads

1.6.2 Memory Performance

Bandwidth Comparison:

Memory Type	Latency	Bandwidth	Capacity
Registers	<1 cycle	~20 TB/s	~256 KB/SM
Shared Memory	1-2 cycles	~15 TB/s	48-164 KB/SM
L1 Cache	~30 cycles	~10 TB/s	128 KB/SM
L2 Cache	~200 cycles	~3 TB/s	6-50 MB
Global Memory	400-800 cycles	1-3 TB/s	8-80 GB

1.6.3 Memory Access Patterns

Coalesced Access (Efficient):

- Consecutive threads access consecutive memory addresses
- GPU combines into single memory transaction
- Example: `data[threadIdx.x]` with 32 threads → one transaction
- Maximum bandwidth utilization

Strided Access (Less Efficient):

- Threads access memory with constant stride
- May require multiple memory transactions

- Example: `data[threadIdx.x * 2]` → two transactions for warp
- Performance: ~50% of coalesced

Random Access (Inefficient):

- Threads access unpredictable memory locations
- Each access may be separate transaction
- Poor cache utilization
- Performance: <10% of coalesced

1.7 Other GPU Architectures

While NVIDIA dominates, other vendors offer alternatives.

1.7.1 AMD ROCm

Platform:

- ROCm: Radeon Open Compute (open-source)
- HIP: Portable GPU programming (CUDA-compatible syntax)
- Growing ecosystem, smaller than CUDA

Python Support:

- CuPy: Experimental ROCm backend
- PyTorch: ROCm support
- TensorFlow: ROCm builds available

Terminology Differences:

Concept	NVIDIA CUDA	AMD ROCm
Processing Unit	Streaming Multiprocessor (SM)	Compute Unit (CU)
Execution Unit	CUDA Core	Stream Processor
Thread Group	Warp (32 threads)	Wavefront (64 threads)
Fast Memory	Shared Memory	Local Data Share (LDS)

Hardware:

- Radeon Instinct MI100/MI200/MI300: Data center compute
- RX 6000/7000: Consumer GPUs with limited compute support

1.7.2 Intel oneAPI

Platform:

- oneAPI: Cross-architecture programming
- SYCL: Open standard based on C++
- Supports Intel GPUs, CPUs, FPGAs

Hardware:

- Intel Data Center GPU Max series

- Intel Arc: Consumer GPUs
- Integrated GPUs in Intel CPUs

Status: Emerging platform, smaller ecosystem

1.7.3 Apple Silicon

Platform:

- Metal: GPU programming framework
- Unified memory (CPU and GPU share RAM)
- Metal Performance Shaders (MPS)

Python Support:

- PyTorch: MPS backend
- TensorFlow: Metal plugin

Characteristics:

- No discrete GPU memory copies needed
- Good power efficiency
- Limited to Apple hardware

1.7.4 Platform Comparison

Platform	Vendor	Maturity	Ecosystem	Python Support
NVIDIA CUDA	NVIDIA only	Mature (18+ years)	Extensive	Excellent
AMD ROCm	AMD only	Growing (8+ years)	Moderate	Good
Intel oneAPI	Intel only	New (3+ years)	Emerging	Limited
Apple Metal	Apple only	Mature (graphics)	Moderate	Limited
OpenCL	Multi-vendor	Mature but stagnant	Moderate	Moderate

1.8 Summary: Key Concepts for Python Programming

1.8.1 Essential Concepts

1. CPU SIMD:

- 4-16 way parallelism using vector registers
- NumPy leverages AVX/AVX-512 automatically
- Good for moderate parallelism on CPU
- Limited scalability compared to GPU

2. GPU SIMT:

- Thousands of threads executing concurrently

- Warp-based execution (32 threads per warp on NVIDIA)
- Massive parallelism for data-parallel workloads
- Higher throughput than CPU SIMD

3. Latency vs Throughput:

- CPU optimized for latency (single operation fast)
- GPU optimized for throughput (many operations total)
- Choose based on problem characteristics

4. Memory Hierarchy:

- GPU has complex memory hierarchy
- Coalesced access patterns critical for performance
- Global memory is large but slow
- Shared memory is small but fast

5. Control Flow:

- CPU SIMD struggles with divergence
- GPU SIMT handles divergence naturally (with performance cost)
- Minimize divergence within warps for best performance

1.8.2 For Python GPU Frameworks

NumPy (CPU SIMD):

```
import numpy as np
c = a + b  # Uses AVX-512 on supported CPUs
```

CuPy (GPU, High-Level):

```
import cupy as cp
c = a + b  # Same API, runs on GPU with massive parallelism
```

Numba (GPU, Medium-Level):

```
from numba import cuda
@cuda.jit
def kernel(a, b, c):
    i = cuda.grid(1)
    if i < c.size:
        c[i] = a[i] + b[i]  # Explicit thread programming
```

JAX (GPU, Functional):

```
import jax.numpy as jnp
c = a + b  # Auto-parallelized, auto-differentiated
```

1.8.3 When GPU Accelerates Your Code

Good for GPU:

- Large arrays (millions of elements)

- Data-parallel operations (element-wise, matrix ops)
- Regular memory access patterns
- Many operations per data element
- Data stays on GPU across multiple operations

Not Good for GPU:

- Small datasets (< 10,000 elements)
- Sequential algorithms
- Complex control flow with divergence
- Irregular memory access
- One-off operations (transfer overhead dominates)

1.8.4 Performance Expectations

Typical Speedups:

- CPU SIMD over scalar: 3-15x
- GPU over CPU (well-suited problems): 10-1000x
- GPU over CPU (poorly-suited problems): <2x or slower

Example:

- Matrix multiplication (5000×5000):
  - Pure Python: ~300 seconds
  - NumPy (CPU SIMD): ~3 seconds (100x faster)
  - CuPy (GPU): ~0.03 seconds (10,000x faster than Python, 100x faster than NumPy)

This foundation prepares you for GPU programming in Python using CUDA, CuPy, Numba, CUDA Python, and JAX covered in the following sections.

2 CUDA C

This document provides a foundation for understanding CUDA (Compute Unified Device Architecture) and writing simple CUDA C kernels. These concepts are essential for using CuPy's RawKernel feature and understanding GPU-accelerated Python frameworks.

2.1 Overview

CUDA is NVIDIA's parallel computing platform and programming model for GPU computing. It allows developers to harness the massive parallelism of NVIDIA GPUs for general-purpose computing, not just graphics.

Key Concepts:

- **Parallel Computing Platform**: Enables GPU acceleration for scientific computing, data analysis, machine learning, and more
- **Extension of C/C++**: CUDA C/C++ adds GPU programming keywords to standard C/C++
- **Heterogeneous Computing**: Programs execute on both CPU (host) and GPU (device)
- **Massive Parallelism**: Modern GPUs have thousands of cores executing threads concurrently

2.2 Brief History

CUDA was introduced by NVIDIA in 2006 as a way to make GPU computing accessible beyond graphics programming.

Key Milestones:

- **2006**: CUDA 1.0 released, making GPU computing accessible to C programmers
- **2008**: CUDA 2.0 added shared memory and synchronization primitives
- **2010**: Fermi architecture introduced unified address space
- **2012**: Kepler architecture brought dynamic parallelism
- **2014**: Maxwell architecture improved power efficiency
- **2016**: Pascal architecture added unified memory improvements
- **2017**: Volta architecture introduced Tensor Cores for AI workloads
- **2018**: Turing architecture enhanced ray tracing capabilities
- **2020**: Ampere architecture (A100) for data centers
- **2022**: Hopper architecture (H100) for extreme-scale AI
- **Present**: CUDA is the dominant GPU computing platform for scientific computing and AI

CUDA has become the de facto standard for GPU computing, with extensive library support (cuBLAS, cuFFT, cuDNN) and a large developer community.

2.3 CUDA Programming Model

CUDA uses a heterogeneous computing model where code runs on both CPU and GPU.

2.3.1 Host and Device

Host (CPU):

- Manages overall program flow
- Allocates GPU memory
- Copies data between CPU and GPU
- Launches kernels (GPU functions)
- Retrieves results from GPU

Device (GPU):

- Executes massively parallel computations
- Operates on data in GPU memory
- Thousands of threads run concurrently
- Optimized for throughput, not latency

Typical Workflow:

1. Allocate memory on GPU
2. Copy input data from CPU to GPU
3. Launch kernel to process data on GPU
4. Copy results from GPU back to CPU
5. Free GPU memory

2.3.2 Kernels

What are Kernels:

- Functions that execute on the GPU
- Launched by CPU, executed by thousands of GPU threads
- Each thread runs the same kernel code but operates on different data
- Written in CUDA C/C++ with special syntax

Kernel Launch:

```
// Kernel definition
__global__ void my_kernel(float* data, int n) {
    // Kernel code here
}

// Kernel launch from host
my_kernel<<<blocks, threads>>>(data, n);
```

Key Points:

- `__global__` keyword marks a function as a kernel
- Triple angle brackets `<<<...>>>` specify execution configuration
- First parameter: number of thread blocks
- Second parameter: number of threads per block

2.4 GPU Architecture Essentials

Understanding GPU hardware organization helps you write efficient CUDA code.

2.4.1 Streaming Multiprocessors (SMs)

Architecture Overview:

- GPU consists of multiple Streaming Multiprocessors (SMs)
- Each SM contains many CUDA cores (Streaming Processors)
- Modern GPUs have 10-100+ SMs, each with 64-128 cores
- Example: A100 GPU has 108 SMs with 64 FP32 cores each = 6,912 CUDA cores

Execution Model:

- Each SM executes one or more thread blocks
- Threads within a block can cooperate via shared memory
- Threads in different blocks cannot directly communicate
- SM schedules threads in groups of 32 called warps

2.4.2 Thread Hierarchy: Grids, Blocks, and Threads

CUDA organizes threads in a hierarchical structure to map computation onto GPU hardware.

Thread:

- Smallest unit of execution

- Executes kernel code on one data element
- Has unique thread ID within its block
- Runs on a CUDA core

Warp:

- Group of **32 threads** that execute together in lockstep
- The actual scheduling unit on an SM
- All threads in a warp execute the same instruction simultaneously (SIMT)
- If threads diverge (different branches), execution is serialized until reconvergence
- Performance consideration: Keep threads in a warp following the same path

Block (Thread Block):

- Group of **threads** (typically 128-1024 threads)
- Divided into warps of 32 threads each
- Threads in a block execute on the same SM
- Can cooperate via shared memory
- Can synchronize with barriers
- Has unique block ID within the grid

Grid:

- Collection of **blocks** executing the same kernel
- Can contain thousands of blocks
- Blocks execute independently (any order)
- Grid spans entire problem domain

Visual Hierarchy:

```
Grid (entire problem)
  Block 0
     Thread 0
     Thread 1
     ...
     Thread N-1
  Block 1
     Thread 0
     Thread 1
     ...
     Thread N-1
  Block M-1
      Thread 0
      Thread 1
      ...
      Thread N-1
```

2.5 Thread Coordinates and Indexing

Every CUDA thread needs to know its unique position to determine which data element to process.

2.5.1 Built-in Variables

CUDA provides built-in variables accessible within kernel code:

Thread Dimensions and Indices:

Variable	Type	Description
`threadIdx.x`	uint3	Thread index within its block (0 to blockDim.x-1)
`blockIdx.x`	uint3	Block index within the grid (0 to gridDim.x-1)
`blockDim.x`	dim3	Number of threads per block (set at launch)
`gridDim.x`	dim3	Number of blocks in the grid (set at launch)

Note: Variables have `.x`, `.y`, `.z` components for 1D, 2D, or 3D indexing

2.5.2 Calculating Global Thread Index

The most common task is calculating a unique global index for each thread to access array elements.

1D Grid and 1D Blocks (Most Common):

```
int idx = blockIdx.x * blockDim.x + threadIdx.x;
```

Breakdown:

- `blockIdx.x * blockDim.x`: Starting position of current block in global array
- `+ threadIdx.x`: Offset within the block
- Result: Unique index for each thread across entire grid

Example:

```
Grid: 3 blocks, each with 4 threads

Block 0:                Block 1:                Block 2:
Thread 0 → idx = 0     Thread 0 → idx = 4     Thread 0 → idx = 8
Thread 1 → idx = 1     Thread 1 → idx = 5     Thread 1 → idx = 9
Thread 2 → idx = 2     Thread 2 → idx = 6     Thread 2 → idx = 10
Thread 3 → idx = 3     Thread 3 → idx = 7     Thread 3 → idx = 11
```

Formula Visualization:

```
Block 0: (0 * 4) + 0,1,2,3  →  0,1,2,3
Block 1: (1 * 4) + 0,1,2,3  →  4,5,6,7
Block 2: (2 * 4) + 0,1,2,3  →  8,9,10,11
```

2.5.3 2D Indexing

For 2D problems (images, matrices):

```
int x = blockIdx.x * blockDim.x + threadIdx.x;
int y = blockIdx.y * blockDim.y + threadIdx.y;
int idx = y * width + x;  // Convert 2D to 1D for array access
```

Use case: Image processing, matrix operations

2.6 Writing Simple CUDA C Kernels

Function Qualifiers:

- `__global__`: Marks function as a kernel (callable from host, runs on device)
- `__device__`: Function runs on device, callable only from device
- `__host__`: Function runs on host (normal C function)

`extern "C"`:

- Prevents C++ name mangling
- Allows Python to find the kernel by name
- Required when using CUDA C++ code with Python

Note: C++ name mangling is a technique where the C++ compiler encodes additional information into function names to support features like function overloading and namespaces.

Pointer Types:

- `const float* input`: Read-only input array
- `float* output`: Output array (writable)
- Pointers reference GPU global memory

2.6.1 Example: Vector Addition Kernel

Complete CUDA C Kernel:

```
extern "C" __global__
void vector_add(const float* a, const float* b, float* c, int n) {
    // Calculate global thread index
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    // Boundary check: ensure thread doesn't access beyond array
    if (idx < n) {
        c[idx] = a[idx] + b[idx];
    }
}
```

Notes:

1. `extern "C" __global__`: Kernel declaration for Python interop
2. `void vector_add(...)`: Kernel name and parameters
    - `const float* a, b`: Input arrays (read-only)
    - `float* c`: Output array (writable)
    - `int n`: Array size
3. `int idx = ...`: Calculate this thread's unique global index

4. `if (idx < n)`: Boundary check (explained below)
5. `c[idx] = a[idx] + b[idx]`: Actual computation

2.6.2 Boundary Checking

- Grid size may not perfectly divide array size
- You might launch 256 threads but only have 250 elements
- Extra threads must not access invalid memory
- Accessing out-of-bounds causes crashes or corruption

Example Scenario:

- Array size: 1000 elements
- Block size: 256 threads
- Blocks needed: ceil(1000/256) = 4 blocks
- Total threads launched: 4 * 256 = 1024 threads
  - Threads 0-999: Process valid data
  - Threads 1000-1023: Must be filtered out with boundary check

2.7 Memory Hierarchy

Understanding GPU memory is important for performance optimization.

Memory Types:

Memory Type	Location	Access Speed	Scope	Lifetime
Global Memory	GPU DRAM	Slow (~400 cycles)	All threads	Program duration
Shared Memory	On-chip (SM)	Fast (~1 cycle)	Block threads	Block lifetime
Registers	On-chip (SM)	Fastest	Single thread	Thread lifetime
Local Memory	GPU DRAM	Slow	Single thread	Thread lifetime
Constant Memory	GPU DRAM	Slow (cached)	All threads (read-only)	Program duration

For Simple Kernels:

- Use global memory for input/output arrays
- Let compiler manage registers for local variables
- Shared memory for advanced optimization (not covered here)

Accessing Global Memory:

```
__global__ void kernel(float* global_array, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < n) {
```

```
        // Read from global memory
        float value = global_array[idx];

        // Compute (uses registers)
        float result = value * 2.0f;

        // Write to global memory
        global_array[idx] = result;
    }
}
```

Accessing Shared Memory:

```
__global__ void kernel_with_shared(float* global_array, int n) {
    // Declare shared memory (shared by all threads in the block)
    __shared__ float shared_data[256];

    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int local_idx = threadIdx.x;

    if (idx < n) {
        // Read from global memory into shared memory
        shared_data[local_idx] = global_array[idx];

        // Synchronize to ensure all threads have loaded data
        __syncthreads();

        // Now all threads can access shared memory (fast)
        float value = shared_data[local_idx];
        float result = value * 2.0f;

        // Synchronize before writing back
        __syncthreads();

        // Write result to global memory
        global_array[idx] = result;
    }
}
```

Note: Global vs Local Memory

Despite the name, **local memory is NOT faster than global memory** - both reside in slow GPU DRAM (~400 cycles). The "local" refers to visibility (single thread only), not speed.

Local memory is automatically used when: - Too many variables to fit in registers (register spilling) - Large thread-local arrays

Example:

```
__global__ void kernel() {
    float temp[100];  // Too large for registers → spills to local memory (slow!)
}
```

Note: Minimize local memory by keeping thread-local variables small so they stay in registers (fastest).

Constant Memory

Constant memory is for **read-only data that all threads access**. Although stored in GPU DRAM, it's cached and broadcast-efficient.

Use for: - Mathematical constants (, gravity, etc.) - Lookup tables and coefficients - Configuration parameters

Example:

```
__constant__ float PI = 3.14159f;
__constant__ float coefficients[256];

__global__ void kernel() {
    float result = data * PI;  // All threads read same value (cached & broadcast)
}
```

Key benefit: When all threads in a warp read the same location, one fetch is broadcast to all threads (much faster than global memory). Limited to 64KB per kernel.

2.8 Summary: CUDA C

Key Concepts to Remember:

1. **Thread Indexing**: Always calculate global index with `blockIdx.x * blockDim.x + threadIdx.x`
2. **Boundary Checking**: Always check `if (idx < n)` before array access
3. **Kernel Syntax**: Use `__global__` to mark GPU kernel functions
4. **Launch Configuration**: Determine blocks and threads:

   ```
   threads_per_block = 256
   blocks = (array_size + threads_per_block - 1) / threads_per_block
   kernel<<<blocks, threads_per_block>>>(args);
   ```

5. **Memory**: Pointers in kernels reference GPU global memory

Resources for CUDA

- NVIDIA CUDA C Programming Guide
- Programming Massively Parallel Processors: A Hands-on Approach

3 GPU Computing Ecosystem in Python

This document provides a quick overview of Python frameworks for GPU computing. Use this guide to understand your options and choose the right tool for your needs. Detailed tutorials are in subsequent documents.

3.1 Overview

Python offers multiple frameworks for GPU acceleration, each designed for different use cases and programming styles.

