ACCELERATED PYTHON: The Numba & CUDA Handbook

Comprehensive Guide: From Numba Essentials & Series Simulations to Mathematical Computation & Capstone Project.

Contents

Ultimate CUDA Programming with Python & Numba

A complete, from-first-principles to advanced, project-driven book

---

Phase 0 — Scope, Philosophy, and Non-Negotiables

This phase locks the **intent, scope, and quality bar** of the book. Nothing technical is taught here. This exists to prevent shallow chapters, toy projects, and incoherent depth later.

This book is written with one core assumption: > **GPU programming is a way of thinking, not a syntax skill.**

0.1 What This Book Is

- A **full CUDA–Numba systems book**, not a tutorial series
- Mathematics-first where required
- Architecture-aware throughout
- Project-driven, but never project-rushed

Every chapter must satisfy at least one of the following: - Build a mental model - Prove correctness - Demonstrate a real limitation

---

0.2 What This Book Is NOT

- Not a cheat sheet
- Not a NumPy-on-GPU trick book
- Not CUDA C++ rewritten in Python
- Not focused on benchmarks for their own sake

If a concept can be done better on CPU, **we explicitly say so**.

0.3 Technology Lock

These choices are deliberate and final.

Programming Language - Python 3.9+

GPU Interface - Numba CUDA only

CPU Stack - NumPy - Standard Python data structures

Visualization (CPU-side only) - matplotlib (no GPU rendering shortcuts)

No CUDA C++, no PyCUDA, no cuBLAS magic until the very end.

0.4 Mathematical Scope

Mathematics is treated as a first-class dependency, not an appendix.

Included explicitly: - Floating-point arithmetic behavior - Parallel reductions - Complex numbers and series - Coordinate geometry - Numerical stability

Excluded deliberately: - Symbolic algebra - Auto-differentiation frameworks

---

0.5 Project Quality Bar (Non-Negotiable)

Every major project must: 1. Have a **CPU baseline** 2. Have a **naive GPU version** 3. Have an **optimized GPU version** 4. Include a **correctness check** 5. Include **performance reasoning** (not just numbers)

If a project cannot meet all five, it does not belong in this book.

0.6 Project Count and Classification

This book includes **10 major projects**, each designed to be non-trivial and system-level.

Core Projects (Math & Geometry) 1. Mandelbrot & Julia Set Engine 2. Parametric & Lissajous Curve Generator 3. Affine Geometry Transformation Engine

Simulation Projects 4. N-Body Gravitational Simulation 5. GPU-Based Numerical PDE Solver

Data & Algorithms Projects 6. GPU Histogram & Statistical Engine 7. Parallel Graph Traversal (BFS)

Systems & Applied Projects 8. Image Convolution & Edge Detection Engine 9. Real-Time Signal Processing Pipeline 10. End-to-End Scientific Compute Engine

Each project is expected to span multiple sections and evolve over time.

0.7 Chapter Dependency Rules

- No optimization before correctness
- No advanced memory tricks before memory models
- No simulations before numerical stability

This book is intentionally **linear**. Skipping chapters is discouraged.

0.8 Coding Standards

All code in this book follows these rules: - Explicit variable names - No hidden magic - No one-line cleverness - Every kernel has bounds checks - Every GPU result is validated on CPU

Performance comes **after clarity**, never before.

0.9 How to Read This Book

You are expected to: - Type the code - Break it - Measure it - Fix it

Reading without execution defeats the purpose.

Phase 0 Complete

At this point: - Scope is frozen - Quality bar is locked - Project expectations are explicit

Only now do we begin technical work.

Table of Contents

---

1. Introduction

This book is not a handbook. It is a **full system-level guide** to GPU computing using **CUDA with Python via Numba**. You will build intuition, mathematical grounding, architectural understanding, and real systems.

You are expected to know: - Python basics - NumPy - Basic linear algebra

You are *not* expected to know CUDA beforehand.

---

2. Parallel Computing Fundamentals

Why GPUs Exist

CPUs optimize for: - Low latency - Complex control flow

GPUs optimize for: - Massive throughput - Simple repeated computation

SIMD and SIMT

- SIMD: Single Instruction Multiple Data
- SIMT: Single Instruction Multiple Threads (CUDA model)

Every CUDA thread executes the same kernel but on different data.

---

3. CUDA Architecture Explained Clearly

Core Components

- SM (Streaming Multiprocessor)
- Warp (32 threads)
- Global Memory
- Shared Memory
- Registers

Execution Model

Threads → Blocks → Grid

```
i = cuda.grid(1)
```

This gives a **global linear thread index**.

---

4. Python, Numba, and the CUDA Ecosystem

Numba is a **JIT compiler**, not a wrapper.

```
@cuda.jit
def kernel(a):
    i = cuda.grid(1)
    if i < a.size:
        a[i] *= 2
```

Numba translates this to PTX at runtime.

---

5. Environment Setup

Required Stack

- NVIDIA GPU
- NVIDIA Driver
- CUDA Toolkit
- Python 3.9+
- Numba

Verification:

```
from numba import cuda
print(cuda.is_available())
```

---

6. Threads, Blocks, and Grids

1D Kernel

```
@cuda.jit
def add(a, b, c):
    i = cuda.grid(1)
    if i < a.size:
        c[i] = a[i] + b[i]
```

Launch Configuration

```
threads = 256
blocks = (N + threads - 1) // threads
```

---

7. CUDA Memory Model

Global Memory

- Large
- Slow

Shared Memory

```
shared = cuda.shared.array(shape=256, dtype=float32)
```

- Per block
- Extremely fast

Registers

- Per thread
- Fastest

---

8. Writing Efficient Kernels

Rules: - Minimize global memory access - Coalesce memory reads - Avoid branching - Use shared memory

---

9. Debugging and Profiling

Debugging

- Cannot print from kernels
- Use assertions sparingly

Profiling

```
cuda.profile_start()
# kernel call
cuda.profile_stop()
```

---

10. Mathematical Foundations

Parallel Reduction

Used in: - Sum - Mean - Variance

GPU reduction is hierarchical.

---

11. Complex Numbers and Series on GPU

Complex Arithmetic

```
@cuda.jit
def complex_add(a_real, a_imag, b_real, b_imag, out_r, out_i):
    i = cuda.grid(1)
    if i < out_r.size:
        out_r[i] = a_real[i] + b_real[i]
        out_i[i] = a_imag[i] + b_imag[i]
```

Project 1: Mandelbrot Set (GPU)

- Complex iteration
- Escape-time algorithm

- GPU acceleration

---

12. Coordinate Geometry on CUDA

Vector Operations

```
@cuda.jit
def translate(x, y, dx, dy):
    i = cuda.grid(1)
    if i < x.size:
        x[i] += dx
        y[i] += dy
```

Project 2: GPU Affine Transform Engine

---

13. Geometry Plotting Pipeline

CUDA computes geometry, CPU plots.

Project 3: GPU-Generated Lissajous Curves

---

14. Simulations

Physics Example: Particle Motion

```
@cuda.jit
def update(pos, vel, dt):
    i = cuda.grid(1)
    if i < pos.shape[0]:
        pos[i] += vel[i] * dt
```

Project 4: N-Body Gravity Simulation

---

15. Data Structures on GPU

- Arrays
- Prefix sums
- Histogram

Project 5: GPU Histogram Engine

---

16. Hybrid CPU–GPU Design

GPU: - Heavy computation

CPU: - Logic - I/O

Project 6: Real-Time Signal Processing Pipeline

---

17. Optimization Techniques

- Occupancy tuning
- Loop unrolling
- Memory alignment

---

18. Multi-Dimensional Kernels

```
x, y = cuda.grid(2)
```

Project 7: GPU Image Convolution Engine

---

19. Advanced CUDA Features

- Atomic operations
- Streams
- Unified memory

Project 8: GPU Parallel BFS

---

20. Production Systems

- Error handling
- Fallback to CPU
- Testing GPU code

Project 9: GPU-Accelerated Numerical Solver

Project 10: Full Scientific Compute Engine

---

21. Complete Project Portfolio (Detailed)

This section fully specifies all major projects promised in this book. Each project is designed to integrate **CUDA kernels + Python control logic + data structures + mathematics**.

Project 1: GPU Mandelbrot & Julia Set Engine

Concepts covered: - Complex numbers on GPU - Escape-time algorithms - Memory coalescing

Deliverables: - CUDA kernel for iteration - Python visualization using matplotlib - CPU vs GPU performance comparison

---

Project 2: GPU Affine & Rigid Body Geometry Engine

Concepts covered: - Coordinate geometry - Matrix multiplication on GPU - Transform pipelines

Features: - Translation, rotation, scaling - Batch geometry processing

---

Project 3: GPU-Generated Parametric & Lissajous Curves

Concepts covered: - Trigonometric series - Parametric equations - Parallel sampling

---

Project 4: N-Body Gravitational Simulation

Concepts covered: - Newtonian physics - Numerical integration (Euler, Verlet) - $O(N^2)$ parallel force computation

---

Project 5: GPU Histogram & Statistical Engine

Concepts covered: - Atomic operations - Parallel reductions - Probability distributions

---

Project 6: Real-Time Signal Processing Pipeline

Concepts covered: - FFT-style thinking (without cuFFT) - Sliding window computation - Hybrid CPU–GPU architecture

---

Project 7: GPU Image Convolution & Edge Detection

Concepts covered: - 2D kernels - Shared memory tiling - Convolution mathematics

---

Project 8: Parallel Graph Algorithms on GPU

Concepts covered: - BFS traversal - Prefix sums - Frontier-based expansion

---

Project 9: GPU Numerical PDE Solver

Concepts covered: - Finite difference methods - Stability conditions - Time-stepping schemes