Key Principle:

- Start high-level (CuPy, JAX) for productivity
- Drop to mid-level (Numba) for custom algorithms
- Use low-level (CUDA Python) only when necessary
- Write CUDA C directly

Most scientific computing stays at high/mid level

3.2 Framework Summaries

3.2.1 CuPy: NumPy Drop-in Replacement

- GPU-accelerated array library with NumPy/SciPy-compatible API
- Often just requires changing `import numpy as np` to `import cupy as cp`
- Leverages optimized CUDA libraries (cuBLAS, cuFFT, cuDNN)

Syntax Style:

```
import cupy as cp
c = a @ b  # Matrix multiply on GPU, just like NumPy
```

Best for:

- Existing NumPy/SciPy code that needs GPU acceleration
- Standard operations: linear algebra, FFT, statistics
- Quick prototyping with minimal code changes
- When you think in terms of arrays and operations

Not ideal for:

- Custom algorithms not expressible as array operations
- Code requiring explicit control over GPU threads

Installation:

```
pip install cupy-cuda12x  # For CUDA 12.x
```

3.2.2 Numba: JIT Compiler for Custom Kernels

- Just-in-Time compiler for Python and NumPy code
- Compiles Python functions to native machine code (CPU or GPU)

- Write GPU kernels in Python syntax, not CUDA C

Syntax Style:

```
from numba import cuda

@cuda.jit
def my_kernel(data):
    idx = cuda.grid(1)
    data[idx] *= 2  # Each GPU thread processes one element
```

Best for:

- Custom algorithms with explicit loops
- Code that doesn't fit NumPy's array operations model
- When you need control over GPU thread organization
- CPU/GPU portable code (same function runs on both)

Not ideal for:

- Simple NumPy operations (use CuPy instead)
- When pre-optimized libraries exist

Example:

```
# CuPy: One line, uses optimized cuBLAS library
import cupy as cp
result = cp.matmul(a, b)  # Or just: a @ b

# Numba: Would require writing a full matrix multiply kernel
# (dozens of lines, harder to optimize than vendor libraries)
@cuda.jit
def matmul_kernel(A, B, C):
    # Complex tiling logic needed for performance
    # Must handle shared memory manually
    # Easy to write slow code
    ...  # 50+ lines of kernel code
```

Installation:

```
pip install numba
```

3.2.3 JAX: Functional Programming with Auto-Differentiation

- Functional array computing library with NumPy-like API
- Automatic differentiation (gradients) built-in
- JIT compilation via XLA compiler
- Composable function transformations

Syntax Style:

```
import jax.numpy as jnp
from jax import jit, grad
```

```
@jit  # Compile for performance
def loss(params):
    return jnp.sum(params ** 2)

gradient = grad(loss)  # Automatic differentiation
```

Best for:

- Machine learning research and optimization
- Problems requiring gradients (inverse problems, parameter estimation)
- Functional programming style
- Multi-GPU/TPU computing with `pmap`
- Same code runs on CPU, GPU, or TPU (hardware-agnostic)

Example:

```
import jax.numpy as jnp
from jax import jit

@jit
def compute(x):
    return jnp.sum(x ** 2)

# Same code, different backends:
# jax.default_device('cpu')  → runs on CPU
# jax.default_device('gpu')  → runs on GPU
# jax.default_device('tpu')  → runs on TPU
```

Not ideal for:

- Simple array operations without gradients (use CuPy)
- Imperative programming style
- Code with side effects or global state

Installation:

```
pip install jax[cuda12]  # For CUDA 12.x
```

3.2.4 CUDA Python: Low-Level GPU Control

- Official Python bindings to CUDA runtime and driver APIs
- Direct access to all CUDA features
- Thin wrapper around CUDA C APIs

Syntax Style:

```
from cuda import cuda
err, device = cuda.cuDeviceGet(0)
err, context = cuda.cuCtxCreate(0, device)
# Manual memory allocation, kernel loading, etc.
```

Best for:

- Integrating existing CUDA C/C++ code
- Building higher-level frameworks
- Performance optimization requiring low-level control
- Custom memory management strategies

Not ideal for:

- Application development (too low-level)
- Most scientific computing tasks

Installation:

```
pip install cuda-python
```

3.3 Comparison

Framework	Abstraction	Ease of Use	Flexibility	Auto-Diff	CPU/GPU Support	Best For
CuPy	High	Easiest	Moderate	No	GPU only	NumPy code acceleration
JAX	High	Easy	High	Yes	CPU/GPU/TPU	Numerical computing with gradients, ML, optimization
Numba	Mid	Moderate	High	No	CPU/GPU	Custom algorithms, loops
CUDA Python	Low	Hard	Maximum	No	GPU only	Framework building, CUDA integration

Performance:

- All can achieve excellent performance when used correctly
- CuPy: Leverages highly optimized vendor libraries
- JAX: XLA compiler produces efficient code
- Numba: Performance matches hand-written CUDA C
- CUDA Python: Maximum control, performance depends on your code

3.4 Interoperability

These frameworks work together through standard protocols:

CUDA Array Interface:

- Zero-copy data exchange between CuPy and Numba
- Arrays share GPU memory, no copying needed

DLPack:

- Zero-copy exchange between CuPy, JAX, PyTorch, TensorFlow
- Industry standard for GPU array interchange

Common Pattern:

```
# Use CuPy for high-level operations
import cupy as cp
data = cp.random.randn(1000000)
result = cp.fft.fft(data)  # CuPy's optimized FFT

# Use Numba for custom kernel on same data
from numba import cuda
@cuda.jit
def custom_process(arr):
    idx = cuda.grid(1)
    if idx < arr.size:
        arr[idx] *= 2

threads = 256
blocks = (data.size + threads - 1) // threads
custom_process[blocks, threads](result)  # Works directly with CuPy array!
```

4 Numba

Numba is a Just-in-Time (JIT) compiler for Python designed to accelerate computationally intensive Python code, particularly on hardware accelerators like NVIDIA GPUs.

4.1 Overview

Numba provides a powerful way to speed up Python code without requiring developers to leave the Python ecosystem or learn lower-level languages. It translates Python functions to optimized machine code at runtime using the industry-standard LLVM compiler library. The key advantage is that developers can write in familiar Python syntax while achieving performance comparable to compiled languages like C or Fortran.

Numba is particularly effective for:

- Array-oriented and math-heavy Python code
- Code with many loops that would typically be slow in pure Python
- NumPy operations that need additional acceleration

- Parallel algorithms that can leverage multiple CPU cores or GPU devices

4.2 History

Numba was created by Travis Oliphant, founder of NumPy and Anaconda, and was first released in 2012. The project emerged from the need to accelerate scientific Python code without requiring developers to write C extensions or Cython.

Key milestones:

- **2012**: Initial release focused on CPU JIT compilation
- **2013**: Added CUDA GPU support, enabling Python programmers to write GPU kernels directly
- **2014**: Became part of the NumFOCUS sponsored projects
- **2015-Present**: Continuous improvements in performance, expanded hardware support, and broader Python language coverage

Numba is now maintained by Anaconda Inc. and has become a core component of the PyData ecosystem, with widespread adoption in scientific computing, data science, and machine learning communities.

4.3 What Numba Does

Core Functionality:

- Compiler for Python array and numerical functions
- Optimized for numerically-oriented code with loops
- Works exceptionally well with NumPy arrays and functions

JIT Compilation:

- Decorators: `@jit`, `@cuda.jit` trigger compilation
- Reads Python bytecode + analyzes input argument types
- Compilation pipeline: **Python → LLVM IR → Machine code**
  - LLVM IR = Intermediate Representation (platform-independent assembly-like code)
  - Enables optimization and portability across different CPUs/GPUs
- Uses LLVM compiler library for optimization
- Generates native machine code for target CPU or GPU

Performance Benefits:

- Achieves C/C++/Fortran-level speeds
- Best performance in `nopython` mode (no Python interpreter overhead)
- Compiles to native machine instructions at runtime

GPU Target Support:

- Supports NVIDIA CUDA GPU programming
  - No AMD GPU support
- Write parallel code in pure Python syntax
- Compiles restricted Python subset to CUDA kernels
- Manages CUDA execution model (threads, blocks, grid)

4.4 Use Cases

Numba has become essential in many scientific computing domains where Python's ease of use needs to be combined with high performance:

- Numerical Simulations: N-body simulations, finite element analysis, computational fluid dynamics
- Signal and Image Processing: Custom FFT implementations, wavelet transforms, medical image analysis
- Monte Carlo Methods: Financial risk analysis, quantum Monte Carlo, Bayesian inference
- Machine Learning and AI: Custom gradient computations, reinforcement learning environments, graph neural networks
- Computational Biology: BLAST-like algorithms, gene expression analysis, population genetics simulations
- Optimization Problems: Genetic algorithms, particle swarm optimization, linear programming solvers

4.5 Compilation Modes

Numba offers different compilation modes that balance performance and compatibility.

4.5.1 Nopython Mode

- The compiled code runs entirely **without the Python interpreter**
- Maximum performance - executes at native machine code speed
- No Python object overhead
- Enabled by default with `@jit` or explicitly with `@jit(nopython=True)`

Benefits:

- Fastest execution possible
- Predictable performance
- No Global Interpreter Lock (GIL) limitations
- Can release the GIL for parallel execution

Limitations:

- Only supports a subset of Python and NumPy features
- Cannot use arbitrary Python objects
- Limited to Numba-supported operations

Example:

```
from numba import jit

@jit(nopython=True)
def compute_sum(arr):
    total = 0.0
    for i in range(arr.shape[0]):
```

```
        total += arr[i]
    return total
```

4.5.2 Object Mode

- Falls back to Python interpreter for unsupported operations
- Allows use of arbitrary Python objects and libraries
- Enabled automatically when nopython mode fails, or explicitly with `@jit(forceobj=True)`

When to use:

- Code contains operations not supported in nopython mode
- Rapid prototyping before optimizing to nopython mode
- Wrapping functions that call other Python libraries

Performance:

- Slower than nopython mode
- Still provides some speedup through type specialization
- Python overhead remains present

4.5.3 Loop-Lifting

Hybrid approach:

- Automatically identifies loops that can be compiled in nopython mode
- Compiles those loops to native code even when the function uses object mode
- Provides partial acceleration without full nopython compatibility

Example:

```
from numba import jit

@jit  # Will use loop-lifting
def mixed_function(arr):
    result = []  # Python list (object mode)
    for i in range(len(arr)):  # This loop can be lifted to nopython
        result.append(arr[i] * 2)
    return result
```

4.6 Type Specialization and Compilation Caching

4.6.1 Type Specialization

- Numba compiles a separate version of the function for each unique combination of input types
- First call with specific types triggers compilation (at runtime)
- Subsequent calls with same types use cached compiled version
- Different type combinations trigger new compilations

Example:

```python
from numba import jit
import numpy as np

@jit
def add_arrays(a, b):
    return a + b

# First call: compiles for float64 arrays
x = np.array([1.0, 2.0, 3.0])
result1 = add_arrays(x, x)  # Compilation happens here

# Second call: uses cached version
y = np.array([4.0, 5.0, 6.0])
result2 = add_arrays(y, y)  # Fast, no compilation

# Third call: different types, triggers new compilation
z = np.array([1, 2, 3], dtype=np.int32)
result3 = add_arrays(z, z)  # New compilation for int32
```

Benefits:

- Optimal code generation for each type combination
- Amortizes compilation cost over multiple calls
- Automatic optimization without manual type declarations

Considerations:

- First-call overhead for each type combination
- Multiple type combinations increase memory usage
- Consider using explicit signatures for critical paths

4.6.2 Compilation Caching to Disk

- Saves compiled machine code to disk
- Eliminates compilation overhead on subsequent program runs
- Enabled with `cache=True` parameter

Example:

```python
from numba import jit

@jit(nopython=True, cache=True)
def expensive_computation(n):
    result = 0
    for i in range(n):
        result += i * i
    return result

# First run: compiles and caches to disk
# Subsequent runs: loads from cache, no compilation
```

Benefits:

- Faster startup times for subsequent runs
- Especially valuable for large codebases
- Reduces first-call latency in production

Cache location:

- Stored in `__pycache__` directory by default
- Automatically invalidated when source code changes
- Can be cleared manually if needed

Best practices:

- Enable caching for production code
- Use for functions with expensive compilation
- Particularly valuable for large GPU kernels

4.7 Vectorization

Numba provides high-level decorators that automatically parallelize functions across array elements without writing explicit loops or kernels.

4.7.1 @vectorize Decorator

- Converts scalar functions into universal functions (ufuncs)
- Automatically applies the function to each element of arrays
- Supports CPU and GPU targets
- Similar to NumPy's vectorize but much faster

Basic Usage:

```
from numba import vectorize
import numpy as np

@vectorize
def scalar_add(a, b):
    return a + b

# Automatically works on arrays
x = np.array([1, 2, 3, 4, 5])
y = np.array([10, 20, 30, 40, 50])
result = scalar_add(x, y)  # [11, 22, 33, 44, 55]
```

GPU Target:

```
from numba import vectorize
import numpy as np

@vectorize(['float64(float64, float64)'], target='cuda')
```

```python
def gpu_add(a, b):
    return a + b

# Automatically executes on GPU
x = np.random.randn(1000000)
y = np.random.randn(1000000)
result = gpu_add(x, y)
```

Supported targets:

- `'cpu'`: Single-threaded CPU execution
- `'parallel'`: Multi-threaded CPU execution
- `'cuda'`: NVIDIA GPU execution

Benefits:

- Write scalar logic, get parallel execution
- No explicit loop management
- Clean, readable code
- Easy to switch between CPU and GPU

4.7.2 @guvectorize Decorator

- Generalized universal functions (gufuncs)
- Operates on subarrays rather than scalars
- Supports reduction operations and complex array manipulations
- More flexible than `@vectorize`

Example: Moving Average:

```python
from numba import guvectorize
import numpy as np

@guvectorize(['void(float64[:], intp[:], float64[:])'],
             '(n),()->(n)', target='parallel')
def moving_average(data, window, result):
    for i in range(data.shape[0]):
        start = max(0, i - window[0] + 1)
        result[i] = 0.0
        for j in range(start, i + 1):
            result[i] += data[j]
        result[i] /= (i - start + 1)

# Usage
data = np.array([1.0, 2.0, 3.0, 4.0, 5.0])
window = np.array([3])
result = np.empty_like(data)
moving_average(data, window, result)
```

Signature format:

- Function signature: type specification
- Layout signature: input/output dimensions
- `'(n),()->(n)'` means:
  - input: 1D array of length n, scalar (0-dimensional)
  - output: 1D array of length n

Use cases:

- Reduction operations (sum, mean, max over windows)
- Sliding window computations
- Custom array transformations
- Matrix operations on subarrays

4.8 Multi-Core CPU Parallelization

Numba can automatically parallelize code across multiple CPU cores, similar to SMP (Symmetric Multi-Processing) multi-threaded execution. This provides significant speedups on multi-core systems without requiring explicit thread management.

4.8.1 Automatic Parallelization with parallel=True

Numba can automatically identify parallel loops and distribute iterations across CPU cores.

```
from numba import jit
import numpy as np
import time

# Sequential version
@jit(nopython=True)
def sum_squares_sequential(arr):
    """Sequential computation"""
    n = arr.shape[0]
    result = np.zeros(n)
    for i in range(n):
        result[i] = arr[i] ** 2
    return result

# Parallel version - Numba automatically parallelizes the loop
@jit(nopython=True, parallel=True)
def sum_squares_parallel(arr):
    """Parallel computation across CPU cores"""
    n = arr.shape[0]
    result = np.zeros(n)
    for i in range(n):
        result[i] = arr[i] ** 2
    return result

# Benchmark
n = 100_000_000
```

```
data = np.random.randn(n)

# Sequential
start = time.time()
result_seq = sum_squares_sequential(data)
seq_time = time.time() - start

# Parallel
start = time.time()
result_par = sum_squares_parallel(data)
par_time = time.time() - start

print(f"Sequential time: {seq_time:.3f}s")
print(f"Parallel time: {par_time:.3f}s")
print(f"Speedup: {seq_time/par_time:.2f}x")
print(f"Results match: {np.allclose(result_seq, result_par)}")
```

Key Features:

- `parallel=True`: Enables automatic parallelization
- Numba analyzes loops for data dependencies
- Independent iterations are distributed across CPU threads
- Releases the Python Global Interpreter Lock (GIL)
- Works with multiple CPU cores simultaneously

4.8.2 Explicit Parallel Loops with prange

For more control, use `prange` (parallel range) to explicitly mark loops for parallelization.

```
from numba import jit, prange
import numpy as np
import time

@jit(nopython=True, parallel=True)
def parallel_computation(a, b):
    """Explicitly parallel loop using prange"""
    n = a.shape[0]
    result = np.zeros(n)

    # prange explicitly parallelizes this loop
    for i in prange(n):
        # Each iteration runs on different CPU core
        result[i] = np.sqrt(a[i]**2 + b[i]**2)

    return result

# Usage
n = 50_000_000
```

```
a = np.random.randn(n)
b = np.random.randn(n)

# Time the execution
start = time.time()
result = parallel_computation(a, b)
elapsed = time.time() - start

print(f"Time: {elapsed:.3f}s")
print(f"Utilized {np.ceil(n / 1e6 / elapsed):.0f}M operations/second")
```

prange vs range:

- `range`: Sequential execution
- `prange`: Parallel execution across CPU cores
  - similiar to OpenMP's "parallel for"
- Only use `prange` when iterations are independent
- Numba handles thread creation and synchronization

4.8.3 Parallel Reductions

Numba can parallelize reduction operations (sum, mean, etc.) automatically.

```
from numba import jit, prange
import numpy as np
import time

@jit(nopython=True, parallel=True)
def parallel_sum(arr):
    """Parallel sum reduction"""
    total = 0.0
    # Numba parallelizes this reduction automatically
    for i in prange(arr.shape[0]):
        total += arr[i]
    return total

@jit(nopython=True, parallel=True)
def parallel_dot_product(a, b):
    """Parallel dot product"""
    result = 0.0
    for i in prange(a.shape[0]):
        result += a[i] * b[i]
    return result

# Benchmark
n = 100_000_000
arr = np.random.randn(n)
```

```
# Parallel sum
start = time.time()
par_sum = parallel_sum(arr)
par_time = time.time() - start

# NumPy sum (single-threaded for large arrays)
start = time.time()
np_sum = np.sum(arr)
np_time = time.time() - start

print(f"Parallel sum: {par_sum:.6f} ({par_time:.3f}s)")
print(f"NumPy sum: {np_sum:.6f} ({np_time:.3f}s)")
print(f"Speedup: {np_time/par_time:.2f}x")
```

How it works:

- Numba splits the reduction across threads
- Each thread computes partial result
- Results are combined efficiently
- Automatic handling of race conditions

4.8.4 Nested Parallel Loops

Numba can handle nested parallel loops for 2D operations.

Example: Parallel Matrix Operations

```
from numba import jit, prange
import numpy as np
import time

@jit(nopython=True, parallel=True)
def parallel_matrix_op(A, B):
    """Element-wise operation on matrices with nested parallel loops"""
    m, n = A.shape
    C = np.zeros((m, n))

    # Outer loop parallelized
    for i in prange(m):
        for j in range(n):
            C[i, j] = np.sqrt(A[i, j]**2 + B[i, j]**2)

    return C

# Benchmark
m, n = 10000, 10000
A = np.random.randn(m, n).astype(np.float32)
B = np.random.randn(m, n).astype(np.float32)
```

```
start = time.time()
C = parallel_matrix_op(A, B)
elapsed = time.time() - start

print(f"Matrix size: {m}×{n}")
print(f"Time: {elapsed:.3f}s")
print(f"Throughput: {m*n/elapsed/1e6:.1f} Million ops/sec")
```

Best practices:

- Parallelize outer loop, keep inner loop sequential
- Only parallelize one level to avoid overhead
- Ensure sufficient work per thread (avoid too fine-grained parallelism)

Or vectorize the inner loop:

```
# Outer loop parallelized
    for i in prange(m):
        # Inner operation vectorized (NumPy)
        C[i, :] = np.sqrt(A[i, :]**2 + B[i, :]**2)
```

4.8.5 When to Use Multi-Core Parallelization

Good candidates for parallel=True:

- Large arrays (>10^6 elements)
- Independent operations per element
- Compute-intensive operations (not just memory access)
- Nested loops with substantial work

Not beneficial for:

- Small arrays (overhead dominates)
- Sequential dependencies between iterations
- Memory-bound operations (already limited by bandwidth)
- Very simple operations (addition, copying)

Sequential vs Parallel Performance

```
from numba import jit, prange
import numpy as np
import time

def benchmark_sizes():
    """Show when parallel execution helps"""
    sizes = [1000, 10000, 100000, 1000000, 10000000]

    @jit(nopython=True)
    def compute_seq(arr):
        result = np.zeros_like(arr)
        for i in range(arr.shape[0]):
```

```
            result[i] = np.sqrt(arr[i]**2 + arr[i]**3)
        return result

    @jit(nopython=True, parallel=True)
    def compute_par(arr):
        result = np.zeros_like(arr)
        for i in prange(arr.shape[0]):
            result[i] = np.sqrt(arr[i]**2 + arr[i]**3)
        return result

    print(f"{'Size':<12} {'Sequential':<12} {'Parallel':<12} {'Speedup':<12}")
    print("-" * 50)

    for size in sizes:
        arr = np.random.randn(size)

        # Warmup
        _ = compute_seq(arr)
        _ = compute_par(arr)

        # Sequential
        start = time.time()
        _ = compute_seq(arr)
        seq_time = time.time() - start

        # Parallel
        start = time.time()
        _ = compute_par(arr)
        par_time = time.time() - start

        speedup = seq_time / par_time if par_time > 0 else 0

        print(f"{size:<12} {seq_time:<12.6f} {par_time:<12.6f} {speedup:<12.2f}")

benchmark_sizes()
```