---

Project 10: End-to-End Scientific Compute Engine

Concepts covered: - Modular kernel design - CPU fallback paths - Testing GPU correctness

---

22. Exercises & Thought Problems

Each chapter should be paired with: - Conceptual questions - Small kernel-writing tasks - Performance experiments

---

23. CUDA Mental Models (How Experts Think)

- Think in warps, not threads
- Memory access dominates speed
- Math is cheap, memory is expensive

---

24. Where to Go Next

- CUDA C++ transition
- cuBLAS and cuFFT
- Multi-GPU systems

Final Words

This book is intentionally deep. If you complete it fully, you will not merely *use* CUDA. You will **reason in parallel**, design GPU-first algorithms, and build serious compute systems.

End of Book

Phase 1 — CUDA Core Foundations

This phase builds the mental models required to reason about GPUs correctly. We deliberately avoid optimization tricks until correctness and architecture are understood.

Chapter 1 — Parallel Computing Fundamentals (CPU vs GPU)

This chapter establishes *how* parallel machines think and *why* GPUs exist. No CUDA syntax yet. Only models and consequences.

1.1 Serial Thinking and Its Limits

Most Python code is written assuming a single instruction stream:

```python
# Serial loop
s = 0.0
for i in range(len(a)):
    s += a[i]
```

This model assumes: - One instruction at a time - Strong ordering - Low overhead per operation

It scales poorly because the loop body is fundamentally sequential.

1.2 Data Parallelism

Data parallelism applies the *same operation* to many independent elements.

```python
# Data-parallel intent
for i in range(N):
    out[i] = f(inp[i])
```

Key property: > Each iteration is independent.

This independence is the gateway to parallel execution.

1.3 CPU Parallelism Model

CPUs achieve parallelism through: - Few powerful cores - Deep caches - Branch prediction - Out-of-order execution

Python parallelism options: - Multiprocessing - Vectorization (NumPy)

Limitations: - High thread management cost - Memory bandwidth bottlenecks

---

1.4 GPU Parallelism Model

GPUs assume: - Millions of lightweight threads - Minimal control flow - Massive memory latency hidden by concurrency

Instead of speeding up one thread, GPUs run *many* threads.

Consequence: > Latency is tolerated, not avoided.

1.5 SIMD vs SIMT

- **SIMD**: One instruction, multiple data lanes
- **SIMT (CUDA)**: One instruction, multiple *threads*

Threads are grouped into **warps** (typically 32 threads).

All threads in a warp: - Execute the same instruction - Diverge only via masking

---

1.6 Control Flow and Divergence

Consider:

```
if x[i] > 0:
    y[i] = x[i]
else:
    y[i] = -x[i]
```

On GPU: - Both branches execute - Threads are selectively masked

Result: - Divergence reduces throughput

Rule: > Simple control flow scales. Complex branching does not.

1.7 Memory as the True Bottleneck

Arithmetic is cheap. Memory access is expensive.

Rough intuition: - Floating-point add: ~1 cycle - Global memory load: hundreds of cycles

GPUs survive this by running other warps while one waits.

1.8 A First Parallel Thought Experiment

Problem: Square one million numbers.

CPU approach:

```
for i in range(N):
    out[i] = a[i] * a[i]
```

GPU mindset: > Launch one thread per element.

This simple mapping drives nearly all GPU algorithms.

1.9 When NOT to Use a GPU

Do not use GPU when: - N is small - Logic dominates arithmetic - Data transfer outweighs compute

GPU is not a universal accelerator.

1.10 Chapter Summary

You should now understand: - Why GPUs exist - What data parallelism means - Why control flow and memory matter more than math

No CUDA code yet. That begins in Chapter 2.

End of Phase 1 (Part 1)

Next file: **Phase 1 – Chapter 2: CUDA Execution Model & Hardware Mental Models** # Phase 1 — Chapter 2

CUDA Execution Model & Hardware Mental Models

This chapter introduces **how CUDA actually runs your code on the GPU**. The goal is not syntax mastery but a correct *mental picture* of execution. If this picture is wrong, every optimization later will be guesswork.

2.1 The GPU as a Throughput Machine

A GPU is not a faster CPU. It is a different machine.

CPU philosophy: - Finish one task quickly - Minimize latency - Complex control logic

GPU philosophy: - Run thousands of tasks together - Hide latency with parallelism - Simple control, massive data flow

Key idea: > GPUs trade single-thread speed for massive concurrency.

2.2 Streaming Multiprocessors (SMs)

The GPU is divided into **Streaming Multiprocessors (SMs)**.

Each SM contains: - Many CUDA cores - Register files - Shared memory - Warp schedulers

Think of an SM as a *small parallel factory* that executes groups of threads.