4.8.6 Thread Control

You can control the number of threads Numba uses for parallel execution.

```
from numba import jit, prange, set_num_threads, get_num_threads
import numpy as np
import time
import os

# Check default thread count
print(f"Default threads: {get_num_threads()}")
```

```
print(f"CPU cores: {os.cpu_count()}")

@jit(nopython=True, parallel=True)
def parallel_work(arr):
    result = np.zeros_like(arr)
    for i in prange(arr.shape[0]):
        result[i] = np.sqrt(arr[i]**2)
    return result

arr = np.random.randn(50_000_000)

# Test with different thread counts
for nthreads in [1, 2, 4, 8]:
    set_num_threads(nthreads)

    start = time.time()
    result = parallel_work(arr)
    elapsed = time.time() - start

    print(f"Threads: {nthreads}, Time: {elapsed:.3f}s")

# Environment variable alternative
# export NUMBA_NUM_THREADS=4
```

Thread control methods:

- `set_num_threads(n)`: Set at runtime
- `NUMBA_NUM_THREADS` environment variable
- Default: all available CPU cores
- Optimal: usually equals physical CPU cores

4.8.7 Parallel NumPy Operations

Numba automatically parallelizes many NumPy operations when `parallel=True`.

```
from numba import jit
import numpy as np
import time

@jit(nopython=True, parallel=True)
def parallel_numpy_ops(a, b, c):
    """NumPy operations automatically parallelized"""
    # These operations are automatically parallel
    result1 = a + b + c
    result2 = np.sqrt(result1)
    result3 = np.sin(result2) + np.cos(result2)
    return result3
```

```
n = 50_000_000
a = np.random.randn(n)
b = np.random.randn(n)
c = np.random.randn(n)

# Warmup
_ = parallel_numpy_ops(a, b, c)

start = time.time()
result = parallel_numpy_ops(a, b, c)
elapsed = time.time() - start

print(f"Time with automatic parallelization: {elapsed:.3f}s")
```

Automatically parallelized NumPy operations:

- Element-wise arithmetic (`+`, `-`, `*`, `/`)
- Math functions (`np.sin`, `np.cos`, `np.sqrt`, etc.)
- Array creation (`np.zeros`, `np.ones`)
- Reductions with explicit axis
- Boolean operations and comparisons

4.8.8 Performance Considerations

Factors affecting speedup:

- Number of CPU cores
- Operation complexity (more complex = better speedup)
- Array size (larger = better speedup)
- Memory bandwidth (can become bottleneck)
- Thread synchronization overhead

4.8.9 Key Takeaways

Advantages of Numba's Multi-Core Parallelization:

- Simple: Just add `parallel=True` or use `prange`
- Automatic: Numba identifies parallelizable loops
- Efficient: Releases GIL, uses native threads
- Portable: Same code works on different CPU counts
- No manual thread management required

Best Practices:

- Use `parallel=True` for large datasets
- Use `prange` when you need explicit control
- Ensure loop iterations are independent
- Profile to verify speedup (overhead for small arrays)
- Consider memory bandwidth limitations
- Test with different thread counts to find optimum

4.9 CPU and GPU Code Adaptation

Numba enables Python developers to write GPU code without learning CUDA C/C++. While CPU and GPU versions require different code (due to different execution models), the core algorithm logic remains in Python, making the adaptation straightforward.

4.9.1 Adapting Algorithms Between CPU and GPU

What changes: - CPU uses loop-based execution (`@jit`) - GPU uses thread-based execution (`@cuda.jit` with thread indexing)

What stays the same: - Python syntax - Core mathematical operations - Algorithm logic

CPU Version with @jit:

```
from numba import jit
import numpy as np

@jit(nopython=True)
def matrix_multiply_cpu(A, B, C):
    # Loop-based: iterate sequentially or in parallel
    for i in range(A.shape[0]):
        for j in range(B.shape[1]):
            for k in range(A.shape[1]):
                C[i, j] += A[i, k] * B[k, j]
```

GPU Version with @cuda.jit:

```
from numba import cuda

@cuda.jit
def matrix_multiply_gpu(A, B, C):
    # Thread-based: each thread handles one output element
    i, j = cuda.grid(2)  # Get thread position
    if i < C.shape[0] and j < C.shape[1]:
        tmp = 0.0
        for k in range(A.shape[1]):
            tmp += A[i, k] * B[k, j]  # Same computation!
        C[i, j] = tmp
```

Key difference: Loop iteration (`for i in range(...)`) becomes thread indexing (`i, j = cuda.grid(2)`), but the mathematical operations `A[i, k] * B[k, j]` remain identical.

4.9.2 True Portability with @vectorize

For **genuine write-once, run-anywhere code**, use `@vectorize` where you can switch hardware targets by simply changing one parameter:

```
from numba import vectorize
import numpy as np
```

```python
# Define the function once
@vectorize(['float64(float64, float64)'], target='cpu')  # Single-threaded CPU
# @vectorize(['float64(float64, float64)'], target='parallel')  # Multi-core CPU
# @vectorize(['float64(float64, float64)'], target='cuda')  # GPU
def compute(a, b):
    return a * a + b * b  # Same code for all targets!

# Usage is identical regardless of target
data = np.random.randn(1_000_000)
result = compute(data, data)
```

This is truly portable - the function body never changes, only the `target` parameter determines where it runs.

4.9.3 Benefits of This Approach

Compared to writing CUDA C/C++:

- Stay in Python ecosystem - no separate CUDA C compilation
- Use familiar NumPy array operations
- Easier debugging with Python tools
- Faster prototyping and iteration

Development workflow:

1. **Develop on CPU** with `@jit` for rapid prototyping and debugging
2. **Test and optimize** with small datasets using Python tools
3. **Adapt to GPU** by converting to `@cuda.jit` (requires thread indexing changes)
4. **Or use `@vectorize`** for element-wise operations (truly portable, just change `target`)
5. **Benchmark and profile** on target hardware

Key advantages:

- **Lower barrier to entry**: Python instead of CUDA C/C++
- **Systematic translation**: CPU loops → GPU thread indexing follows predictable patterns
- **True portability with `@vectorize`**: Same function code across CPU/GPU targets
- **Incremental adoption**: Start with CPU, move to GPU when needed

4.10 NVIDIA GPU Support with Numba CUDA

Numba's CUDA support is one of its most powerful features, allowing Python developers to harness GPU computing power without learning CUDA C/C++. This democratizes GPU programming and significantly reduces the barrier to entry for parallel computing.

4.10.1 Key Features

Direct CUDA Kernel Programming:

- Use `@cuda.jit` decorator to write CUDA kernels in Python
- Compiles to actual GPU code
- Access to CUDA-specific features:

  - Thread indexing
  - Shared memory
  - Synchronization primitives

Memory Management:

- `cuda.to_device()`: Transfer host data to GPU
- `cuda.device_array()`: Allocate uninitialized GPU memory
- `.copy_to_host()`: Transfer GPU data back to host

Automatic Parallelization:

- `@vectorize` and `@guvectorize` decorators
- Automatically parallelize array operations across GPU threads
- No explicit kernel code required

Multi-GPU Support:

- Compatible with multi-GPU systems
- Select specific devices
- Coordinate computation across multiple GPUs

4.10.2 Performance Characteristics

GPU acceleration with Numba is most effective when:

- Operations are data-parallel (same operation on many data elements)
- Datasets are large enough to amortize transfer costs (typically >10^6 elements)
- Computation is arithmetic-intensive relative to memory access
- Algorithm can leverage GPU shared memory and coalesced memory access

Typical speedups range from 10-100x for well-suited problems, though results vary based on the algorithm and GPU hardware.

4.10.3 Hardware Requirements

- NVIDIA GPU with CUDA Compute Capability 3.0 or higher
- CUDA Toolkit installed (drivers and runtime libraries)
- Compatible with most modern NVIDIA GPUs (GeForce, Quadro, Tesla, A100, H100)

4.11 Examples

4.11.1 Vector Addition (GPU Kernel Basics)

Vector addition demonstrates the fundamental concepts of GPU kernel programming with Numba.

Key Concepts:

- Defining CUDA kernels with `@cuda.jit`
- GPU memory management
- Thread indexing with `cuda.grid()`
- Kernel launch configuration (blocks and threads)

Implementation:

```python
from numba import cuda
import numpy as np
import math

@cuda.jit
def vector_add_kernel(a, b, c):
    """
    GPU kernel for element-wise vector addition
    Each thread computes one element of the output
    """
    # Calculate global thread index
    idx = cuda.grid(1)

    # Boundary check
    if idx < c.size:
        c[idx] = a[idx] + b[idx]

def vector_add_gpu(a, b):
    """
    Wrapper function to launch GPU kernel
    """
    # Allocate device memory
    d_a = cuda.to_device(a)
    d_b = cuda.to_device(b)
    d_c = cuda.device_array_like(a)

    # Configure kernel launch
    threads_per_block = 256
    blocks_per_grid = math.ceil(a.size / threads_per_block)

    # Launch kernel
    vector_add_kernel[blocks_per_grid, threads_per_block](d_a, d_b, d_c)

    # Copy result back to host
    c = d_c.copy_to_host()
    return c

# Usage
n = 1_000_000
a = np.random.randn(n)
b = np.random.randn(n)
c = vector_add_gpu(a, b)
```

Key Points:

- `cuda.grid(1)`: Computes the global thread index in a 1D grid
- `cuda.to_device()`: Transfers host memory to GPU

- `cuda.device_array_like()`: Allocates GPU memory with same shape/dtype
- `[blocks, threads]`: Kernel launch configuration
- Boundary checking prevents out-of-bounds access

Performance Considerations:

- Thread block size (256) balances occupancy and resource usage
- Grid size ensures enough threads to cover all array elements
- Memory transfers are often the bottleneck for simple operations
- Best for large arrays where computation dominates transfer time

See `examples/vecadd-numba.py` for complete implementation.

4.11.2 Monte Carlo Pi Estimation (CPU/GPU Portability)

Problem:

Estimate by randomly sampling points in a unit square and counting how many fall inside a quarter circle.

Algorithm:

1. Generate random points (x, y) in $[0, 1] \times [0, 1]$
2. Check if $x^2 + y^2 \leq 1$ (inside quarter circle)
3. $\pi \approx 4 \times$ (points inside circle) / (total points)

```
from numba import jit
import numpy as np
import time

@jit(nopython=True)
def monte_carlo_pi_cpu(n_samples):
    """
    Estimate pi using Monte Carlo method on CPU
    """
    count_inside = 0

    for i in range(n_samples):
        x = np.random.random()
        y = np.random.random()

        # Check if point is inside quarter circle
        if x*x + y*y <= 1.0:
            count_inside += 1

    return 4.0 * count_inside / n_samples

# Usage
n = 10_000_000
```

```
start = time.time()
pi_estimate = monte_carlo_pi_cpu(n)
cpu_time = time.time() - start

print(f"CPU Estimate:    {pi_estimate:.6f}")
print(f"CPU Time: {cpu_time:.3f} seconds")
```

4.11.2.1 CPU Version with @jit

```
from numba import cuda
import numpy as np
import math
import time

@cuda.jit
def monte_carlo_pi_kernel(n_samples, counts):
    """
    GPU kernel for Monte Carlo pi estimation
    Each thread generates samples and counts hits
    """
    idx = cuda.grid(1)

    # Create thread-local RNG state
    rng_states = cuda.random.create_xoroshiro128p_states(
        cuda.gridsize(1), seed=idx
    )

    # Each thread processes multiple samples
    samples_per_thread = (n_samples + cuda.gridsize(1) - 1) // cuda.gridsize(1)
    thread_count = 0

    for i in range(samples_per_thread):
        # Generate random point
        x = cuda.random.xoroshiro128p_uniform_float32(rng_states, idx)
        y = cuda.random.xoroshiro128p_uniform_float32(rng_states, idx)

        # Check if inside circle
        if x*x + y*y <= 1.0:
            thread_count += 1

    # Store thread result
    counts[idx] = thread_count

def monte_carlo_pi_gpu(n_samples):
    """
    Wrapper for GPU Monte Carlo pi estimation
```

```
    """
    # Launch configuration
    threads_per_block = 256
    blocks_per_grid = 512
    total_threads = threads_per_block * blocks_per_grid

    # Allocate device memory for counts
    d_counts = cuda.device_array(total_threads, dtype=np.int32)

    # Launch kernel
    monte_carlo_pi_kernel[blocks_per_grid, threads_per_block](
        n_samples, d_counts
    )

    # Copy results and sum
    counts = d_counts.copy_to_host()
    total_inside = counts.sum()

    return 4.0 * total_inside / n_samples

# Usage
n = 100_000_000  # More samples for GPU
start = time.time()
pi_estimate = monte_carlo_pi_gpu(n)
gpu_time = time.time() - start

print(f"GPU Estimate:    {pi_estimate:.6f}")
print(f"GPU Time: {gpu_time:.3f} seconds")
```

4.11.2.2 GPU Version with @cuda.jit

4.11.2.3 Simplified GPU Version with @vectorize

For simpler Monte Carlo simulations, `@vectorize` provides an easier approach:

```
from numba import vectorize, cuda
import numpy as np

@vectorize(['int32(float32, float32)'], target='cuda')
def check_inside_circle(x, y):
    """
    Check if point is inside unit circle
    Automatically parallelized across GPU threads
    """
    return 1 if (x*x + y*y <= 1.0) else 0

def monte_carlo_pi_vectorized(n_samples):
    """
```

```
    Vectorized GPU implementation
    """
    # Generate random points on CPU (or use CuPy for GPU generation)
    x = np.random.random(n_samples).astype(np.float32)
    y = np.random.random(n_samples).astype(np.float32)

    # GPU automatically processes all points in parallel
    inside = check_inside_circle(x, y)

    return 4.0 * inside.sum() / n_samples

# Usage
n = 50_000_000
pi_estimate = monte_carlo_pi_vectorized(n)
print(f"Vectorized GPU Estimate:    {pi_estimate:.6f}")
```

Observations

1. **Custom Algorithm Implementation:**
   - Not available as a library function
   - Requires explicit loops and logic
   - Perfect for Numba's JIT compilation
2. **CPU/GPU Portability:**
   - Same basic algorithm structure for both targets
   - Easy to develop on CPU, deploy on GPU
   - Can run on systems without GPU
3. **True Parallel Computing:**
   - Each sample is independent
   - Naturally data-parallel
   - Scales well to many GPU threads

Expected Performance Characteristics:

- **Pure Python**: Very slow for large sample counts (baseline)
- **CPU @jit**: Typically 50-100x faster than pure Python
- **GPU @cuda.jit**: Can handle 100M+ samples efficiently, often 1000x+ faster than pure Python
- **GPU @vectorize**: Simpler to implement, performance between CPU and explicit GPU kernels

Key Takeaways:

- Same algorithm logic across CPU and GPU
- GPU shines for embarrassingly parallel problems
- `@vectorize` provides easier GPU programming for simple patterns
- `@cuda.jit` gives more control for complex kernels

4.12 Installation

Basic Installation:

```
python -m venv .venv
source .venv/bin/activate
pip install numba
```

For GPU Support:

```
# Ensure CUDA Toolkit is installed first
pip install numba

# Verify CUDA support
python -c "from numba import cuda; print(cuda.is_available())"
```

5 CuPy

CuPy is an array library designed for GPU-accelerated computing with Python. It provides a seamless NumPy/SciPy-compatible API for performing operations on NVIDIA CUDA (or AMD ROCm, experimentally) platforms.

5.1 Overview

CuPy is built on the philosophy of minimal code changes for maximum acceleration. By providing a drop-in replacement for NumPy arrays that transparently execute on the GPU, CuPy allows developers to leverage GPU computing power without learning CUDA programming or significantly refactoring existing code.

Key advantages:

- Nearly identical API to NumPy/SciPy - often just changing `numpy` to `cupy` is sufficient
- Automatic GPU execution of array operations
- Access to optimized CUDA libraries (cuBLAS, cuFFT, cuSPARSE, cuDNN)
- Support for custom kernels when needed
- Excellent interoperability with other GPU libraries

5.2 History

- **2015**: Initial release as an open-source project
- **2017**: Added comprehensive SciPy compatibility
- **2018**: Became part of the Chainer deep learning framework ecosystem
- **2019**: Standalone development accelerated; expanded beyond deep learning use cases
- **2020-Present**: Continuous expansion of NumPy/SciPy coverage, improved performance, and broader hardware support

CuPy is maintained by Preferred Networks and the community, with widespread adoption in scientific computing, data science, and machine learning. It's now a NumFOCUS affiliated project.

5.3 What CuPy Does

Core Functionality:

- Drop-in NumPy/SciPy replacement for GPU acceleration
- Minimal code changes required (often just `import cupy as cp`)
- Complete API coverage for most NumPy/SciPy operations
- GPU acceleration without deep GPU programming knowledge

Array Management:

- `cupy.ndarray` class replaces `numpy.ndarray`
- Arrays allocated directly on GPU device memory
- Automatic device management and synchronization
- Support for multi-GPU systems

GPU Execution:

- Automatic GPU execution of NumPy operations
- Element-wise operations as Universal Functions (ufuncs)
- Leverages optimized CUDA libraries:
  - cuBLAS: Linear algebra operations
  - cuFFT: Fast Fourier transforms
  - cuSPARSE: Sparse matrix operations
  - cuDNN: Deep learning primitives
  - Thrust, CUB, cuTENSOR: Additional optimized operations

Custom Kernel Support:

- ElementwiseKernel: Custom element-wise operations
- ReductionKernel: Custom reduction operations
- RawKernel: Import existing CUDA C/C++ code
- Kernel Fusion: Automatically fuse multiple operations into single kernel

Interoperability:

- CUDA Array Interface: Zero-copy exchange with CUDA libraries
- DLPack: Data exchange with PyTorch, TensorFlow, JAX
- NumPy compatibility: Seamless conversion between CPU and GPU
- Numba integration: Use CuPy arrays in Numba kernels

5.4 Memory Management

CuPy provides sophisticated memory management features to optimize GPU memory usage and performance.

5.4.1 Memory Pools

CuPy uses memory pools by default to reduce the overhead of GPU memory allocation and deallocation.