Important: - Kernels do not run on the whole GPU at once - They are scheduled SM by SM

2.3 Threads: The Smallest Execution Unit

A **thread** executes: - One instance of your kernel code - On its own data index

Threads: - Are extremely lightweight - Have their own registers - Cannot be synchronized globally

CUDA assumes **millions of threads**.

2.4 Warps: The Real Execution Unit

Threads are grouped into **warps**.

Facts: - A warp contains **32 threads** (on modern NVIDIA GPUs) - A warp executes **one instruction at a time** - Threads may be masked on branches

This means: > The warp, not the thread, is the real execution unit.

2.5 Warp Scheduling and Latency Hiding

When one warp stalls (for example, waiting on memory): - The SM immediately switches to another ready warp

This is how GPUs hide memory latency.

Consequence: - You need *many* active warps - Idle threads waste hardware

---

2.6 Thread Blocks

Threads are grouped into **blocks**.

Properties of a block: - All threads in a block run on the same SM - Threads in a block can: - Synchronize - Share shared memory

Threads in different blocks: - Cannot synchronize directly - May run on different SMs

---

2.7 The Grid

A **grid** is the collection of all blocks launched for a kernel.

When you launch a kernel, you specify: - Number of blocks - Threads per block

Example (conceptual): - 1D grid - 256 threads per block - Many blocks

The grid defines *how much parallel work exists*, not how it is scheduled.

2.8 Execution Is Not Sequential

Important mental correction:

- Blocks may execute in any order
- Warps may interleave unpredictably
- No assumption of global ordering is valid

Rule: > If correctness depends on execution order, the kernel is wrong.

2.9 Implications for Algorithm Design

You must design algorithms that: - Are order-independent - Use synchronization only within blocks - Avoid global state assumptions

This is why reductions, scans, and simulations are hard on GPUs.

2.10 What You Should Now Be Able to Visualize

You should be able to imagine: - A kernel launch creating a grid - Blocks being assigned to SMs - Warps executing in lockstep - Latency hidden by warp switching

No code yet. That begins in the next chapter.

Chapter Summary

In this chapter, you learned: - What SMs, warps, blocks, and grids are - Why warps matter more than threads - Why execution order cannot be trusted

This mental model is mandatory for writing correct CUDA code.

End of Chapter 2

Next file: **Phase 1 – Chapter 3: Threads, Blocks, Grids in Practice (First CUDA Kernels)** # Phase 1 — Chapter 3

Threads, Blocks, and Grids in Practice

This chapter is where CUDA programming actually begins. We will write **correct, minimal, and inspectable kernels**. Performance tuning is explicitly postponed. The only goal here is correctness and clear indexing.

3.1 Your First CUDA Kernel (Mental Checklist)

Before writing any CUDA kernel, you must answer:

1. What does **one thread** compute?
2. How do threads map to data?
3. What happens when threads exceed data size?

If any of these are unclear, the kernel is wrong.

3.2 Importing CUDA in Python

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

Before continuing, always verify:

```
print(cuda.is_available())
```

If this prints `False`, **stop here**. Fix your CUDA setup first.

3.3 Anatomy of a CUDA Kernel

A CUDA kernel in Numba:

```
@cuda.jit
def kernel_name(...):
    # device code
```

Important rules: - Runs on the GPU - Cannot use Python objects - No printing - Limited standard library

Think of kernels as **restricted Python**.

3.4 Computing the Global Thread Index

The most important line in basic CUDA:

```
i = cuda.grid(1)
```

This computes:

```
i = blockIdx.x * blockDim.x + threadIdx.x
```

Every thread gets a unique global index.

3.5 The Mandatory Bounds Check

Threads are launched in fixed blocks. Some threads may not correspond to valid data.

Correct pattern:

```
@cuda.jit
def square_kernel(a, out):
    i = cuda.grid(1)
    if i < a.size:
        out[i] = a[i] * a[i]
```

Never omit the bounds check.

3.6 Launch Configuration

Kernel launches use square-bracket syntax:

```
threads_per_block = 256
blocks_per_grid = (N + threads_per_block - 1) // threads_per_block

square_kernel[blocks_per_grid, threads_per_block](a_d, out_d)
```

Interpretation: - Threads per block: how many threads cooperate - Blocks per grid: how many blocks are launched

---

3.7 Complete Minimal Example

```
import numpy as np
from numba import cuda

@cuda.jit
def square_kernel(a, out):
    i = cuda.grid(1)
    if i < a.size:
        out[i] = a[i] * a[i]

N = 1_000_000
a = np.arange(N, dtype=np.float32)
out = np.zeros_like(a)

a_d = cuda.to_device(a)
out_d = cuda.to_device(out)

threads = 256
blocks = (N + threads - 1) // threads

square_kernel[blocks, threads](a_d, out_d)
out = out_d.copy_to_host()
```

This is the canonical CUDA–Numba kernel structure.

3.8 CPU Baseline (Always Required)

```
out_cpu = a * a
```

Validation:

```
assert np.allclose(out, out_cpu)
```

Every GPU kernel must be validated on CPU.

3.9 Common Beginner Errors

Forgetting bounds checks
Using Python lists inside kernels
Assuming threads run in order
Launching too few threads
Debugging with print statements

3.10 Thinking in Threads

Correct mindset:

> One thread = one data element

If you try to make one thread do too much work, you are fighting the GPU.

Chapter Summary

You can now: - Write a correct CUDA kernel - Launch it safely - Validate results - Reason about indexing

You have not optimized anything yet. That is intentional.

End of Chapter 3

Next file: **Phase 1 – Chapter 4: CUDA Memory Model (Global, Shared, Registers)**

Phase 1 — Chapter 4

The CUDA Memory Model (Global, Shared, Registers)

This chapter explains **why GPU performance lives or dies by memory access**. Most slow CUDA programs are correct programs with bad memory behavior. This chapter fixes that.

4.1 The Fundamental Truth

> On a GPU, **math is cheap**. **Memory access is expensive**.

A kernel that performs extra arithmetic but reduces memory access will almost always be faster.

4.2 Types of Memory in CUDA

CUDA exposes several memory spaces. We focus on the ones you will use directly.

Memory Type	Scope	Speed	Lifetime
Registers	Thread	Fastest	Kernel
Shared	Block	Very fast	Kernel
Global	Grid	Slow	Program

Understanding *who can see what* is critical.

4.3 Registers (Thread-Local Memory)

Registers: - Belong to a single thread - Are allocated automatically - Are extremely fast

Example:

```
@cuda.jit
def reg_example(a, out):
    i = cuda.grid(1)
    if i < a.size:
        temp = a[i] * 2.0   # temp lives in a register
        out[i] = temp
```

Every local scalar variable becomes a register.

Caution: > Too many registers per thread reduces parallelism.

4.4 Global Memory

Global memory: - Accessible by all threads - High latency - Very large

Example:

```
val = a[i]  # global memory load
```

Latency is hidden by: - Running many warps - Switching warps while one waits

---

4.5 Memory Coalescing

Global memory is fastest when: - Threads in a warp access **contiguous memory**

Good access pattern:

```
# Thread i reads a[i]
```

Bad access pattern:

```
# Thread i reads a[i * stride]
```

Uncoalesced access wastes bandwidth.

4.6 Shared Memory

Shared memory: - Shared by threads in a block - Much faster than global memory - Programmer-managed

Declared as:

```
from numba import float32

shared = cuda.shared.array(shape=256, dtype=float32)
```

All threads in the block can read/write this array.

4.7 Synchronization Within a Block

When using shared memory, synchronization is required:

```
cuda.syncthreads()
```

Rule: > Every thread in the block must reach `syncthreads()`.

4.8 Example: Naive vs Shared Memory Kernel

Naive Version (Global Memory Only)

```
@cuda.jit
def naive_double(a, out):
    i = cuda.grid(1)
    if i < a.size:
        out[i] = 2.0 * a[i]
```

Shared Memory Version

```
@cuda.jit
def shared_double(a, out):
    shared = cuda.shared.array(256, dtype=float32)

    tid = cuda.threadIdx.x
    i = cuda.grid(1)

    if i < a.size:
        shared[tid] = a[i]
    cuda.syncthreads()

    if i < a.size:
        out[i] = 2.0 * shared[tid]
```

This pattern becomes critical in reductions and convolutions.

4.9 When Shared Memory Helps (and When It Does Not)

Shared memory helps when: - Data is reused by multiple threads - Global loads can be reduced

It does NOT help when: - Each thread reads unique data once

Misusing shared memory can make code slower.

4.10 Registers vs Shared vs Global: Design Rules

Design rules: 1. Keep frequently used scalars in registers 2. Use shared memory for block-level reuse 3. Minimize global memory traffic

Always reason about memory first.

Chapter Summary

You now understand: - Why memory dominates performance - How registers, shared, and global memory differ - When shared memory is beneficial - Why coalesced access matters

This chapter explains 80% of CUDA performance behavior.

End of Chapter 4

Next file: **Phase 1 – Chapter 5: Debugging, Correctness, and Validation on GPU**

Phase 1 — Chapter 5

Debugging, Correctness, and Validation on the GPU

GPU programs often **fail silently**. A kernel may run, return results, and still be wrong. This chapter teaches disciplined methods to detect, isolate, and fix errors in CUDA–Numba programs.

5.1 Why GPU Debugging Is Hard

Key reasons: - No `print()` inside kernels - Thousands of threads fail simultaneously - Execution order is undefined - Memory errors may not crash immediately

Rule: > If you do not validate GPU results, assume they are wrong.