Example: Understanding Memory Pools

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

# CuPy uses a memory pool by default
mempool = cp.get_default_memory_pool()        # GPU memory

# Check memory usage before allocation
print(f"Used memory: {mempool.used_bytes() / 1e9:.2f} GB")
print(f"Total memory: {mempool.total_bytes() / 1e9:.2f} GB")

# Allocate arrays - memory comes from pool
a = cp.random.randn(10000, 10000)  # ~800 MB
b = cp.random.randn(10000, 10000)  # ~800 MB

print(f"After allocation - Used: {mempool.used_bytes() / 1e9:.2f} GB")

# Delete arrays - memory returned to pool, not freed
del a, b

print(f"After deletion - Used: {mempool.used_bytes() / 1e9:.2f} GB")
print(f"Pool still holds: {mempool.total_bytes() / 1e9:.2f} GB")

# Free all unused memory blocks
mempool.free_all_blocks()

print(f"After free_all_blocks - Total: {mempool.total_bytes() / 1e9:.2f} GB")
```

Key Points:

- Memory pool reduces allocation overhead
- Deleted arrays return memory to pool, not to GPU
- Use `free_all_blocks()` to actually free GPU memory
- Useful for managing memory in long-running processes

Custom Memory Pool:

```python
import cupy as cp

# Create custom memory pool using Unified Memory (managed memory)
pool = cp.cuda.MemoryPool(cp.cuda.malloc_managed)
cp.cuda.set_allocator(pool.malloc)

# Or disable memory pool entirely
cp.cuda.set_allocator(cp.cuda.malloc)
```

Default CuPy Memory Pool (`cp.cuda.malloc`):

- GPU-only device memory
- Best performance for pure GPU computation

- Requires explicit CPU GPU transfers
- Recommended for most use cases

Managed Memory Pool (`cp.cuda.malloc_managed`):

- Unified Memory accessible from CPU and GPU
- Automatic data migration
- Simpler programming model
- Potential performance overhead
- Use for CPU-GPU interop or prototyping

5.5 Custom Kernels

While CuPy provides many built-in operations, you can write custom CUDA kernels for specialized computations.

5.5.1 ElementwiseKernel

For element-wise operations not available in CuPy.

Example: Custom Sigmoid Activation

```
import cupy as cp

# Define custom elementwise kernel
sigmoid_kernel = cp.ElementwiseKernel(
    'float32 x',           # Input type and name
    'float32 y',           # Output type and name
    'y = 1.0f / (1.0f + expf(-x))',  # Operation (CUDA C syntax)
    'sigmoid'              # Kernel name
)

# Use the kernel
x = cp.random.randn(1000000, dtype=cp.float32)
y = sigmoid_kernel(x)

# Compare with pure CuPy
y_cupy = 1.0 / (1.0 + cp.exp(-x))

print(f"Results match: {cp.allclose(y, y_cupy)}")
```

Benefits:

- Fuses operations into single kernel (faster)
- CUDA C syntax for complex operations
- Automatic memory management

5.5.2 ReductionKernel

For custom reduction operations (sum, max, etc.).

Example: Weighted Sum

```
import cupy as cp

# Custom weighted sum reduction
weighted_sum_kernel = cp.ReductionKernel(
    'float32 x, float32 w',    # Input arguments
    'float32 y',                # Output type
    'x * w',                    # Map operation (per element)
    'a + b',                    # Reduce operation (combine results)
    'y = a',                    # Post-reduction
    '0',                        # Identity value
    'weighted_sum'              # Kernel name
)

# Use the kernel
data = cp.random.randn(1000000, dtype=cp.float32)
weights = cp.random.rand(1000000, dtype=cp.float32)

result = weighted_sum_kernel(data, weights)

# Verify against pure CuPy
result_cupy = cp.sum(data * weights)
print(f"Weighted sum: {result}")
print(f"Results match: {cp.allclose(result, result_cupy)}")
```

5.5.3 RawKernel

For importing existing CUDA C/C++ code.

Example: Custom Matrix Add with RawKernel

```
import cupy as cp

# CUDA C code as string
cuda_code = '''
extern "C" __global__
void matrix_add(const float* a, const float* b, float* c, int size) {
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    if (idx < size) {
        c[idx] = a[idx] + b[idx];
    }
}
'''

# Compile and load kernel
matrix_add_kernel = cp.RawKernel(cuda_code, 'matrix_add')

# Prepare data
```

```
n = 1000000
a = cp.random.randn(n, dtype=cp.float32)
b = cp.random.randn(n, dtype=cp.float32)
c = cp.empty_like(a)

# Launch kernel
threads_per_block = 256
blocks = (n + threads_per_block - 1) // threads_per_block
matrix_add_kernel((blocks,), (threads_per_block,), (a, b, c, n))

# Verify
print(f"Results match: {cp.allclose(c, a + b)}")
```

Use cases:

- Integrating existing CUDA code
- Complex algorithms needing full CUDA control
- Performance-critical custom kernels

5.6 Kernel Fusion

CuPy automatically fuses multiple operations into a single kernel for better performance.

Example: Automatic Kernel Fusion

```
import cupy as cp
import time

# Create large arrays
x = cp.random.randn(10000000, dtype=cp.float32)
y = cp.random.randn(10000000, dtype=cp.float32)

# Multiple operations - CuPy will fuse them
with cp.fuse():
    # All operations fused into single kernel
    result = (x * 2.0 + y * 3.0) / (x + y + 1.0)
    cp.cuda.Stream.null.synchronize()

# Without fusion (for comparison): launches multiple kernels
result_unfused = (x * 2.0 + y * 3.0) / (x + y + 1.0)

print(f"Results match: {cp.allclose(result, result_unfused)}")

# Demonstrate fusion benefit
def with_fusion():
    with cp.fuse():
        z = (x * 2.0 + y * 3.0) / (x + y + 1.0)
        return z
```

```
def without_fusion():
    z = (x * 2.0 + y * 3.0) / (x + y + 1.0)
    return z

# Benchmark (warm up first)
_ = with_fusion()
_ = without_fusion()

# Measure
start = time.time()
for _ in range(100):
    _ = with_fusion()
cp.cuda.Stream.null.synchronize()
fused_time = time.time() - start

start = time.time()
for _ in range(100):
    _ = without_fusion()
cp.cuda.Stream.null.synchronize()
unfused_time = time.time() - start

print(f"With fusion: {fused_time:.3f}s")
print(f"Without fusion: {unfused_time:.3f}s")
print(f"Speedup: {unfused_time/fused_time:.2f}x")
```

Benefits:

- Reduces kernel launch overhead
- Minimizes memory transfers
- Automatic optimization

5.7 Multi-GPU Support

CuPy makes it easy to use multiple GPUs for parallel computation.

Example: Computing on Multiple GPUs

```
import cupy as cp
import numpy as np

# Check available GPUs
n_gpus = cp.cuda.runtime.getDeviceCount()
print(f"Available GPUs: {n_gpus}")

def compute_on_device(device_id, data):
    """Compute on specific GPU"""
    with cp.cuda.Device(device_id):
        # This computation happens on the specified GPU
```

```
        gpu_data = cp.asarray(data)
        result = cp.sum(gpu_data ** 2)
        return result.get()  # Transfer back to CPU

# Split work across GPUs
data = np.random.randn(10000000)
chunk_size = len(data) // n_gpus

results = []
for i in range(n_gpus):
    start = i * chunk_size
    end = start + chunk_size if i < n_gpus - 1 else len(data)
    chunk = data[start:end]

    result = compute_on_device(i, chunk)
    results.append(result)

total = sum(results)
print(f"Total sum of squares: {total}")

# Verify against single GPU
with cp.cuda.Device(0):
    gpu_data = cp.asarray(data)
    expected = cp.sum(gpu_data ** 2).get()
    print(f"Results match: {np.isclose(total, expected)}")
```

Key Points:

- `cp.cuda.Device(id)` context manager selects GPU
- Each GPU has its own memory space
- Results need explicit transfer between GPUs or CPU

5.8 No Multi-Core CPU Support

CuPy is fundamentally a GPU-accelerated library and does NOT have native multi-core CPU parallelization capabilities.

5.8.1 CuPy's Execution Model

GPU-Only Design:

- CuPy operations execute on NVIDIA GPUs via CUDA
- No built-in support for multi-threaded CPU execution
- Arrays reside in GPU memory, operations run on GPU cores
- Designed for GPU parallelism, not CPU multi-threading

CPU Fallback via NumPy:

When GPU is unavailable or for specific workflows, you can use NumPy as a CPU fallback, but this

is just standard NumPy behavior:

```
import numpy as np
import cupy as cp

# Try GPU first, fall back to NumPy for CPU
try:
    xp = cp  # Use CuPy if available
    x = xp.random.randn(1000000)
    print("Using GPU acceleration")
except:
    xp = np  # Fall back to NumPy
    x = xp.random.randn(1000000)
    print("Using CPU (NumPy)")

# Code works with either
result = xp.sum(x ** 2)
```

NumPy's CPU Parallelization:

- NumPy itself is mostly single-threaded for element-wise operations
- Some operations (matrix multiplication via BLAS) can be multi-threaded
- This depends on the underlying BLAS library (OpenBLAS, MKL, etc.)
- Not under CuPy's control - it's just NumPy's native behavior

5.8.2 When to Use What

Use CuPy when:

- You have access to NVIDIA GPU
- Your data is large enough to benefit from GPU parallelism
- You want to accelerate NumPy code with minimal changes
- GPU memory is sufficient for your dataset

Use Numba for CPU multi-core when:

- No GPU available but you have multi-core CPU
- Custom algorithms with loops that need CPU parallelization
- You want same code to run on both CPU (multi-threaded) and GPU
- Fine-grained control over parallelization strategy

Use both together:

Many workflows combine CuPy and Numba:

```
import cupy as cp
from numba import cuda

# CuPy for high-level operations
data = cp.random.randn(1000000, dtype=cp.float32)
result = cp.fft.fft(data)  # CuPy FFT on GPU
```

```
# Numba CUDA kernel for custom operation on CuPy array
@cuda.jit
def custom_kernel(arr):
    idx = cuda.grid(1)
    if idx < arr.size:
        arr[idx] = arr[idx] * 2.0

threads = 256
blocks = (data.size + threads - 1) // threads
custom_kernel[blocks, threads](result)
```

5.8.3 Key Takeaways

CuPy's parallelization:

- GPU-only: thousands of GPU threads, not CPU cores
- No native multi-core CPU support
- Falls back to NumPy (mostly single-threaded) on CPU
- Designed exclusively for GPU acceleration

Bottom line: CuPy is a GPU acceleration library, not a multi-core CPU parallelization library. If you need multi-core CPU parallelism for array operations, use Numba or other CPU-focused tools.

5.9 Interoperability

CuPy seamlessly exchanges data with other GPU libraries through standard protocols.

5.9.1 DLPack: Zero-Copy Exchange

Example: CuPy-PyTorch

```
import cupy as cp
import torch

# CuPy to PyTorch (zero-copy)
cupy_array = cp.random.randn(1000, 1000, dtype=cp.float32)
print(f"CuPy array on GPU: {cupy_array.device}")

# Convert to PyTorch tensor (zero-copy via DLPack)
torch_tensor = torch.utils.dlpack.from_dlpack(cupy_array.toDlpack())
print(f"PyTorch tensor on: {torch_tensor.device}")

# Modify PyTorch tensor - affects CuPy array (same memory)
torch_tensor[0, 0] = 999.0
print(f"CuPy array modified: {cupy_array[0, 0]}")  # Shows 999.0

# PyTorch to CuPy (zero-copy)
torch_tensor = torch.randn(500, 500, device='cuda')
cupy_array = cp.from_dlpack(torch_tensor)
```

```
print(f"Zero-copy: same memory? {cupy_array.data.ptr == torch_tensor.data_ptr()}")
```

5.9.2 CUDA Array Interface

Example: CuPy-Numba

```
import cupy as cp
from numba import cuda

# CuPy array
cupy_array = cp.arange(1000000, dtype=cp.float32)

# Use in Numba kernel via CUDA Array Interface
@cuda.jit
def multiply_kernel(arr):
    idx = cuda.grid(1)
    if idx < arr.size:
        arr[idx] *= 2.0

# Launch Numba kernel on CuPy array (zero-copy)
threads = 256
blocks = (cupy_array.size + threads - 1) // threads
multiply_kernel[blocks, threads](cupy_array)

print(f"First elements: {cupy_array[:5]}")  # [0, 2, 4, 6, 8]
```

Benefits:

- Zero-copy data sharing
- No format conversion overhead
- Seamless integration across libraries

5.10 Examples

5.10.1 Example 1: Vector Addition (NumPy Drop-in Replacement)

Vector addition demonstrates CuPy's core philosophy: minimal code changes for maximum GPU acceleration.

Key Concepts:

- Drop-in replacement for NumPy
- Automatic GPU parallelization
- Memory transfer between CPU and GPU
- Performance comparison with NumPy

Implementation:

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

```python
import time

# Vector size
n = 50_000_000

# NumPy (CPU) version
print("=" * 50)
print("NumPy (CPU) Version")
print("=" * 50)

# Create arrays on CPU
a_cpu = np.random.randn(n).astype(np.float32)
b_cpu = np.random.randn(n).astype(np.float32)

# Time CPU execution
start = time.time()
c_cpu = a_cpu + b_cpu
cpu_time = time.time() - start

print(f"Array size: {n:,} elements ({n*4/1e6:.1f} MB per array)")
print(f"CPU time: {cpu_time:.4f} seconds")
print(f"First 5 elements: {c_cpu[:5]}")

# CuPy (GPU) version
print("\n" + "=" * 50)
print("CuPy (GPU) Version")
print("=" * 50)

# Create arrays on GPU (just change np to cp!)
a_gpu = cp.random.randn(n).astype(cp.float32)
b_gpu = cp.random.randn(n).astype(cp.float32)

# Warm-up (first call includes compilation overhead)
_ = a_gpu + b_gpu
cp.cuda.Stream.null.synchronize()

# Time GPU execution
start = time.time()
c_gpu = a_gpu + b_gpu
cp.cuda.Stream.null.synchronize()  # Wait for GPU to finish
gpu_time = time.time() - start

print(f"Array size: {n:,} elements ({n*4/1e6:.1f} MB per array)")
print(f"GPU time: {gpu_time:.4f} seconds")
print(f"First 5 elements: {cp.asnumpy(c_gpu[:5])}")

# Speedup calculation
```

```
print("\n" + "=" * 50)
print("Performance Comparison")
print("=" * 50)
print(f"CPU time: {cpu_time:.4f}s")
print(f"GPU time: {gpu_time:.4f}s")
print(f"Speedup: {cpu_time/gpu_time:.1f}x faster")

# Including memory transfer overhead
print("\n" + "=" * 50)
print("Including Memory Transfer Overhead")
print("=" * 50)

# Start from NumPy arrays, transfer to GPU, compute, transfer back
start = time.time()
a_transfer = cp.asarray(a_cpu)  # CPU -> GPU
b_transfer = cp.asarray(b_cpu)  # CPU -> GPU
c_transfer = a_transfer + b_transfer
result = cp.asnumpy(c_transfer)  # GPU -> CPU
cp.cuda.Stream.null.synchronize()
total_time = time.time() - start

print(f"Total time (with transfers): {total_time:.4f}s")
print(f"Pure GPU compute: {gpu_time:.4f}s")
print(f"Transfer overhead: {total_time - gpu_time:.4f}s")
print(f"Speedup (with transfers): {cpu_time/total_time:.1f}x")

# Verify correctness
print("\n" + "=" * 50)
print("Correctness Check")
print("=" * 50)
# Compare small portion (avoid memory)
test_size = 1000
a_test = np.random.randn(test_size).astype(np.float32)
b_test = np.random.randn(test_size).astype(np.float32)

c_numpy = a_test + b_test
c_cupy = cp.asnumpy(cp.asarray(a_test) + cp.asarray(b_test))

print(f"Results match: {np.allclose(c_numpy, c_cupy)}")
print(f"Max difference: {np.max(np.abs(c_numpy - c_cupy))}")
```

Key Points:

- **Minimal code change**: Just replace `numpy` with `cupy`
- **Automatic GPU execution**: The `+` operator automatically triggers GPU kernel
- **Memory transfers**: Use `cp.asarray()` (CPU→GPU) and `cp.asnumpy()` (GPU→CPU)
- **Synchronization**: `synchronize()` ensures GPU finishes before timing

- **Transfer overhead**: For small operations, transfer time can dominate

When CuPy Shines:

- Large arrays (transfer cost amortized)
- Multiple operations on same GPU data
- Already have data on GPU
- Iterative algorithms keeping data on GPU

Best Practice:

```
# Good: Keep data on GPU for multiple operations
a_gpu = cp.asarray(a_cpu)  # Transfer once
b_gpu = cp.asarray(b_cpu)

# Many operations without transfer
c_gpu = a_gpu + b_gpu
d_gpu = c_gpu * 2.0
e_gpu = cp.sin(d_gpu)

result = cp.asnumpy(e_gpu)  # Transfer once at end
```

5.10.2 Example 2: Matrix Multiplication (cuBLAS Performance)

Large-scale matrix multiplication is fundamental to scientific computing, appearing in linear algebra, deep learning, physics simulations, and data analysis. CuPy makes it trivial to accelerate this operation.

- **One-line change**: `numpy` → `cupy`
- **cuBLAS library**: Uses NVIDIA's highly optimized BLAS implementation
- **Perfect for CuPy**: Standard operation where libraries beat custom kernels

Implementation:

```
import numpy as np
import cupy as cp
import time

def benchmark_matmul(size, dtype=np.float32):
    """Benchmark matrix multiplication on CPU vs GPU"""

    print(f"\n{'='*60}")
    print(f"Matrix Multiplication: {size}x{size} ({dtype.__name__})")
    print(f"{'='*60}")

    # ------------ NumPy (CPU) version --------------
    print("\nNumPy (CPU):")
    print("-" * 40)

    A_cpu = np.random.randn(size, size).astype(dtype)
    B_cpu = np.random.randn(size, size).astype(dtype)
```

```
# Warm-up
_ = A_cpu @ B_cpu

# Benchmark
start = time.time()
C_cpu = A_cpu @ B_cpu
cpu_time = time.time() - start

memory_mb = (size * size * 4) / 1e6  # 4 bytes per float32
flops = 2 * size**3  # Matrix multiply operations
cpu_gflops = flops / cpu_time / 1e9

print(f"Time: {cpu_time:.4f} seconds")
print(f"Memory per matrix: {memory_mb:.1f} MB")
print(f"Performance: {cpu_gflops:.1f} GFLOPS")

# ---------- CuPy (GPU) version ---------------
print("\nCuPy (GPU):")
print("-" * 40)

A_gpu = cp.random.randn(size, size).astype(dtype)
B_gpu = cp.random.randn(size, size).astype(dtype)

# Warm-up (includes cuBLAS initialization)
_ = A_gpu @ B_gpu
cp.cuda.Stream.null.synchronize()

# Benchmark
start = time.time()
C_gpu = A_gpu @ B_gpu
cp.cuda.Stream.null.synchronize()
gpu_time = time.time() - start

gpu_gflops = flops / gpu_time / 1e9

print(f"Time: {gpu_time:.4f} seconds")
print(f"Memory per matrix: {memory_mb:.1f} MB")
print(f"Performance: {gpu_gflops:.1f} GFLOPS")

# Comparison
print("\n" + "=" * 60)
print("Performance Summary")
print("=" * 60)
print(f"CPU: {cpu_time:.4f}s ({cpu_gflops:.1f} GFLOPS)")
print(f"GPU: {gpu_time:.4f}s ({gpu_gflops:.1f} GFLOPS)")
print(f"Speedup: {cpu_time/gpu_time:.1f}x")
```

```
    print(f"GPU is {gpu_gflops/cpu_gflops:.1f}x more efficient")

    # Verify correctness (small sample)
    if size <= 1000:
        A_test = A_cpu[:100, :100]
        B_test = B_cpu[:100, :100]
        C_numpy = A_test @ B_test
        C_cupy = cp.asnumpy(cp.asarray(A_test) @ cp.asarray(B_test))
        print(f"\nCorrectness: {np.allclose(C_numpy, C_cupy, rtol=1e-5)}")

# Run benchmarks for different sizes
print("Matrix Multiplication Benchmarks")
print("=" * 60)

# Small matrices
benchmark_matmul(1000, np.float32)

# Medium matrices
benchmark_matmul(5000, np.float32)

# Large matrices (may need significant GPU memory)
# benchmark_matmul(10000, np.float32)

# Compare float32 vs float64
print("\n\n" + "=" * 60)
print("Data Type Comparison (5000x5000)")
print("=" * 60)

benchmark_matmul(5000, np.float32)
benchmark_matmul(5000, np.float64)
```

Additional Linear Algebra Operations:

CuPy accelerates many linear algebra operations through cuBLAS and cuSOLVER:

```
import numpy as np
import cupy as cp
import time

n = 5000

# Create symmetric positive definite matrix
A_cpu = np.random.randn(n, n).astype(np.float32)
A_cpu = A_cpu @ A_cpu.T + np.eye(n) * n  # Make positive definite
A_gpu = cp.asarray(A_cpu)

print("Linear Algebra Operations Comparison")
print("=" * 60)
```

```
# 1. Eigenvalue decomposition
print("\nEigenvalue Decomposition:")
start = time.time()
eigvals_cpu, eigvecs_cpu = np.linalg.eigh(A_cpu)
cpu_time = time.time() - start
print(f"  NumPy (CPU): {cpu_time:.3f}s")

start = time.time()
eigvals_gpu, eigvecs_gpu = cp.linalg.eigh(A_gpu)
cp.cuda.Stream.null.synchronize()
gpu_time = time.time() - start
print(f"  CuPy (GPU): {gpu_time:.3f}s")
print(f"  Speedup: {cpu_time/gpu_time:.1f}x")

# 2. SVD (Singular Value Decomposition)
print("\nSingular Value Decomposition:")
B_cpu = np.random.randn(n, n).astype(np.float32)
B_gpu = cp.asarray(B_cpu)

start = time.time()
U_cpu, s_cpu, Vt_cpu = np.linalg.svd(B_cpu)
cpu_time = time.time() - start
print(f"  NumPy (CPU): {cpu_time:.3f}s")

start = time.time()
U_gpu, s_gpu, Vt_gpu = cp.linalg.svd(B_gpu)
cp.cuda.Stream.null.synchronize()
gpu_time = time.time() - start
print(f"  CuPy (GPU): {gpu_time:.3f}s")
print(f"  Speedup: {cpu_time/gpu_time:.1f}x")