5.2 The Golden Rule: CPU Baseline First

Every GPU kernel must have a **CPU reference implementation**.

Example:

```
# CPU baseline
out_cpu = a * a
```

GPU output is meaningless without comparison.

5.3 Deterministic Test Inputs

Always start with: - Small arrays - Known values - Simple patterns

Example:

```
a = np.array([0, 1, 2, 3, 4], dtype=np.float32)
```

Random data hides bugs.

5.4 Validation Patterns

Exact Comparison (Integers)

```
assert np.array_equal(out_gpu, out_cpu)
```

Tolerant Comparison (Floats)

```
assert np.allclose(out_gpu, out_cpu, rtol=1e-5, atol=1e-6)
```

Never use == for floating-point results.

5.5 Detecting Out-of-Bounds Errors

Out-of-bounds writes may: - Corrupt memory - Produce random errors - Appear to work

Protection pattern:

```
if i < a.size:
    out[i] = a[i]
```

This check is mandatory.

5.6 Using Assertions in Kernels (Carefully)

Numba allows limited assertions:

```
@cuda.jit
def kernel(a):
    i = cuda.grid(1)
    assert i >= 0
```

Notes: - Assertions slow kernels - Use only during debugging

5.7 Isolating Errors

Recommended workflow: 1. Reduce input size 2. Use one block 3. Use one warp 4. Compare step-by-step

Example:

```
threads = 32
blocks = 1
```

This removes scheduling complexity.

5.8 Floating-Point Surprises

GPU floating-point math: - Is not associative - May reorder operations - Differs from CPU results

Example:

```
(a + b) + c != a + (b + c)
```

Small numerical differences are expected.

5.9 Synchronization Bugs

Incorrect synchronization leads to: - Race conditions - Nondeterministic output

Rule: > If shared memory is used, `cuda.syncthreads()` is not optional.

5.10 Common Silent Failure Modes

- Forgetting bounds checks
- Assuming block execution order
- Reading uninitialized shared memory
- Ignoring floating-point tolerance

Chapter Summary

You can now: - Validate GPU results correctly - Detect silent failures - Debug kernels methodically - Trust your outputs

Correctness comes before speed.

End of Chapter 5

Next file: **Phase 1 – Chapter 6: Performance Fundamentals and Measurement**

Phase 1 — Chapter 6

Performance Fundamentals and Measurement

This chapter teaches **how to think about GPU performance correctly**. The goal is not to chase numbers, but to understand *why* a kernel is fast or slow. Premature optimization is avoided; disciplined measurement is required.

6.1 Why Benchmarks Lie

Common mistakes: - Measuring tiny problem sizes - Ignoring data transfer time - Comparing against unoptimized CPU code - Timing a single run

Rule: > If you cannot explain the result, the benchmark is meaningless.

6.2 What Actually Costs Time on a GPU

Primary contributors: 1. Host Device memory transfers 2. Global memory access inside kernels 3. Kernel launch overhead 4. Synchronization

Arithmetic is rarely the bottleneck.

6.3 Measuring Time Correctly

CPU Timing

```
import time

t0 = time.perf_counter()
# CPU computation
t1 = time.perf_counter()
print(t1 - t0)
```

GPU Timing (Important)

GPU launches are asynchronous. You **must synchronize**.

```
from numba import cuda
import time

start = time.perf_counter()
kernel[blocks, threads](a_d, out_d)
```

```
cuda.synchronize()
end = time.perf_counter()

print(end - start)
```

Without `cuda.synchronize()`, timings are wrong.

6.4 Warm-Up Runs

The first kernel launch includes: - JIT compilation - Context creation

Always do warm-up runs:

```
kernel[blocks, threads](a_d, out_d)
cuda.synchronize()
```

Never time the first run.

6.5 Throughput vs Latency

GPU performance should be evaluated by: - Elements processed per second - Not time per element

Example metric:

```
million elements / second
```

GPUs excel at throughput, not single-operation latency.

6.6 Memory-Bound vs Compute-Bound Kernels

A kernel is: - **Memory-bound** if performance is limited by memory access - **Compute-bound** if limited by arithmetic

Most beginner kernels are memory-bound.

Design implication: > Reduce memory traffic before reducing math.

6.7 A Controlled Performance Experiment

Experiment: Squaring Numbers

Steps: 1. CPU baseline 2. Naive GPU kernel 3. Measure correctly

CPU:

```
out_cpu = a * a
```

GPU:

```
square_kernel[blocks, threads](a_d, out_d)
cuda.synchronize()
```

Interpret results, not just numbers.

6.8 When GPU Will Lose

GPU loses when: - N is small - Kernel is trivial - Data transfer dominates

This is expected and acceptable.

6.9 Optimization Comes Later

Do not: - Tune block sizes blindly - Use shared memory without reuse - Obsess over occupancy early

First ask: > Where is time actually going?

Chapter Summary

You now understand: - How to measure GPU performance correctly - Why synchronization matters - Why many benchmarks are misleading - How to reason about performance limits

You are now ready to move beyond fundamentals.

End of Chapter 6

Phase 1 is now complete.

Next phase: **Phase 2 — Mathematical Computing on the GPU** # Phase 2 — Chapter 7

Mathematical Foundations for GPU Computing

This chapter lays the **mathematical groundwork** required for serious GPU work. GPUs amplify both correctness and error. If the math is poorly structured, the GPU will fail faster and more convincingly.

7.1 Why Math Must Be Revisited for GPUs

On CPUs, numerical mistakes often hide behind: - Low parallelism - Deterministic ordering - Higher precision in intermediate steps

On GPUs: - Operations are massively parallel - Ordering is not guaranteed - Floating-point behavior is exposed

Rule: > Parallelism changes the math, not just the speed.

7.2 Associativity Is Broken

Floating-point addition is not associative:

```
(a + b) + c   a + (b + c)
```

In parallel reductions, this matters.

CPU loop:

```
s = 0.0
for x in a:
    s += x
```

GPU reduction: - Adds values in a tree-like pattern - Changes accumulation order

Results differ slightly. This is normal.

7.3 Parallel Reduction Concept

Parallel reduction computes: - Sum - Min / Max - Dot products

Conceptual steps: 1. Each thread computes partial results 2. Results are combined hierarchically

GPU reductions are **logarithmic**, not linear.

7.4 Numerical Stability

Unstable operations amplify error: - Subtracting nearly equal numbers - Accumulating small values repeatedly

Stable alternatives: - Kahan summation (conceptual) - Pairwise reduction

On GPUs, pairwise reduction is preferred.

7.5 Scaling and Normalization

Always scale inputs when possible.

Example: - Normalize vectors before dot products - Scale coordinates before geometry transforms

This reduces floating-point error accumulation.

7.6 Complex Numbers on the GPU

Python complex numbers are **not supported** inside CUDA kernels.

Correct representation: - Real and imaginary parts stored separately

```
# z = a + ib
z_real[i]
z_imag[i]
```

All complex operations must be implemented manually.

7.7 Complex Arithmetic Rules (GPU-Safe)

Addition:

```
(a + ib) + (c + id) = (a + c) + i(b + d)
```

Multiplication:

```
(a + ib)(c + id) = (ac − bd) + i(ad + bc)
```

These formulas are kernel-safe.

7.8 Series Computation in Parallel

Many mathematical series: - Are independent per term - Can be evaluated in parallel

Examples: - Taylor series - Fourier series (partial)

GPU strategy: > Each thread computes one or more terms.

7.9 Error Growth in Iterative Processes

Iterative maps (fractals, simulations): - Accumulate floating-point error - Diverge quickly

GPU implication: - Precision choice matters - Early termination conditions matter

This becomes critical in fractal generation.

7.10 Preparing for the First Math Project

The next chapter will implement:

Project 1: Mandelbrot & Julia Set Engine

This project will use: - Complex arithmetic - Iterative divergence tests - Massive parallelism

This chapter provides the mathematical tools required.

Chapter Summary

You now understand: - Why parallel math behaves differently - Floating-point limits on GPUs - How to structure math safely - How complex numbers are represented

You are now ready to build a serious mathematical GPU system.

End of Chapter 7

Next file: **Phase 2 – Chapter 8: Project 1 – Mandelbrot & Julia Set Engine (GPU)** # Phase 2 — Chapter 8

Project 1: Mandelbrot & Julia Set Engine (GPU)

This chapter implements the **first complete, non-trivial GPU project** in this book. It integrates mathematics, correctness discipline, CUDA kernels, and CPU–GPU coordination. This is not a demo. It is a reference-quality implementation.

8.1 Problem Definition

We want to compute, for each point in the complex plane, how fast an iterative complex map diverges.

Mandelbrot Set

$z_{n+1} = z_n^2 + c, \quad z_0 = 0$

Julia Set

$z_{n+1} = z_n^2 + k$

Divergence condition:

$|z_n|^2 > 4$

Each pixel is **independent** → ideal for GPUs.

8.2 CPU Reference Implementation (Baseline)

This implementation is intentionally simple and slow. Its only purpose is **correctness**.

```python
import numpy as np

def mandelbrot_cpu(xmin, xmax, ymin, ymax, width, height, max_iter):
    output = np.zeros((height, width), dtype=np.int32)

    for j in range(height):
        y = ymin + (ymax - ymin) * j / height
        for i in range(width):
            x = xmin + (xmax - xmin) * i / width

            zr, zi = 0.0, 0.0
            cr, ci = x, y

            count = 0
            while zr*zr + zi*zi <= 4.0 and count < max_iter:
                zr, zi = zr*zr - zi*zi + cr, 2.0*zr*zi + ci
                count += 1

            output[j, i] = count

    return output
```

This version defines the **ground truth**.