# 3. Solving linear systems
print("\nSolving Linear System (Ax = b):")
b_cpu = np.random.randn(n).astype(np.float32)
b_gpu = cp.asarray(b_cpu)

start = time.time()
x_cpu = np.linalg.solve(A_cpu, b_cpu)
cpu_time = time.time() - start
print(f"  NumPy (CPU): {cpu_time:.3f}s")

start = time.time()
x_gpu = cp.linalg.solve(A_gpu, b_gpu)
cp.cuda.Stream.null.synchronize()
gpu_time = time.time() - start
print(f"  CuPy (GPU): {gpu_time:.3f}s")
```

```
print(f"  Speedup: {cpu_time/gpu_time:.1f}x")

# 4. Matrix inversion
print("\nMatrix Inversion:")
start = time.time()
A_inv_cpu = np.linalg.inv(A_cpu)
cpu_time = time.time() - start
print(f"  NumPy (CPU): {cpu_time:.3f}s")

start = time.time()
A_inv_gpu = cp.linalg.inv(A_gpu)
cp.cuda.Stream.null.synchronize()
gpu_time = time.time() - start
print(f"  CuPy (GPU): {gpu_time:.3f}s")
print(f"  Speedup: {cpu_time/gpu_time:.1f}x")
```

Why CuPy Dominates Here:

- **cuBLAS**: NVIDIA's highly tuned BLAS library
- **Hardware acceleration**: Tensor cores on modern GPUs
- **Memory bandwidth**: GPU memory is much faster
- **Parallelism**: Thousands of GPU cores vs handful of CPU cores

Perfect Drop-in Replacement:

```
# Original NumPy code
import numpy as np
A = np.random.randn(5000, 5000)
B = np.random.randn(5000, 5000)
C = A @ B
eigenvalues, eigenvectors = np.linalg.eigh(C)

# GPU-accelerated version (just change import!)
import cupy as cp
A = cp.random.randn(5000, 5000)
B = cp.random.randn(5000, 5000)
C = A @ B
eigenvalues, eigenvectors = cp.linalg.eigh(C)
```

Key Takeaways:

- **Minimal effort**: Often just changing the import statement
- **Maximum performance**: Leverages highly optimized GPU libraries
- **No expertise needed**: Don't need to understand GPU programming
- **Ideal for standard operations**: Where libraries beat custom code

5.11 Installation

Basic Installation:

```
python -m venv .venv
source .venv/bin/activate

# For CUDA 12.x
pip install cupy-cuda12x

# For CUDA 11.x
pip install cupy-cuda11x
```

Check Installation:

```
python -c "import cupy as cp; print(cp.cuda.runtime.getDeviceCount())"
```

Verify cuBLAS/cuFFT:

```
python -c "import cupy as cp; a = cp.random.randn(100, 100); print((a @ a).shape)"
```

6 CUDA Python

CUDA Python is NVIDIA's official Python interface to CUDA, providing low-level access to the CUDA runtime and driver APIs directly from Python. It enables developers to leverage the full power of CUDA while staying within the Python ecosystem.

6.1 Overview

CUDA Python provides Pythonic bindings to CUDA's C APIs, allowing developers to access low-level GPU programming features without writing C/C++ code. Unlike higher-level libraries that abstract away CUDA details, CUDA Python exposes the full CUDA programming model, giving you fine-grained control over GPU resources.

Key advantages:

- Direct access to CUDA runtime and driver APIs from Python
- Full control over GPU resources (streams, events, memory, contexts)
- Thin Python wrapper around CUDA C APIs - minimal overhead
- Seamless interoperability with existing CUDA ecosystem
- Official NVIDIA support and maintenance

6.2 Brief History

CUDA Python represents NVIDIA's evolution toward better Python support for GPU computing. While CUDA has always been accessible from Python through various third-party tools, NVIDIA developed official Python bindings to provide a standardized, well-supported interface.

Key milestones:

- **2020**: NVIDIA announced plans for official Python bindings to CUDA
- **2021**: Initial release of cuda-python package with core runtime APIs
- **2022**: Expanded coverage of CUDA APIs, added Driver API support

- **2023**: Introduction of `cuda.core` - a higher-level Pythonic interface
- **2024-Present**: Continuous expansion of API coverage and improved Python ergonomics

CUDA Python is maintained by NVIDIA and represents the company's commitment to making CUDA accessible to the broader Python scientific computing community while maintaining low-level control.

6.3 What CUDA Python Does

Core Functionality:

- Official Python bindings to CUDA C APIs
- Low-level access to GPU programming features
- Thin wrapper maintaining CUDA's performance characteristics
- Bridges gap between high-level Python and low-level CUDA

API Coverage:

Runtime API Bindings:

- Memory management (malloc, memcpy, memset)
- Stream and event management
- Device management and queries
- Kernel launch configuration
- Texture and surface memory

Driver API Bindings:

- Module and function management
- Context management
- Advanced memory operations
- Unified memory control
- Peer-to-peer device access

cuda.core Module:

- Higher-level Pythonic interface built on top of runtime/driver APIs
- Object-oriented abstractions for CUDA concepts
- Context managers for resource management
- More intuitive API while maintaining low-level control

Memory Management:

- Direct control over device memory allocation
- Support for pinned (page-locked) host memory
- Managed (unified) memory support
- Zero-copy memory for integrated GPUs
- Memory pools and asynchronous allocation

Stream and Synchronization:

- Create and manage CUDA streams for concurrent execution
- Event-based synchronization and timing
- Stream priorities and callbacks

- Graph execution support

Interoperability:

- Works with arrays from CuPy, PyTorch, TensorFlow via pointer interfaces
- Compatible with Numba CUDA kernels
- Integrates with C/C++ CUDA code
- Supports external memory and semaphore sharing

6.4 Use Cases

- Custom GPU Resource Management: Building custom task schedulers, implementing memory-efficient algorithms, multi-GPU workload distribution
- Integrating Existing CUDA C/C++ Code: Wrapping proprietary CUDA libraries, using vendor-optimized kernels, migrating legacy CUDA applications
- Advanced Performance Optimization: Overlapping computation and memory transfers, multi-stream execution, custom profiling and timing
- Low-Level Algorithm Implementation: Custom ray tracing kernels, advanced image processing with texture memory, lock-free data structures
- Framework Development: Developing domain-specific GPU libraries, building workflow engines, creating custom array libraries
- Multi-GPU and Distributed Computing: Multi-GPU molecular dynamics, distributed deep learning, multi-GPU linear solvers

6.5 CUDA Python vs CuPy

Understanding the fundamental differences helps you choose the right tool for your needs.

6.5.1 Abstraction Level

CUDA Python:

- Low-level access to CUDA APIs
- You manage memory, streams, kernels explicitly
- Mirrors CUDA C programming model
- Requires understanding of CUDA concepts

CuPy:

- High-level NumPy-compatible arrays
- Automatic memory and execution management
- Hides CUDA details behind NumPy interface
- Minimal CUDA knowledge required

6.5.2 Programming Model

CUDA Python:

```
# Explicit memory allocation and management
import cuda.core as cuda

device = cuda.Device()
mem_handle = device.allocate(nbytes)
# Manual memory copies, kernel launches, synchronization
```

CuPy:

```
# NumPy-style array operations
import cupy as cp

a = cp.array([1, 2, 3])  # Memory managed automatically
b = a + 5  # Operations execute automatically on GPU
```

6.5.3 Use Case Comparison

CUDA Python

- You need low-level control over GPU resources
- Integrating existing CUDA C/C++ code
- Implementing custom memory management strategies
- Building frameworks or libraries
- Performance tuning requires explicit stream/memory control
- You're a CUDA expert wanting Python bindings

CuPy

- You have NumPy code to accelerate
- You want array-level operations
- You prefer automatic resource management
- You want pre-optimized library functions
- Development speed matters more than fine-grained control
- You're a Python/NumPy expert learning GPU computing

6.5.4 Complementary Usage

They work well together:

- Use CuPy for high-level array operations
- Use CUDA Python to access CuPy array pointers for custom operations
- Leverage CuPy's memory allocator with CUDA Python's stream management
- Combine CuPy's convenience with CUDA Python's control

Example workflow:

```
import cupy as cp
import cuda.core as cuda

# CuPy creates and manages arrays
data = cp.random.randn(1000000)
```

```
# Access underlying pointer for CUDA Python operations
ptr = data.data.ptr
stream = cuda.Stream()

# Use CUDA Python for custom low-level operations
# while CuPy handles memory lifecycle
```

6.5.5 Comparison Table

Criterion	CUDA Python	CuPy
Abstraction level	Low (direct CUDA APIs)	High (NumPy arrays)
Learning curve	Steep (requires CUDA knowledge)	Gentle (NumPy knowledge)
Control	Complete control over GPU	Automatic management
Code verbosity	More verbose	Concise
Best for	Custom kernels, framework building	Array operations, algorithm implementation
Memory management	Manual (explicit allocation)	Automatic (array lifecycle)
CUDA expertise	Required	Not required
Development speed	Slower (more code)	Faster (less code)
Use case	Low-level optimization, integration	Accelerating NumPy workflows

6.6 Getting Started with CUDA Python

Installation:

```
python -m venv .venv
source .venv/bin/activate
pip install cuda-python
```

Basic Example - Device Query:

```
from cuda import cuda

# Initialize CUDA
err, = cuda.cuInit(0)
assert err == cuda.CUresult.CUDA_SUCCESS

# Get device count
err, device_count = cuda.cuDeviceGetCount()
print(f"Found {device_count} CUDA devices")
```

```python
# Query device properties
err, device = cuda.cuDeviceGet(0)
err, name = cuda.cuDeviceGetName(128, device)
print(f"Device 0: {name.decode()}")
```

Using cuda.core (Higher-level interface):

```python
import cuda.core as cuda

# More Pythonic interface
device = cuda.Device()
print(f"Device: {device.name}")
print(f"Compute capability: {device.compute_capability}")
print(f"Total memory: {device.total_memory / 1e9:.2f} GB")

# Context managers for resource safety
with cuda.Stream() as stream:
    # Operations in this stream
    pass
```

Hardware Requirements:

- NVIDIA GPU with CUDA support (Compute Capability 3.0+)
- CUDA Toolkit installed (version 11.2 or later recommended)
- Compatible with all CUDA-capable NVIDIA GPUs

Note: CUDA Python is typically used when you need the lowest-level access or are integrating existing CUDA C/C++ code. For most scientific computing tasks, CuPy or Numba provide better productivity while CUDA Python excels at framework development and custom optimization.

7 JAX

JAX is a high-performance numerical computing library that combines NumPy's familiar API with automatic differentiation and just-in-time compilation. It's designed for machine learning research and high-performance scientific computing.

7.1 Overview

JAX provides a powerful framework for numerical computing with automatic differentiation, making it ideal for optimization problems, machine learning, and scientific computing requiring gradients. Unlike traditional array libraries, JAX is built around composable function transformations that enable elegant solutions to complex problems.

Key Characteristics:

- **NumPy-Compatible API**: Familiar `jax.numpy` interface for array operations
- **Automatic Differentiation**: Built-in gradient computation with `grad`, `vjp`, `jvp`
- **JIT Compilation**: XLA compiler produces optimized code for **CPU, GPU, TPU**
- **Functional Programming**: Pure functions enable powerful transformations

- **Composable Transformations**: Combine `jit`, `grad`, `vmap`, `pmap` freely

Why JAX for Scientific Computing:

- Essential for gradient-based optimization and inverse problems
- Clean, composable code through functional programming
- High performance through XLA compilation
- Multi-device parallelization with minimal code
- Growing ecosystem for scientific computing and ML

7.2 History

JAX was developed by Google Research and emerged from the machine learning research community's need for flexible automatic differentiation.

Key Milestones:

- **2018**: JAX announced by Google Research, building on Autograd
- **2019**: Open-sourced, gained traction in ML research community
- **2020**: Added TPU support, improved XLA compilation
- **2021**: `pmap` for multi-device parallelization stabilized
- **2022**: Growing adoption in scientific computing beyond ML
- **2023**: Array API compatibility, improved documentation
- **Present**: Core library for ML research, expanding into scientific computing

JAX is maintained by Google Research and has become essential infrastructure for machine learning research, with growing adoption in physics, chemistry, and computational science.

7.3 What JAX Does

Core Functionality:

- **Automatic Differentiation**: Compute gradients of arbitrary Python/NumPy functions
- **JIT Compilation**: XLA compiler optimizes and compiles functions to machine code
- **Vectorization**: Automatically map functions over batch dimensions
- **Parallelization**: Distribute computation across multiple GPUs/TPUs
- **Functional Transformations**: Composable tools that transform functions

Function Transformations:

- `grad()`: Compute gradients automatically
- `jit()`: Just-in-time compile for performance
- `vmap()`: Vectorize over batch dimensions
- `pmap()`: Parallelize across devices
- `vjp()` / `jvp()`: Vector-Jacobian and Jacobian-vector products

XLA Compilation:

- Uses Google's Accelerated Linear Algebra (XLA) compiler
- Fuses operations for efficiency
- Generates optimized code for CPU, GPU, TPU
- Significant performance improvements over NumPy

Hardware Support:

- CPU: Optimized execution via XLA
- NVIDIA GPUs: Full CUDA support
- Google TPUs: Native support
- AMD GPUs: Experimental ROCm support

7.4 Use Cases

JAX has become essential in domains requiring gradients, optimization, and high-performance computing.

7.4.1 1. Machine Learning Research

Custom models, loss functions, and training loops benefit from automatic differentiation and JIT compilation.

Example Use Cases:

- Novel neural network architectures
- Custom optimization algorithms
- Reinforcement learning environments
- Probabilistic programming

7.4.2 2. Optimization and Inverse Problems

Gradient-based optimization is natural with JAX's automatic differentiation.

Example Use Cases:

- Parameter estimation in differential equations
- Inverse problems in physics
- Optimal control problems
- Variational methods

7.4.3 3. Physics-Informed Neural Networks

Combining physics equations with neural networks requires automatic differentiation.

Example Use Cases:

- Solving PDEs with neural networks
- Enforcing conservation laws in models
- Hybrid physics-ML simulations
- Scientific machine learning

7.4.4 4. Computational Physics

Simulations requiring gradients benefit from JAX's autodiff capabilities.

Example Use Cases:

- Molecular dynamics with learned potentials

- Quantum many-body systems
- Variational Monte Carlo
- Hamiltonian mechanics
- Compressible flows

7.4.5 5. Bayesian Inference

Probabilistic modeling and sampling algorithms leverage JAX's transformations.

Example Use Cases:

- Hamiltonian Monte Carlo (HMC)
- Variational inference
- Normalizing flows
- Uncertainty quantification

7.4.6 6. Multi-Device Scientific Computing

Large-scale simulations benefit from JAX's easy multi-GPU/TPU parallelization.

Example Use Cases:

- Distributed array computations
- Large-scale optimization
- Data-parallel training
- Multi-GPU simulations

7.5 Functional Programming Model

JAX requires **pure functions** for its transformations to work correctly. This is a fundamental difference from NumPy.

7.5.1 Pure Functions

What is a Pure Function:

- Output depends only on inputs (no hidden state)
- No side effects (no print, file I/O, global variables)
- Same inputs always produce same outputs
- Can be safely transformed and optimized

Good (Pure Function):

```
def compute(x, y):
    return x ** 2 + y ** 2  # Pure: no side effects
```

Bad (Impure Function):

```
counter = 0  # Global state

def compute(x, y):
    global counter
```

```
    counter += 1  # Side effect!
    print(f"Call {counter}")  # Side effect!
    return x ** 2 + y ** 2
```

Why Purity Matters:

- JAX traces functions once, replays later
- Side effects during tracing won't repeat during execution
- Transformations assume functions are pure
- Optimization requires predictable behavior

7.5.2 Immutable Arrays

JAX Arrays are Immutable:

- Cannot modify arrays in-place
- Operations return new arrays
- Functional programming style

NumPy (Mutable):

```
import numpy as np
x = np.array([1, 2, 3])
x[0] = 10  # In-place modification OK
```

JAX (Immutable):

```
import jax.numpy as jnp
x = jnp.array([1, 2, 3])
# x[0] = 10  # Error! Cannot modify
x = x.at[0].set(10)  # Create new array
```

Benefits:

- Safe parallelization
- No race conditions
- Easier to reason about code
- Enables optimizations

7.6 Core Transformations

JAX's power comes from composable function transformations.

7.6.1 jit: Just-in-Time Compilation

What it does:

- Compiles function to optimized machine code
- Uses XLA compiler
- Significant speedup for numerical code
- Caches compiled functions

Example:

```
import jax
import jax.numpy as jnp

def slow_function(x):
    return jnp.sum(x ** 2) + jnp.mean(x ** 3)

# Compiled version
fast_function = jax.jit(slow_function)

# Or use decorator
@jax.jit
def fast_function(x):
    return jnp.sum(x ** 2) + jnp.mean(x ** 3)

# First call: compilation overhead
x = jnp.arange(1000000)
result = fast_function(x)  # Compiles here

# Subsequent calls: fast
result = fast_function(x)  # Uses cached version
```

Performance:

- First call: compilation overhead (slower)
- Subsequent calls: 5-100x faster than uncompiled
- Best for functions called many times
- Most effective with complex computations

When to Use:

- Functions called repeatedly
- Numerical computations
- Inner loops in algorithms
- After verifying correctness (debug without jit first)

JAX vs Numba JIT Compilation:

Aspect	JAX	Numba
Compiler	XLA (Google)	LLVM
Compilation Trigger	Shape + dtype	Type signature
Tracing	Traces Python to XLA IR	Compiles Python bytecode
Re-compilation	Per shape change	Per type change
Target	CPU/GPU/TPU	CPU/GPU (CUDA)
Automatic Differentiation	Yes (built-in)	No
First Call	Slow (compilation overhead)	Slow (compilation overhead)

Aspect	JAX	Numba
Subsequent Calls	Fast (cached)	Fast (cached)

Both JAX and Numba perform **just-in-time (JIT) compilation at runtime**, not ahead-of-time compilation. The first call to a JIT-decorated function triggers compilation, which is then cached for reuse.

7.6.2 grad: Automatic Differentiation

What it does:

- Computes gradients automatically
- Reverse-mode autodiff (backpropagation)
- Works with arbitrary Python control flow
- Essential for optimization

Example:

```
import jax
import jax.numpy as jnp

# Define function
def loss(params):
    x, y = params
    return x ** 2 + 3 * x * y + 5 * y ** 2

# Compute gradient
grad_loss = jax.grad(loss)

# Evaluate gradient at point
params = jnp.array([1.0, 2.0])
gradient = grad_loss(params)
print(f"Gradient: {gradient}")  # [8.0, 23.0]
```

With Multiple Arguments:

```
def loss(x, y, z):
    return x ** 2 + y * z

# Gradient w.r.t. first argument (default)
grad_x = jax.grad(loss, argnums=0)

# Gradient w.r.t. all arguments
grad_all = jax.grad(loss, argnums=(0, 1, 2))
```

Combining with jit:

```
@jax.jit
@jax.grad
```

```
def fast_gradient(params):
    return loss(params)
```

Use Cases:

- Training neural networks
- Gradient-based optimization
- Inverse problems
- Sensitivity analysis

7.6.3 vmap: Automatic Vectorization

What it does:

- Vectorizes function over batch dimension
- No explicit loops needed
- Efficient parallel execution
- Clean, readable code

Example:

```
import jax
import jax.numpy as jnp

# Function for single input
def f(x):
    return x ** 2 + jnp.sin(x)

# Vectorize over batch
batched_f = jax.vmap(f)

# Apply to batch
batch = jnp.arange(10)
results = batched_f(batch)  # Processes all elements in parallel
```

With Multiple Arguments:

```
def compute(x, y):
    return x * y + x ** 2

# Vectorize over first dimension of both arguments
batched_compute = jax.vmap(compute)

x_batch = jnp.array([1, 2, 3])
y_batch = jnp.array([4, 5, 6])
result = batched_compute(x_batch, y_batch)
```

Specify Vectorization Axis:

```
# Vectorize over axis 1
batched_f = jax.vmap(f, in_axes=1, out_axes=1)
```

Use Cases:

- Processing batches of data
- Parallelizing over samples
- Monte Carlo simulations
- Parameter scans

7.6.4 pmap: Multi-Device Parallelization

What it does:

- Distributes computation across multiple GPUs/TPUs
- Data parallelism with minimal code
- Automatic communication
- Scales to many devices

Example:

```
import jax
import jax.numpy as jnp

# Function to run on each device
def compute(x):
    return jnp.sum(x ** 2)

# Parallelize across devices
parallel_compute = jax.pmap(compute)

# Data for each device (first dimension = number of devices)
n_devices = jax.device_count()
data = jnp.ones((n_devices, 1000))

# Compute on all devices in parallel
results = parallel_compute(data)
print(f"Results shape: {results.shape}")  # (n_devices,)
```

Collective Operations:

```
@jax.pmap
def parallel_sum(x):
    # Sum across all devices
    return jax.lax.psum(x, axis_name='devices')
```

Use Cases:

- Multi-GPU training
- Distributed computing
- Large-scale simulations
- Data parallelism

7.6.5 Composing Transformations

JAX transformations compose naturally:

```
# Gradient of vectorized function
grad_batched = jax.grad(jax.vmap(f))

# JIT-compiled gradient
fast_grad = jax.jit(jax.grad(loss))

# Parallel gradient computation
parallel_grad = jax.pmap(jax.grad(loss))

# Gradient of JIT-compiled vectorized function
complex = jax.grad(jax.jit(jax.vmap(f)))
```

7.7 Examples

7.7.1 Gradient-Based Optimization

Demonstrates JAX's core strength: automatic differentiation for optimization.

Problem:

Find parameters that minimize a loss function using gradient descent.

Key Concepts:

- Automatic differentiation with `grad`
- JIT compilation for speed
- Optimization loop
- Parameter updates

Implementation:

```
import jax
import jax.numpy as jnp
import matplotlib.pyplot as plt

# Loss function: Rosenbrock function (banana function)
def rosenbrock(params):
    x, y = params
    return (1 - x) ** 2 + 100 * (y - x ** 2) ** 2

# Gradient function
grad_rosenbrock = jax.jit(jax.grad(rosenbrock))

# Gradient descent optimization
def optimize(initial_params, learning_rate=0.001, n_steps=1000):
    params = initial_params
    history = [params]
```

```
    for step in range(n_steps):
        gradient = grad_rosenbrock(params)
        params = params - learning_rate * gradient
        history.append(params)

        if step % 100 == 0:
            loss = rosenbrock(params)
            print(f"Step {step}, Loss: {loss:.6f}")

    return params, jnp.array(history)

# Run optimization
initial = jnp.array([-1.0, 1.0])
final_params, history = optimize(initial)

print(f"Final parameters: {final_params}")
print(f"Final loss: {rosenbrock(final_params):.6f}")
print(f"Optimum at [1, 1]: {rosenbrock(jnp.array([1.0, 1.0])):.6f}")
```

Key Points:

- `jax.grad` computes gradient automatically
- No manual derivative calculation needed
- `jax.jit` makes gradient computation fast
- Works with arbitrary Python functions

Performance:

- Gradient computation: automatic and fast
- JIT compilation: 10-100x speedup
- Scales to high-dimensional problems

7.7.2 Physics-Informed Neural Network

Demonstrates JAX's power for scientific computing with automatic differentiation.

Problem:

Solve differential equation: dy/dx = -y, y(0) = 1 (solution: y = exp(-x))

Key Concepts:

- Neural network as function approximator
- Automatic differentiation for physics constraints
- Training to satisfy differential equation
- Combining ML and physics

Implementation:

```
import jax
import jax.numpy as jnp
```

```python
# Simple neural network
def neural_net(params, x):
    """Two-layer neural network"""
    w1, b1, w2, b2 = params
    hidden = jnp.tanh(w1 * x + b1)
    output = w2 * hidden + b2
    return output

# Physics loss: enforce dy/dx = -y
def physics_loss(params, x):
    """Loss based on differential equation"""
    # Function value
    y = neural_net(params, x)

    # Derivative dy/dx using autodiff
    dy_dx = jax.grad(lambda params: neural_net(params, x).sum())(params)

    # Physics constraint: dy/dx + y = 0
    residual = dy_dx[0] * x + y + y  # Simplified for this network

    return residual ** 2

# Initial condition loss: y(0) = 1
def initial_condition_loss(params):
    x0 = jnp.array(0.0)
    y0_pred = neural_net(params, x0)
    y0_true = 1.0
    return (y0_pred - y0_true) ** 2

# Total loss
def total_loss(params, x_points):
    phys_loss = jnp.mean(jax.vmap(lambda x: physics_loss(params, x))(x_points))
    ic_loss = initial_condition_loss(params)
    return phys_loss + ic_loss

# Training
@jax.jit
def update(params, x_points, learning_rate):
    loss = total_loss(params, x_points)
    grads = jax.grad(total_loss)(params, x_points)
    # Update parameters
    params = jax.tree_map(lambda p, g: p - learning_rate * g, params, grads)
    return params, loss

# Initialize network
key = jax.random.PRNGKey(0)
```

```
params = [
    jax.random.normal(key, ()) * 0.1,  # w1
    jax.random.normal(key, ()) * 0.1,  # b1
    jax.random.normal(key, ()) * 0.1,  # w2
    jax.random.normal(key, ()) * 0.1,  # b2
]

# Training points
x_points = jnp.linspace(0, 2, 20)

# Train
for step in range(1000):
    params, loss = update(params, x_points, learning_rate=0.01)
    if step % 100 == 0:
        print(f"Step {step}, Loss: {loss:.6f}")

# Evaluate
x_test = jnp.linspace(0, 2, 100)
y_pred = jax.vmap(lambda x: neural_net(params, x))(x_test)
y_true = jnp.exp(-x_test)

print(f"Mean error: {jnp.mean(jnp.abs(y_pred - y_true)):.6f}")
```

Key Points:

- `jax.grad` computes derivatives for physics constraints
- Neural network learned to satisfy differential equation
- No numerical solver needed
- Combines deep learning with physics

Why JAX Excels Here:

- Automatic differentiation through neural network
- Can compute higher-order derivatives
- Fast gradient computation with JIT
- Essential for physics-informed ML

7.8 JAX vs NumPy: Key Differences

Similarities:

- Very similar API (`jax.numpy` mirrors `numpy`)
- Same array operations and functions
- Easy to port NumPy code to JAX

Key Differences:

Feature	NumPy	JAX
Arrays	Mutable	Immutable

Feature	NumPy	JAX
Random Numbers	Global state	Explicit keys
Performance	CPU optimized	CPU/GPU/TPU via XLA
Differentiation	No	Automatic with `grad`
Compilation	Interpreted	JIT via XLA
Vectorization	Manual loops	`vmap` transformation
Parallelization	No	`pmap` for multi-device

Porting NumPy to JAX:

```
# NumPy code
import numpy as np
x = np.random.randn(1000)
y = np.sum(x ** 2)

# JAX equivalent
import jax.numpy as jnp
key = jax.random.PRNGKey(0)
x = jax.random.normal(key, (1000,))
y = jnp.sum(x ** 2)
```

7.9 Best Practices

7.9.1 When to Use JAX

Excellent for:

- Gradient-based optimization
- Machine learning research
- Physics-informed neural networks
- Automatic differentiation needs
- Multi-GPU/TPU computing

Not ideal for:

- Simple NumPy operations (use CuPy)
- Code with many side effects (need to refactor/revise)
- Heavy I/O operations
- When you don't need gradients

7.9.2 Development Workflow

1. Start Without Transformations:

```
# First: write and test without jit/grad
def my_function(x):
    return x ** 2
```

```
# Test it works
result = my_function(jnp.array([1, 2, 3]))
```

2. Add Transformations Gradually:

```
# Then: add jit for performance
@jax.jit
def my_function(x):
    return x ** 2
```

3. Debug Without JIT:

```
# Remove @jax.jit for debugging
def my_function(x):
    print(f"x = {x}")  # Debug print
    return x ** 2
```

7.9.3 Performance Tips

Do:

- Use JIT for functions called repeatedly
- Keep data on GPU across operations
- Use `vmap` instead of explicit loops
- Combine transformations for efficiency

Don't:

- JIT functions with side effects
- Transfer data between CPU/GPU unnecessarily
- Use Python loops when `vmap` works
- Compile functions called only once

7.9.4 Common Pitfalls

Pitfall 1: Mutable Operations

```
# Wrong: trying to mutate
x[0] = 5  # Error!

# Right: functional update
x = x.at[0].set(5)
```

Pitfall 2: Side Effects in JIT

```
# Wrong: side effects in jitted function
@jax.jit
def bad(x):
    print(f"x = {x}")  # Only prints during tracing!
    return x ** 2

# Right: remove jit for debugging
```

```python
def good(x):
    print(f"x = {x}")
    return x ** 2
```

Pitfall 3: Global Random State

```python
# Wrong: reusing same key
key = jax.random.PRNGKey(0)
x1 = jax.random.normal(key, (100,))
x2 = jax.random.normal(key, (100,))  # Same random numbers!

# Right: split keys
key = jax.random.PRNGKey(0)
key, subkey1 = jax.random.split(key)
key, subkey2 = jax.random.split(key)
x1 = jax.random.normal(subkey1, (100,))
x2 = jax.random.normal(subkey2, (100,))  # Different random numbers
```

7.10 Installation

Basic Installation:

```bash
python -m venv .venv
source .venv/bin/activate

# CPU only
pip install jax

# CUDA 12.x (NVIDIA GPUs)
pip install jax[cuda12]

# CUDA 11.x (NVIDIA GPUs)
pip install jax[cuda11]
```

Verify Installation:

```python
import jax
print(f"JAX version: {jax.__version__}")
print(f"Devices: {jax.devices()}")
print(f"Default backend: {jax.default_backend()}")

# Check GPU support
try:
    import jax.numpy as jnp
    x = jnp.array([1, 2, 3])
    print(f"Array device: {x.device()}")
except:
    print("GPU not available")
```

Hardware Requirements:

- **CPU**: Any modern processor
- **NVIDIA GPU**: CUDA-capable (Compute Capability 3.5+)
- **TPU**: Google Cloud TPU (TPU v2, v3, v4)
- **AMD GPU**: Experimental ROCm support

7.11 Summary

JAX's Core Strengths:

1. **Automatic Differentiation**: Essential for optimization and ML
2. **JIT Compilation**: XLA produces fast code for CPU/GPU/TPU
3. **Composable Transformations**: `grad`, `jit`, `vmap`, `pmap` work together
4. **Functional Programming**: Pure functions enable safe transformations
5. **NumPy-Compatible**: Easy to learn if you know NumPy

When to Use JAX:

- Need gradients for optimization
- Doing machine learning research
- Solving inverse problems
- Physics-informed neural networks
- Multi-GPU/TPU computing

When to Use Other Frameworks:

- **CuPy**: NumPy code without gradients
- **Numba**: Custom kernels, loop-heavy code
- **CUDA Python**: Low-level GPU control
- **PyTorch**: Standard deep learning

JAX bridges the gap between high-level NumPy-style programming and high-performance computing with automatic differentiation, making it ideal for gradient-based scientific computing and machine learning research.

8 GPU Python Frameworks Comparison

8.1 Comparison Matrix

Aspect	Numba	CuPy	CUDA Python	JAX
Abstraction Level	Medium (functions + kernels)	High (arrays)	Low (CUDA APIs)	High (functional arrays)
Learning Curve	Moderate	Gentle	Steep	Moderate
Primary Use Case	Loop-heavy custom algorithms	NumPy code acceleration	CUDA integration & framework building	Auto-diff & functional programming

Aspect	Numba	CuPy	CUDA Python	JAX
Programming Model	JIT-compiled Python functions	NumPy-compatible arrays	Explicit CUDA API calls	Pure functional transformations
Memory Management	Semi-automatic	Automatic	Manual (explicit)	Automatic
Compilation	JIT (runtime)	Ahead-of-time (kernels)	None (thin wrapper)	JIT (XLA compiler)
GPU Vendor Support	NVIDIA only	NVIDIA (AMD experimental)	NVIDIA only	NVIDIA, TPU, AMD (partial)
CPU Fallback	Native (same code)	Via NumPy interop	Not applicable	Native (same code)
Custom Kernels	Primary feature (@cuda.jit)	Available (ElementwiseKernel)	Pre-compiled or manual	Not directly (use XLA)
Auto-differentiation	Not built-in	Not built-in	Not applicable	Core feature (grad, vjp, jvp)
Parallelization	Explicit (kernel/parallel)	Automatic	Manual	Automatic (jit, pmap, vmap)
Best For	Custom algorithms, physics sims	Existing NumPy workflows	Low-level control, CUDA experts	ML research, optimization, auto-diff
Code Verbosity	Moderate	Low (concise)	High (verbose)	Low-Moderate
Debugging	Moderate difficulty	Easier	Difficult	Moderate difficulty
Ecosystem Integration	NumPy, SciPy	NumPy, SciPy, PyTorch, TF	All CUDA libraries	NumPy, SciPy, ML frameworks
Maintained By	Anaconda Inc.	Preferred Networks	NVIDIA	Google Research
Open Source	Yes (BSD)	Yes (MIT)	Yes (NVIDIA license)	Yes (Apache 2.0)
Documentation Quality	Good	Excellent	Moderate (improving)	Excellent

Aspect	Numba	CuPy	CUDA Python	JAX
Performance	Excellent (custom tuned)	Excellent (optimized libs)	Excellent (full control)	Excellent (XLA optimized)
Type System	Static typing via JIT	Dynamic (NumPy-like)	Static (CUDA)	Dynamic (traced)
Multi-GPU	Manual	Supported	Full control	Built-in (pmap)
Typical Speedup	10-100x over Python	10-100x over NumPy	Maximum possible	10-100x + auto-diff
Installation Complexity	Easy (pip)	Easy (pip)	Easy (pip)	Easy (pip)
Production Ready	Yes	Yes	Yes (for experts)	Yes (research-focused)
When to Use	Custom loops & algorithms	Drop-in NumPy replacement	CUDA integration	ML, optimization, auto-diff

8.2 Framework Selection Guide

8.2.1 Choose Numba When:

Custom Algorithm Implementation - You're implementing novel algorithms not available in libraries - Your code has nested Python loops that need acceleration - You need explicit control over parallelization logic - You want to write CUDA kernels in Python syntax

Cross-Platform Requirements - You need the same code to run efficiently on both CPU and GPU - CPU-only deployment is also a requirement - You want to avoid maintaining separate implementations

Physics Simulations and Scientific Computing - N-body simulations with custom force calculations - Monte Carlo methods with complex sampling logic - Computational fluid dynamics with custom solvers - Molecular dynamics with specific interaction models

Fine-Grained Optimization - You need control over thread indexing and shared memory - Your algorithm benefits from manual kernel optimization - You understand parallel programming concepts

Example Use Cases: - Custom differential equation solvers - Particle simulations with complex interactions - Custom signal processing algorithms - Optimization routines with specific heuristics

8.2.2 Choose CuPy When:

NumPy Code Acceleration - You have existing NumPy code that needs GPU acceleration - Your algorithms are already vectorized with NumPy operations - You want minimal code changes (often just import changes) - You prefer array-level thinking over kernel-level

High-Level Array Operations - Matrix multiplication and linear algebra (cuBLAS) - FFT and

spectral analysis (cuFFT) - Statistical operations on large arrays - Standard mathematical operations

Rapid Prototyping - Development speed is critical - You want to experiment quickly with GPU acceleration - You're exploring whether GPU acceleration helps your workload

Working with Large Datasets - Climate data analysis - Genomics data processing - Financial time series analysis - Image processing pipelines

Example Use Cases: - Porting existing NumPy scientific code to GPU - Large-scale linear algebra operations - Statistical analysis on massive datasets - Standard signal processing workflows

8.2.3 Choose CUDA Python When:

Low-Level Control Needed - You need precise control over GPU memory layout - Your application requires custom memory management strategies - You're implementing memory pools or custom allocators

Integrating Existing CUDA Code - You have existing CUDA C/C++ kernels to use from Python - You're wrapping proprietary CUDA libraries - You need to load pre-compiled PTX or cubin files

Framework and Library Development - You're building a higher-level GPU computing framework - You need programmatic access to CUDA APIs - You're creating domain-specific GPU libraries

Advanced Multi-GPU Coordination - Complex peer-to-peer memory transfers - Custom multi-GPU scheduling strategies - Explicit stream and event management across devices

Performance-Critical Applications - Every millisecond matters - You need to manually overlap computation and memory transfers - You're squeezing maximum performance from hardware

Example Use Cases: - Building custom deep learning frameworks - Developing GPU-accelerated simulation engines - Creating domain-specific GPU libraries - Implementing custom runtime systems

8.2.4 Choose JAX When:

Machine Learning and Auto-Differentiation - You need automatic differentiation for gradients - You're implementing custom ML models or training loops - You're doing gradient-based optimization - Research requires flexible differentiation

Functional Programming Style - You prefer pure functional programming paradigms - You want composable function transformations - You value immutability and functional purity

Advanced Parallelization Needs - Automatic vectorization (vmap) over batch dimensions - Distributed computing across multiple devices (pmap) - You want parallelization without explicit CUDA programming

Research and Experimentation - You're implementing research papers in ML/optimization - You need to experiment with different model architectures - You want rapid iteration with strong mathematical foundations

Optimization Problems - Gradient-based optimization - Variational inference - Physics-informed neural networks - Optimal control problems

Example Use Cases: - Custom neural network architectures - Bayesian inference and probabilistic programming - Scientific computing with automatic differentiation - Reinforcement learning research - Computational physics with gradient-based methods

8.3 PyTorch for Scientific Computing: Beyond Machine Learning

While PyTorch is primarily known as a deep learning framework, it's increasingly being considered as a general-purpose GPU-accelerated array library for scientific computing. This section explores PyTorch's viability as an alternative to CuPy and JAX for non-ML workloads.

8.3.1 PyTorch as a NumPy Alternative

Core Capabilities: - GPU-accelerated tensor operations (similar to CuPy's arrays) - Automatic differentiation built-in (like JAX) - NumPy-like API (`torch.tensor` vs `np.array`) - Rich ecosystem of mathematical operations - Excellent documentation and community support - Both CPU and GPU execution with minimal code changes

Tensor vs Array Philosophy: - PyTorch uses "tensors" (ML terminology) rather than "arrays" (NumPy terminology) - Core operations are nearly identical to NumPy - Designed with gradient computation in mind - Dynamic computation graph (eager execution)

8.3.2 PyTorch vs CuPy

8.3.2.1 Similarities

- Both provide GPU-accelerated array operations
- Both offer NumPy-compatible APIs
- Both handle memory management automatically
- Both support multiple GPUs
- Both have excellent performance for standard operations

8.3.2.2 Where PyTorch Excels Over CuPy **Automatic Differentiation:** - Built-in autograd system tracks operations for gradient computation - Can compute gradients without manual implementation - Essential for optimization problems, not just ML - More mature and battle-tested than third-party autodiff for CuPy

```
import torch

# PyTorch: automatic differentiation is native
x = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)
y = (x ** 2).sum()
y.backward()  # Automatic gradient computation
print(x.grad)  # [2.0, 4.0, 6.0]

# CuPy: would need external library for autodiff
```

Broader Hardware Support: - NVIDIA GPUs (CUDA) - AMD GPUs (ROCm support improving) - Apple Silicon (MPS backend) - CPU fallback is highly optimized - Better cross-platform portability than CuPy

ML Ecosystem Integration: - Seamless integration with PyTorch Lightning, Hugging Face - Easy to combine scientific computing with ML components - Large community familiar with PyTorch - Can leverage pre-trained models for hybrid workflows

Dynamic Computation: - More flexible control flow in autograd - Easier debugging with eager execution - Python-like programming without special considerations

8.3.2.3 Where CuPy Excels Over PyTorch True NumPy Compatibility: - Drop-in replacement philosophy: `import cupy as cp` - More complete SciPy coverage (via `cupy.scipy`) - Closer API match to NumPy/SciPy - Less cognitive overhead if you know NumPy

Lighter Dependency: - Smaller installation footprint - Faster import times - No ML framework overhead - Purpose-built for array computing

Scientific Computing Focus: - API design prioritizes scientific computing patterns - Better integration with scientific Python tools - More comprehensive sparse matrix support (cuSPARSE) - cuFFT and cuBLAS integration more transparent

Memory Efficiency: - More predictable memory management for scientific workloads - Memory pools optimized for scientific computing patterns - Less overhead from ML-specific features - Better control over memory layout

Performance for Non-ML Workloads: - Often faster for pure linear algebra (direct cuBLAS) - Less overhead for simple array operations - Kernel fusion more predictable - Optimized for scientific computing rather than backpropagation

8.3.3 PyTorch vs JAX

8.3.3.1 Similarities

- Both provide automatic differentiation
- Both target ML and scientific computing
- Both support multiple hardware backends
- Both offer functional programming capabilities
- Both have strong community support

8.3.3.2 Where PyTorch Excels Over JAX Eager Execution by Default: - More intuitive debugging (Python executes line by line) - Easier to learn for Python programmers - No tracing surprises - Standard Python control flow just works

```
import torch

# PyTorch: eager execution is natural
for i in range(10):
    x = torch.randn(100)
    if x.sum() > 0:  # Conditional logic works naturally
        result = x * 2
    else:
        result = x / 2
```

Larger Ecosystem and Community: - More tutorials, Stack Overflow answers, examples - Larger base of users and contributors - More third-party libraries and tools - Industry standard for deep learning

Maturity and Stability: - Longer development history (since 2016) - More stable APIs (fewer breaking changes) - Better tested in production environments - More conservative evolution

Industrial Adoption: - Widely deployed in production systems - Better tooling for deployment (TorchScript, ONNX) - More enterprise support options - Larger trained model ecosystem

Object-Oriented Design: - Module system for composable components - Stateful operations when needed - Easier for imperative programming style - Better for building complex systems

8.3.3.3 Where JAX Excels Over PyTorch

Functional Programming Paradigm: - Pure functions encourage better design - Composable transformations (`grad`, `vmap`, `pmap`) - Easier to reason about correctness - Better for mathematical/algorithmic thinking

JIT Compilation and Performance: - XLA compiler often generates faster code - Better optimization for complex computations - More aggressive kernel fusion - Automatic parallelization more sophisticated

Parallelization Primitives: - `vmap`: automatic vectorization/batching - `pmap`: distributed computing across devices - `jit`: compilation with minimal syntax - More elegant multi-GPU programming

```
import jax
import jax.numpy as jnp

# JAX: elegant parallelization
@jax.jit  # Compile for performance
@jax.vmap  # Auto-vectorize over batch
def process(x):
    return x ** 2 + x

# Automatically parallelized and optimized
data = jnp.arange(1000000)
result = process(data)
```

Advanced Differentiation: - Forward-mode and reverse-mode AD - Higher-order derivatives more natural - `jvp` and `vjp` for advanced gradient computations - Better for research in optimization methods

Immutability and Safety: - Immutable arrays reduce bugs - No hidden state changes - Easier to parallelize safely - Better for functional correctness

TPU Support: - First-class TPU support (Google hardware) - Better TPU performance than PyTorch - Important for very large-scale computing

8.3.4 Practical Considerations for Scientific Computing

8.3.4.1 Choose PyTorch for Scientific Computing When:

You Need Gradients (But Not Pure Functional Style) - Implementing physics-informed neural networks - Gradient-based

optimization problems - Inverse problems in scientific computing - Optimal control and parameter estimation

You're Already in the PyTorch Ecosystem - Your team knows PyTorch from ML work - You want to integrate with ML models - You need to combine traditional simulation with ML - You want to leverage PyTorch-based tools

You Value Ease of Debugging - Complex algorithms with conditional logic - Iterative development with frequent debugging - Prototyping and experimentation - Learning GPU computing while doing science

You Need Cross-Platform Deployment - Deployment to various GPU vendors - Apple Silicon support important - Heterogeneous computing environments

Example Scenarios: - Computational physics with learned components - Optimization problems needing gradients - Hybrid simulation-ML workflows - Parameter estimation in dynamical systems

8.3.4.2 Prefer CuPy for Scientific Computing When: You Have Pure NumPy Code

- Minimal code changes desired - No auto-differentiation needed - Standard array operations dominate - Porting existing NumPy workflows

You Want Minimal Dependencies - Lightweight deployment requirements - Fast import times critical - Avoiding ML framework overhead - Purpose-built scientific computing

You Need SciPy Compatibility - Using scipy.signal, scipy.ndimage, etc. - Scientific computing algorithms not in PyTorch - Standard scientific Python workflows

Performance for Non-ML Operations - Pure linear algebra workloads - FFT-heavy applications - Standard scientific computing patterns

8.3.4.3 Prefer JAX for Scientific Computing When: You Embrace Functional Programming

- Pure functions align with your thinking - Mathematical correctness is paramount - Composable transformations attractive - Research-oriented development

You Need Advanced Parallelization - Multi-GPU/multi-TPU computing - Elegant vectorization with `vmap` - Distributed scientific computing - Large-scale simulations

You're Doing Optimization Research - Experimenting with optimization algorithms - Higher-order derivatives needed - Gradient-based methods research - Advanced auto-differentiation requirements

TPU Access - Using Google Cloud TPUs - Very large-scale computations - Taking advantage of XLA optimizations

8.3.5 PyTorch Drawbacks for Scientific Computing

ML-Centric Design: - API design priorities reflect ML use cases - Some scientific computing patterns feel unnatural - Gradient tracking overhead when not needed - More features than necessary for pure computing

Memory Overhead: - Larger memory footprint than CuPy - Autograd graph construction even when unused - Less efficient for simple array operations - More memory consumed by framework machinery

Learning Curve for Non-ML Users: - Concepts like `requires_grad`, `.detach()`, `.item()` add complexity - Tensor vs array terminology confusion - ML-focused documentation less relevant - Need to learn what to ignore from ML features

Performance Considerations: - Not always fastest for pure linear algebra - Autograd overhead even when gradients not needed - CuPy can be faster for simple operations - JAX XLA compilation can produce faster code

Less Scientific Computing Focus: - Fewer scientific algorithms built-in - Community tilted toward ML - Less emphasis on traditional scientific computing - Sparse matrix support less mature than CuPy

8.3.6 PyTorch as a CuPy/JAX Alternative?

PyTorch is a viable option when: - You need automatic differentiation without committing to JAX's functional style - You're already familiar with PyTorch from ML work - You want to integrate scientific computing with ML models - Cross-platform deployment is important - You value mature tooling and large community

PyTorch is NOT ideal when: - You want the lightest-weight NumPy replacement (use CuPy) - Pure functional programming appeals to you (use JAX) - You have existing NumPy code with no gradients needed (use CuPy) - Maximum performance for linear algebra is critical (use CuPy) - You're building on pure scientific Python stack (use CuPy)

The Hybrid Approach: Many successful scientific computing projects use multiple frameworks:

- **PyTorch** for components needing auto-diff in imperative style
- **CuPy** for pure array computing without gradients
- **JAX** for functional gradient-based methods
- Mix and match via DLPack for zero-copy data exchange

Bottom Line: PyTorch can serve as a GPU-accelerated NumPy alternative, especially if you need automatic differentiation. However, it carries ML framework overhead that may be unnecessary for pure scientific computing. For most non-ML scientific computing, CuPy offers better ergonomics and performance, while JAX is superior for gradient-based research. PyTorch shines in the middle ground: scientific computing that benefits from gradients but doesn't fit JAX's functional paradigm, or when ML integration is valuable.

8.4 Discussion

8.4.1 Abstraction Level and Control

Numba: Mid-Level Control - Gives you access to CUDA programming model (threads, blocks, shared memory) - Abstracts away C/C++ syntax while maintaining CUDA concepts - Suitable when you understand parallel programming but want Python syntax - Good balance between control and productivity

CuPy: High-Level Convenience - Completely abstracts CUDA details behind NumPy interface - You think in terms of arrays and operations, not threads - Excellent for users who want GPU acceleration without GPU programming - Trade control for massive productivity gains

CUDA Python: Maximum Control - Direct access to all CUDA runtime and driver APIs - No abstraction - you manage everything explicitly - Suitable for CUDA experts who need Python integration - Maximum flexibility and control over GPU resources

JAX: High-Level with Transformations - Abstracts execution through functional transformations - Focus on mathematical operations and composition - XLA compiler handles low-level optimizations - You describe what to compute, not how

8.4.2 Compilation and Execution Model

Numba: JIT Compilation - Compiles Python functions to machine code at runtime - First call incurs compilation overhead - Subsequent calls execute compiled code directly - Type specialization based on input types - Can cache compiled functions to disk

CuPy: Library with Kernel Fusion - Uses pre-compiled, optimized CUDA libraries (cuBLAS, cuFFT) - Kernel fusion automatically combines operations - No compilation overhead for standard operations - Custom kernels compiled on-demand

CUDA Python: No Compilation (Wrapper) - Thin Python wrapper around CUDA C APIs - No compilation of Python code - Works with pre-compiled kernels (PTX/cubin) - Essentially zero overhead beyond Python interpreter

JAX: XLA Just-In-Time Compilation - Uses XLA (Accelerated Linear Algebra) compiler - Compiles entire computation graphs - Aggressive optimization across operations - Caches compiled functions for reuse - Can ahead-of-time compile with `jax.jit`

8.4.3 Memory Management Philosophy

Numba: Semi-Automatic - Explicit device array creation (`cuda.device_array()`) - Manual data transfers (`cuda.to_device()`, `.copy_to_host()`) - Python-managed lifetime (garbage collection) - Fine control when needed

CuPy: Fully Automatic - Arrays automatically allocated on GPU - Memory pool for efficient reuse - Python garbage collection handles cleanup - Can explicitly free memory when needed - Transparent host-device transfers

CUDA Python: Fully Manual - Explicit allocation (`cuMemAlloc()`) - Explicit transfers (`cuMemcpyHtoD()`) - Explicit deallocation (`cuMemFree()`) - Complete responsibility for memory lifecycle - Maximum control over memory behavior

JAX: Automatic with Tracing - Arrays are abstract during tracing - Actual allocation handled by XLA runtime - Efficient memory reuse through compiler optimization - Users work with immutable arrays - Memory managed by backend

8.4.4 Hardware Portability

Numba: NVIDIA Only (CPU Fallback) - CUDA support only for NVIDIA GPUs - Same code runs on CPU with `@jit` - CPU performance excellent due to LLVM - No support for AMD or other accelerators

CuPy: NVIDIA Primary (AMD Experimental) - Primary target: NVIDIA CUDA GPUs - Experimental ROCm support for AMD GPUs - Can fall back to NumPy for CPU - Best portability

among GPU-specific frameworks

CUDA Python: NVIDIA Only - Exclusively NVIDIA CUDA - Direct CUDA API bindings - No CPU fallback (doesn't make sense) - Most NVIDIA-specific of all frameworks

JAX: Multi-Backend - NVIDIA GPUs via CUDA - Google TPUs (native support) - AMD GPUs via ROCm (improving) - CPU backend (often quite fast) - Most portable across accelerators

8.4.5 Ecosystem and Interoperability

Numba: Scientific Python Stack - Works well with NumPy arrays - Can use in SciPy workflows - Integrates with CuPy arrays via CUDA Array Interface - Limited ML framework integration

CuPy: Broad Interoperability - NumPy compatible (drop-in replacement) - SciPy compatible (cupy.scipy) - PyTorch integration via DLPack - TensorFlow integration via DLPack - Numba integration via CUDA Array Interface - Excellent with other GPU libraries

CUDA Python: Universal CUDA Integration - Works with any CUDA-based library - Can access memory from any framework - Useful for gluing different CUDA tools - Low-level integration layer

JAX: ML Ecosystem Focus - NumPy API compatibility (jax.numpy) - SciPy API compatibility (jax.scipy) - Integrates with Flax, Haiku, Optax (ML libraries) - TensorFlow/PyTorch interop via DLPack - Growing scientific computing ecosystem

8.4.6 Auto-Differentiation Capabilities

Numba: Not Built-In - No automatic differentiation support - Must implement gradients manually - Can use finite differences if needed - Focus is on forward computation

CuPy: Not Built-In - No native auto-diff - Can interface with frameworks that have it - Array operations only - Use with autodiff frameworks via interop

CUDA Python: Not Applicable - Low-level API wrapper - No computational graph concept - Would need to build on top - Not designed for auto-diff

JAX: Core Feature - `grad()`: automatic differentiation - `vjp()`: vector-Jacobian product - `jvp()`: Jacobian-vector product - Forward-mode and reverse-mode AD - Higher-order derivatives - Can differentiate through control flow - Best-in-class auto-diff capabilities

8.4.7 Debugging and Development Experience

Numba: Moderate Difficulty - Compilation errors can be cryptic - Limited support for print debugging in CUDA kernels - `NUMBA_DISABLE_JIT=1` for debugging without compilation - Cuda-memcheck for GPU errors - Decent error messages for type issues

CuPy: Easier Debugging - Standard Python debugging works - Clear error messages - Can test with NumPy first, then switch - Stack traces are readable - Good development experience

CUDA Python: Challenging - Manual error checking required - CUDA error codes to interpret - No automatic bounds checking - Easy to corrupt memory - Requires solid CUDA debugging skills

JAX: Moderate Difficulty - Tracing can be confusing initially - Side effects not allowed in jitted functions - Good error messages for common issues - `jax.debug.print()` for debugging jitted code - Strong typing helps catch errors

8.4.8 Performance Characteristics

Numba: Excellent for Custom Kernels - Performance matches hand-written CUDA C - You control optimization level - Can achieve theoretical maximum with effort - Performance depends on your kernel quality - Great for compute-bound operations

CuPy: Excellent for Standard Operations - Uses highly optimized vendor libraries - Hard to beat for matrix operations (cuBLAS) - Memory bandwidth often the bottleneck - Kernel fusion reduces overhead - Best when operations map to optimized libraries

CUDA Python: Maximum Performance Potential - No abstraction overhead - Full control over optimization - Can implement absolute fastest code - Requires expert-level optimization knowledge - Performance ceiling is highest

JAX: Excellent via XLA - XLA compiler does aggressive optimization - Often matches or beats hand-written code - Automatic kernel fusion - Excellent for complex computations - Performance improves with larger functions

8.4.9 Multi-GPU and Distributed Computing

Numba: Manual Multi-GPU - Must explicitly select devices - Manual data placement - Manual peer-to-peer transfers - Full control but requires effort - Good for custom multi-GPU algorithms

CuPy: Supported Multi-GPU - `cupy.cuda.Device()` context manager - Manual device selection - Straightforward multi-GPU patterns - Good for data-parallel workloads

CUDA Python: Complete Multi-GPU Control - Full access to P2P APIs - Unified virtual addressing - Custom scheduling across GPUs - Maximum flexibility - Most complex to implement

JAX: Built-In Parallelization - `pmap()`: automatic data parallelism across devices - `jit()`: can target specific devices - Sharding APIs for large arrays - Great for model parallelism - Easiest for distributed computing

8.5 Hybrid Approaches and Combinations

8.5.1 Numba + CuPy

Use Case: High-level operations with custom kernels - Use CuPy for standard array operations and memory management - Use Numba for custom kernels operating on CuPy arrays - CuPy handles memory, Numba handles custom logic

Example: Image processing pipeline using CuPy FFT and Numba custom filters

8.5.2 JAX + CuPy

Use Case: Auto-diff with specialized operations - Use JAX for gradient-based optimization - Use CuPy for specialized operations not in JAX - Transfer data via DLPack

Example: Physics simulation with JAX-based parameter optimization and CuPy-based specialized solvers

8.5.3 All Three (Numba + CuPy + JAX)

Use Case: Complex research workflows - JAX for ML model and auto-diff - CuPy for data preprocessing - Numba for specialized simulation kernels - Each tool for its strength

Example: Scientific ML project with custom physics simulation (Numba), data processing (CuPy), and neural network training (JAX)

8.6 Conclusion

There is no single "best" framework - each excels in different scenarios:

- **CuPy**: Best productivity for NumPy users, excellent for standard operations
- **Numba**: Best for custom algorithms and kernels in Python
- **CUDA Python**: Best for low-level control and CUDA integration
- **JAX**: Best for auto-differentiation and functional programming

Most successful projects use a combination:

- CuPy for high-level operations
- Numba for custom kernels
- JAX for gradient-based optimization
- CUDA Python for low-level integration when needed

Start with the highest-level tool that meets your needs (usually CuPy or JAX), then drop to lower levels only when necessary. This approach maximizes productivity while maintaining performance.

9 MPI + GPU

This document explains how to combine MPI (Message Passing Interface) with GPU computing for distributed parallel computing across multiple GPUs and nodes. We cover device mapping strategies and provide practical examples with Numba, CuPy, CUDA Python, and JAX.

9.1 Overview

MPI is the standard for distributed-memory parallel computing, allowing processes on different nodes to communicate. Combining MPI with GPU computing enables scaling beyond a single GPU to entire clusters of GPUs.

Key Concepts:

- **MPI Process (Rank)**: Independent process running on a CPU core
- **GPU Device**: Physical GPU that can be accessed by one or more MPI ranks
- **Device Mapping**: Assigning specific GPUs to specific MPI ranks
- **GPU-Aware MPI**: MPI implementations that can directly transfer GPU memory between nodes

Common Scenarios:

- Single node, multiple GPUs (1 rank per GPU)
- Multiple nodes, multiple GPUs per node (distributed computing)
- Multi-GPU training in machine learning
- Domain decomposition in scientific simulations

9.2 Why MPI + GPU?

Scaling Beyond Single GPU:

- Single GPU memory limited (8-80 GB)
- Problems requiring more compute power
- Data-parallel workloads across multiple GPUs
- Large-scale simulations requiring distributed memory

Typical Use Cases:

- Climate modeling across hundreds of GPUs
- Molecular dynamics simulations
- Large-scale machine learning (distributed training)
- Computational fluid dynamics on HPC clusters
- Big data processing pipelines

Performance Benefits:

- Near-linear scaling for well-suited problems
- Access to aggregate GPU memory across nodes
- Utilize entire HPC clusters efficiently
- Combine MPI parallelism with GPU parallelism

9.3 MPI Basics for GPU Computing

9.3.1 Essential MPI Concepts

MPI Communicator:

- Group of processes that can communicate
- `MPI.COMM_WORLD`: All processes in the job
- Used for collective operations and point-to-point communication

MPI Rank:

- Unique identifier for each process (0, 1, 2, …)
- Each rank typically controls one GPU
- Rank 0 often designated as “master” for I/O

MPI Size:

- Total number of processes in communicator
- Typically equals number of GPUs to use

Basic MPI Operations:

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD      # Communicator (all processes)
rank = comm.Get_rank()     # This process's rank (0, 1, 2, ...)
size = comm.Get_size()     # Total number of processes

print(f"Rank {rank} of {size}")
```

Installation:

```bash
# Install MPI implementation (choose one)
# Ubuntu/Debian
sudo apt-get install libopenmpi-dev openmpi-bin

# macOS
brew install open-mpi

# Install mpi4py (Python bindings)
pip install mpi4py
```

Running MPI Programs:

```bash
# Run with 4 processes (4 GPUs)
mpirun -np 4 python my_gpu_script.py

# On HPC cluster with SLURM
srun -n 4 python my_gpu_script.py
```

9.4 GPU Device Selection Strategies

9.4.1 Strategy 1: One Rank Per GPU (Most Common)

Assign each MPI rank to a unique GPU on the same node.

Principle:

- MPI rank 0 → GPU 0
- MPI rank 1 → GPU 1
- MPI rank 2 → GPU 2
- etc.

Code Pattern:

```python
from mpi4py import MPI
import os

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

# Method 1: Set CUDA_VISIBLE_DEVICES before importing GPU library
```

```
os.environ['CUDA_VISIBLE_DEVICES'] = str(rank)

# Now import GPU library - it sees only one GPU
import cupy as cp
# This rank sees GPU 0 (which is actually physical GPU {rank})
```

When to Use:

- Single node with multiple GPUs
- Each rank needs dedicated GPU resources
- Simplest approach for most use cases

9.4.2 Strategy 2: Manual Device Selection

Explicitly select GPU device in code after querying available devices.

Code Pattern:

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

# Query available GPUs
import cupy as cp
n_gpus = cp.cuda.runtime.getDeviceCount()

# Map rank to GPU (handle more ranks than GPUs)
gpu_id = rank % n_gpus

# Select this rank's GPU
cp.cuda.Device(gpu_id).use()

print(f"Rank {rank} using GPU {gpu_id}")
```

When to Use:

- More ranks than GPUs (oversubscription)
- Need explicit control over device selection
- Complex mapping schemes

9.4.3 Strategy 3: Multi-Node Mapping

For clusters with multiple nodes, each with multiple GPUs.

Principle:

- Node 0: Ranks 0-3 → GPUs 0-3
- Node 1: Ranks 4-7 → GPUs 0-3
- Node 2: Ranks 8-11 → GPUs 0-3
- etc.

Code Pattern:

```
from mpi4py import MPI
import os

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

# Get node-local rank (0-N on each node)
# This requires MPI implementation support
node_local_rank = int(os.environ.get('OMPI_COMM_WORLD_LOCAL_RANK', rank))

# Map to GPU on this node
import cupy as cp
n_gpus = cp.cuda.runtime.getDeviceCount()
gpu_id = node_local_rank % n_gpus

cp.cuda.Device(gpu_id).use()
print(f"Global rank {rank}, local rank {node_local_rank}, using GPU {gpu_id}")
```

Environment Variables for Local Rank:

- OpenMPI: `OMPI_COMM_WORLD_LOCAL_RANK`
- MPICH: `MPI_LOCALRANKID`
- SLURM: `SLURM_LOCALID`

When to Use:

- Multi-node HPC clusters
- Need to map ranks to node-local GPUs

9.5 GPU-Aware MPI

What is GPU-Aware MPI:

- MPI implementation can directly transfer GPU memory between nodes
- No explicit CPU staging required
- Uses RDMA (Remote Direct Memory Access) and GPUDirect
- Significant performance improvement for GPU-to-GPU communication

Benefits:

- Lower latency for GPU data transfers
- Higher bandwidth utilization
- Simplified code (no manual CPU staging)
- Reduced memory copies

Requirements:

- GPU-aware MPI implementation (OpenMPI 4.0+, MVAPICH2-GDR, etc.)
- InfiniBand or RoCE network with GPUDirect support
- CUDA-aware MPI compilation

Checking GPU-Aware MPI:

```
from mpi4py import MPI

# Check if MPI is CUDA-aware
print(f"CUDA-aware MPI: {MPI.CUDA_AWARE}")
```

Non-GPU-Aware vs GPU-Aware:

```
import cupy as cp
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

# GPU array
data_gpu = cp.random.randn(1000000)

if rank == 0:
    # Non-GPU-aware: Must transfer to CPU first
    data_cpu = cp.asnumpy(data_gpu)
    comm.Send(data_cpu, dest=1)
else:
    # Receive on CPU, transfer to GPU
    data_cpu = np.empty(1000000)
    comm.Recv(data_cpu, source=0)
    data_gpu = cp.asarray(data_cpu)

# GPU-aware: Direct GPU-to-GPU transfer
if rank == 0:
    comm.Send(data_gpu, dest=1)  # Send GPU array directly!
else:
    comm.Recv(data_gpu, source=0)  # Receive directly to GPU!
```

9.6 MPI + CuPy

CuPy works seamlessly with MPI for distributed GPU computing.

9.6.1 Basic Example: Distributed Array Operations

```
from mpi4py import MPI
import cupy as cp
import numpy as np

# Initialize MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()
```

```python
# Assign GPU to this rank
cp.cuda.Device(rank).use()

# Each rank creates local data on its GPU
local_size = 1000000 // size
local_data = cp.random.randn(local_size, dtype=cp.float32)

# Perform local computation
local_result = cp.sum(local_data ** 2)

# Gather results to rank 0
if rank == 0:
    results = np.empty(size, dtype=np.float32)
else:
    results = None

# Transfer GPU result to CPU for MPI communication
local_result_cpu = float(local_result.get())
comm.Gather(local_result_cpu, results, root=0)

if rank == 0:
    total = np.sum(results)
    print(f"Total sum of squares: {total}")
```

9.6.2 Distributed Matrix Multiplication

```python
from mpi4py import MPI
import cupy as cp

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

# Select GPU
cp.cuda.Device(rank).use()

# Matrix dimensions
N = 10000
M = N // size  # Each rank gets M rows

# Rank 0 creates and distributes matrix A
if rank == 0:
    A = cp.random.randn(N, N, dtype=cp.float32)
    A_splits = cp.split(A, size, axis=0)
else:
    A_splits = None
```

```
# Scatter rows of A to all ranks
local_A = comm.scatter(A_splits, root=0)
if rank != 0:
    local_A = cp.asarray(local_A)  # Convert to CuPy array

# All ranks need full B matrix (replicate)
if rank == 0:
    B = cp.random.randn(N, N, dtype=cp.float32)
    B_cpu = cp.asnumpy(B)
else:
    B_cpu = None

B_cpu = comm.bcast(B_cpu, root=0)
B = cp.asarray(B_cpu)

# Each rank computes its portion: C_local = A_local @ B
local_C = local_A @ B

# Gather results
C_parts = comm.gather(cp.asnumpy(local_C), root=0)

if rank == 0:
    C = cp.asarray(np.vstack(C_parts))
    print(f"Result shape: {C.shape}")
```

9.6.3 GPU-Aware MPI with CuPy

```
from mpi4py import MPI
import cupy as cp

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

cp.cuda.Device(rank).use()

# Create GPU data
data = cp.arange(1000000, dtype=cp.float32) * rank

if MPI.CUDA_AWARE:
    # GPU-aware: Direct GPU-to-GPU communication
    if rank == 0:
        # Send GPU array directly
        comm.Send(data, dest=1, tag=11)
    elif rank == 1:
        # Receive directly to GPU
```

```
        received = cp.empty_like(data)
        comm.Recv(received, source=0, tag=11)
        print(f"Rank 1 received data from rank 0: {received[:5]}")
else:
    # Non-GPU-aware: Stage through CPU
    if rank == 0:
        comm.Send(cp.asnumpy(data), dest=1, tag=11)
    elif rank == 1:
        data_cpu = np.empty(1000000, dtype=np.float32)
        comm.Recv(data_cpu, source=0, tag=11)
        received = cp.asarray(data_cpu)
```

9.7 MPI + Numba

Numba's CUDA support works well with MPI for custom GPU kernels in distributed settings.

9.7.1 Basic Example: Distributed Custom Kernel

```
from mpi4py import MPI
from numba import cuda
import numpy as np

# Initialize MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

# Select GPU for this rank
cuda.select_device(rank)

# Define CUDA kernel
@cuda.jit
def square_kernel(data):
    idx = cuda.grid(1)
    if idx < data.size:
        data[idx] = data[idx] ** 2

# Each rank processes local data
local_size = 1000000 // size
local_data = np.random.randn(local_size).astype(np.float32)

# Transfer to GPU
d_data = cuda.to_device(local_data)

# Launch kernel
threads_per_block = 256
```

```
blocks_per_grid = (local_size + threads_per_block - 1) // threads_per_block
square_kernel[blocks_per_grid, threads_per_block](d_data)

# Transfer back to CPU
result = d_data.copy_to_host()

# Compute local sum
local_sum = np.sum(result)

# Reduce across all ranks
total_sum = comm.reduce(local_sum, op=MPI.SUM, root=0)

if rank == 0:
    print(f"Total sum: {total_sum}")
```

9.7.2 Multi-GPU Monte Carlo Simulation

```
from mpi4py import MPI
from numba import cuda
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

# Select GPU
cuda.select_device(rank)

@cuda.jit
def monte_carlo_pi(n_samples, results):
    """Each thread generates samples and counts hits"""
    idx = cuda.grid(1)

    # Simple RNG (not cryptographically secure)
    seed = idx + rank * 1000000

    count = 0
    for i in range(n_samples):
        # Generate random point
        x = (seed * 1103515245 + 12345) % 2147483648 / 2147483648.0
        seed = (seed * 1103515245 + 12345) % 2147483648
        y = (seed * 1103515245 + 12345) % 2147483648 / 2147483648.0
        seed = (seed * 1103515245 + 12345) % 2147483648

        # Check if inside circle
        if x*x + y*y <= 1.0:
```

```
            count += 1

    results[idx] = count

# Each rank does local computation
threads = 1024
samples_per_thread = 10000

d_results = cuda.device_array(threads, dtype=np.int32)
monte_carlo_pi[1, threads](samples_per_thread, d_results)

# Transfer results back
results = d_results.copy_to_host()
local_hits = np.sum(results)
local_samples = threads * samples_per_thread

# Reduce across all ranks
total_hits = comm.reduce(local_hits, op=MPI.SUM, root=0)
total_samples = comm.reduce(local_samples, op=MPI.SUM, root=0)

if rank == 0:
    pi_estimate = 4.0 * total_hits / total_samples
    print(f"Pi estimate: {pi_estimate}")
    print(f"Total samples: {total_samples}")
```

9.8 MPI + JAX

JAX provides excellent support for multi-GPU computing through `pmap` and explicit device management.

9.8.1 Basic Example: Data Parallel with pmap

```
from mpi4py import MPI
import jax
import jax.numpy as jnp
from jax import pmap

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

# JAX sees all local GPUs
n_local_devices = jax.local_device_count()
print(f"Rank {rank}: {n_local_devices} local devices")

# Define computation
```

```
def compute(x):
    return jnp.sum(x ** 2)

# Create data on each device
local_data = jnp.ones((n_local_devices, 1000000))

# pmap automatically distributes across local devices
parallel_compute = pmap(compute)
local_results = parallel_compute(local_data)

# Sum local results
local_sum = jnp.sum(local_results)

# MPI reduce across nodes
total_sum = comm.reduce(float(local_sum), op=MPI.SUM, root=0)

if rank == 0:
    print(f"Total sum: {total_sum}")
```

9.8.2 Distributed Training Example

```
from mpi4py import MPI
import jax
import jax.numpy as jnp
from jax import grad, jit, pmap

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

# Model and loss function
def predict(params, x):
    return params['w'] @ x + params['b']

def loss(params, x, y):
    pred = predict(params, x)
    return jnp.mean((pred - y) ** 2)

# Gradient function
grad_fn = jit(grad(loss))

# Initialize parameters on each rank
key = jax.random.PRNGKey(rank)
params = {
    'w': jax.random.normal(key, (10, 100)),
    'b': jnp.zeros(10)
}
```

```
# Each rank has local training data
x_local = jax.random.normal(key, (1000, 100))
y_local = jax.random.normal(key, (1000, 10))

# Training loop
n_steps = 100
lr = 0.01

for step in range(n_steps):
    # Compute local gradients
    grads = grad_fn(params, x_local, y_local)

    # Average gradients across all ranks
    grads_w = comm.allreduce(grads['w'], op=MPI.SUM) / comm.Get_size()
    grads_b = comm.allreduce(grads['b'], op=MPI.SUM) / comm.Get_size()

    # Update parameters
    params['w'] -= lr * grads_w
    params['b'] -= lr * grads_b

    if rank == 0 and step % 10 == 0:
        current_loss = loss(params, x_local, y_local)
        print(f"Step {step}, Loss: {current_loss:.4f}")
```

9.8.3 Multi-Node pmap (Advanced)

```
from mpi4py import MPI
import jax
import jax.numpy as jnp
from jax.experimental import maps

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

# Configure JAX for multi-host
jax.distributed.initialize(
    coordinator_address=f"localhost:{8476 + rank}",
    num_processes=size,
    process_id=rank
)

# Now pmap can work across nodes
@jax.pmap
def distributed_compute(x):
    return x ** 2 + jnp.sum(x)
```

```
# Create data sharded across all devices (including remote)
n_devices = jax.device_count()  # Total across all nodes
data = jnp.ones((n_devices, 1000))

result = distributed_compute(data)
print(f"Rank {rank}: Result shape {result.shape}")
```

9.9 MPI + CUDA Python

CUDA Python provides low-level control for MPI + GPU scenarios.

9.9.1 Basic Example: Manual Device Management

```
from mpi4py import MPI
from cuda import cuda
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

# Initialize CUDA
err, = cuda.cuInit(0)
assert err == cuda.CUresult.CUDA_SUCCESS

# Get device for this rank
err, device = cuda.cuDeviceGet(rank)
assert err == cuda.CUresult.CUDA_SUCCESS

# Create context
err, context = cuda.cuCtxCreate(0, device)
assert err == cuda.CUresult.CUDA_SUCCESS

print(f"Rank {rank} using device {rank}")

# Allocate device memory
n = 1000000
nbytes = n * 4  # float32

err, d_data = cuda.cuMemAlloc(nbytes)
assert err == cuda.CUresult.CUDA_SUCCESS

# Create host data
h_data = np.random.randn(n).astype(np.float32)

# Copy host to device
err, = cuda.cuMemcpyHtoD(d_data, h_data.ctypes.data, nbytes)
```

```
assert err == cuda.CUresult.CUDA_SUCCESS

# Process on GPU (simplified - would load and run kernel here)

# Copy device to host
result = np.empty(n, dtype=np.float32)
err, = cuda.cuMemcpyDtoH(result.ctypes.data, d_data, nbytes)
assert err == cuda.CUresult.CUDA_SUCCESS

# Compute local sum
local_sum = np.sum(result)

# Reduce across ranks
total_sum = comm.reduce(local_sum, op=MPI.SUM, root=0)

if rank == 0:
    print(f"Total sum: {total_sum}")

# Cleanup
err, = cuda.cuMemFree(d_data)
err, = cuda.cuCtxDestroy(context)
```

9.9.2 GPU-Aware MPI with CUDA Python

```
from mpi4py import MPI
from cuda import cuda, cudart
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

# Initialize CUDA
err, = cudart.cudaSetDevice(rank)
assert err == cudart.cudaError_t.cudaSuccess

# Allocate device memory
n = 1000000
err, d_data = cudart.cudaMalloc(n * 4)
assert err == cudart.cudaError_t.cudaSuccess

# Initialize data on device
h_data = (np.arange(n) * rank).astype(np.float32)
err, = cudart.cudaMemcpy(d_data, h_data.ctypes.data, n * 4,
                         cudart.cudaMemcpyKind.cudaMemcpyHostToDevice)

if MPI.CUDA_AWARE and rank == 0:
```

```
    # Send GPU pointer directly
    comm.Send([d_data, MPI.FLOAT], dest=1, tag=11)
elif MPI.CUDA_AWARE and rank == 1:
    # Receive directly to GPU
    err, d_recv = cudart.cudaMalloc(n * 4)
    comm.Recv([d_recv, MPI.FLOAT], source=0, tag=11)

    # Copy back to verify
    result = np.empty(n, dtype=np.float32)
    err, = cudart.cudaMemcpy(result.ctypes.data, d_recv, n * 4,
                             cudart.cudaMemcpyKind.cudaMemcpyDeviceToHost)
    print(f"Rank 1 received: {result[:5]}")

    err, = cudart.cudaFree(d_recv)

# Cleanup
err, = cudart.cudaFree(d_data)
```

9.10 Best Practices

9.10.1 Device Selection

Always Select Device Early:

```
# Bad: Import before device selection
import cupy as cp
cp.cuda.Device(rank).use()  # May be too late

# Good: Select device first
from mpi4py import MPI
rank = MPI.COMM_WORLD.Get_rank()

import os
os.environ['CUDA_VISIBLE_DEVICES'] = str(rank)
import cupy as cp  # Now sees only assigned GPU
```

Check Device Assignment:

```
print(f"Rank {rank} on {socket.gethostname()}, GPU {cp.cuda.runtime.getDevice()}")
comm.Barrier()  # Synchronize before continuing
```

9.10.2 Communication Patterns

Minimize Communication:

```
# Bad: Communicate every iteration
for i in range(1000):
    local_result = compute()
    global_sum = comm.allreduce(local_result)  # Expensive!
```

```
# Good: Batch communications
local_results = []
for i in range(1000):
    local_results.append(compute())
global_sum = comm.allreduce(sum(local_results))  # Once
```

Use Collective Operations:

```
# Bad: Manual gathering
if rank == 0:
    results = [local_result]
    for i in range(1, size):
        results.append(comm.recv(source=i))

# Good: Use MPI collective
results = comm.gather(local_result, root=0)
```

Overlap Computation and Communication:

```
# Use non-blocking communication
req = comm.Isend(data, dest=target)
# Do other computation while sending
other_work()
req.Wait()  # Wait for send to complete
```

9.10.3 Memory Management

Reuse GPU Memory:

```
# Bad: Allocate every iteration
for i in range(100):
    temp = cp.empty(1000000)  # Allocate
    compute(temp)
    # temp deallocated

# Good: Allocate once
temp = cp.empty(1000000)
for i in range(100):
    compute(temp)  # Reuse allocation
```

Free Resources:

```
# Ensure cleanup on exit
try:
    # MPI + GPU computation
    pass
finally:
    # Free GPU memory
    del large_arrays
```

```
    cp.get_default_memory_pool().free_all_blocks()
```

9.10.4 Error Handling

Synchronize on Errors:

```
try:
    result = compute()
    error = 0
except Exception as e:
    print(f"Rank {rank} error: {e}")
    error = 1

# All ranks must agree on error state
error = comm.allreduce(error, op=MPI.MAX)
if error:
    comm.Abort(1)  # Terminate all ranks
```

9.10.5 Performance Monitoring

Time Communication vs Computation:

```
import time

# Computation time
t0 = time.time()
local_result = compute_on_gpu()
comp_time = time.time() - t0

# Communication time
t0 = time.time()
global_result = comm.allreduce(local_result)
comm_time = time.time() - t0

if rank == 0:
    print(f"Computation: {comp_time:.4f}s, Communication: {comm_time:.4f}s")
    print(f"Communication overhead: {100*comm_time/(comp_time+comm_time):.1f}%")
```

9.11 Common Patterns

9.11.1 Pattern 1: Domain Decomposition

Divide problem domain across GPUs, communicate boundaries.

```
# Each rank owns a slice of the domain
nx_global = 10000
nx_local = nx_global // size

# Include ghost cells for boundaries
```

```
data = cp.zeros(nx_local + 2)  # +2 for left/right ghosts

# Exchange boundaries with neighbors
left_neighbor = (rank - 1) % size
right_neighbor = (rank + 1) % size

# Send right boundary to right neighbor, receive into left ghost
comm.Sendrecv(data[-2], dest=right_neighbor,
              recvbuf=data[0], source=left_neighbor)

# Send left boundary to left neighbor, receive into right ghost
comm.Sendrecv(data[1], dest=left_neighbor,
              recvbuf=data[-1], source=right_neighbor)
```

9.11.2 Pattern 2: Parallel Reduction

Each rank computes partial result, combine with reduction.

```
# Each rank computes local partial result
local_result = compute_local()

# Reduce to get global result
global_result = comm.allreduce(local_result, op=MPI.SUM)

# All ranks now have global_result
```

9.11.3 Pattern 3: Scatter-Compute-Gather

Distribute data, compute, collect results.

```
if rank == 0:
    # Create global data
    global_data = cp.random.randn(size * 1000000)
    splits = cp.split(global_data, size)
else:
    splits = None

# Scatter to all ranks
local_data = comm.scatter(splits, root=0)
if rank != 0:
    local_data = cp.asarray(local_data)

# Each rank computes
local_result = compute(local_data)

# Gather results
results = comm.gather(cp.asnumpy(local_result), root=0)
```

```
if rank == 0:
    final = cp.concatenate([cp.asarray(r) for r in results])
```

9.12 Summary

Key Takeaways:

1. **Device Selection**: Always map MPI ranks to GPUs explicitly
2. **Communication**: Minimize and batch MPI communication
3. **GPU-Aware MPI**: Use when available for direct GPU-to-GPU transfers
4. **Framework Choice**:
   - CuPy: Easiest for array-based computations
   - Numba: Best for custom kernels
   - JAX: Excellent for ML with `pmap`
   - CUDA Python: Maximum control, most complex
5. **Scaling**: MPI + GPU enables scaling to hundreds/thousands of GPUs

10 References

- CUDA C++ Programming Guide: `https://docs.nvidia.com/cuda/cuda-c-programming-guide`
- Numba: `https://numba.pydata.org`
- CuPy: `https://cupy.dev`
- CUDA Python: `https://nvidia.github.io/cuda-python/latest`
- JAX: `https://docs.jax.dev/en/latest`