8.3 Mapping the Problem to the GPU

Key decisions:

- One thread → one pixel
- 2D grid → image plane
- Store real and imaginary parts explicitly

Thread responsibility: > Given pixel index → compute divergence iteration count

---

8.4 GPU Kernel: Mandelbrot

```
from numba import cuda
import numpy as np

@cuda.jit
def mandelbrot_kernel(xmin, xmax, ymin, ymax, width, height, max_iter, output):
    x, y = cuda.grid(2)

    if x < width and y < height:
        real = xmin + (xmax - xmin) * x / width
        imag = ymin + (ymax - ymin) * y / height

        zr = 0.0
        zi = 0.0

        count = 0
        while zr*zr + zi*zi <= 4.0 and count < max_iter:
            zr, zi = zr*zr - zi*zi + real, 2.0*zr*zi + imag
            count += 1
        output[y, x] = count
```

This kernel is: - Correct - Deterministic per thread - Independent across threads

---

8.5 Kernel Launch Configuration

```
threads_per_block = (16, 16)
blocks_x = (width + threads_per_block[0] - 1) // threads_per_block[0]
blocks_y = (height + threads_per_block[1] - 1) // threads_per_block[1]
blocks_per_grid = (blocks_x, blocks_y)

output_d = cuda.device_array((height, width), dtype=np.int32)

mandelbrot_kernel[blocks_per_grid, threads_per_block](
    xmin, xmax, ymin, ymax,
    width, height, max_iter,
    output_d
)

cuda.synchronize()
```

This is the **canonical 2D CUDA launch pattern**.

8.6 Validation Against CPU

```
out_cpu = mandelbrot_cpu(xmin, xmax, ymin, ymax, width, height, max_iter)
out_gpu = output_d.copy_to_host()

assert np.array_equal(out_cpu, out_gpu)
```

Validation is mandatory before visualization.

8.7 Julia Set Variant

Only the constant changes.

```
@cuda.jit
def julia_kernel(xmin, xmax, ymin, ymax, width, height, max_iter, cr, ci, output):
    x, y = cuda.grid(2)

    if x < width and y < height:
        zr = xmin + (xmax - xmin) * x / width
        zi = ymin + (ymax - ymin) * y / height

        count = 0
        while zr*zr + zi*zi <= 4.0 and count < max_iter:
            zr, zi = zr*zr - zi*zi + cr, 2.0*zr*zi + ci
            count += 1

        output[y, x] = count
```

This demonstrates **kernel reuse with parameterization**.

8.8 Visualization (CPU-Side)

```
import matplotlib.pyplot as plt

plt.imshow(out_gpu, cmap='inferno')
plt.colorbar()
plt.title('Mandelbrot Set (GPU)')
plt.show()
```

GPU computes. CPU renders. Clean separation.

8.9 Performance Discussion

Observations: - Kernel is compute-heavy - Minimal global memory access - Excellent GPU utilization

This is why fractals scale extremely well on GPUs.

8.10 Extensions and Experiments

Suggested experiments: - Increase resolution - Animate zoom - Compare float32 vs float64 - Add smooth coloring

---

Project Summary

You have built: - A mathematically correct fractal engine - A validated GPU kernel - A full CPU–GPU pipeline

This project establishes a template for all future projects.

End of Project 1

Next file: **Phase 2 – Chapter 9: Complex Series and Parallel Evaluation on GPU** # Phase 2 — Chapter 9

Complex Series and Parallel Evaluation on the GPU

This chapter focuses on **evaluating mathematical series in parallel**, especially series involving complex numbers. Series appear everywhere: signal processing, physics simulations, geometry, and numerical analysis. GPUs change how these series must be structured.

9.1 Why Series Matter in GPU Computing

Many problems reduce to series of the form:

$S = \sum_{k=0}^{N-1} f(k)$

On CPUs, this is computed sequentially. On GPUs, **ordering changes**, which affects numerical behavior.

Key insight: > Series evaluation is both a mathematical and architectural problem.

9.2 Independent-Term Series (GPU-Friendly)

A series is GPU-friendly if: - Each term is independent - Terms can be computed in any order

Example:

$S = \sum_{k=0}^{N-1} a_k$

Each thread computes one or more terms.

9.3 Parallel Strategy for Series Evaluation

Common strategies:

1. One thread per term
2. Partial sums per block
3. Hierarchical reduction

This avoids a single long dependency chain.

9.4 Example: Complex Exponential Series

Consider the truncated series:

$e^{ix} = \sum_{k=0}^{N-1} \frac{(ix)^k}{k!}$

We compute real and imaginary parts separately.

9.5 CPU Reference Implementation

```python
import numpy as np
import math

def complex_exp_series_cpu(x, N):
    real = 0.0
    imag = 0.0

    for k in range(N):
        coeff = (x ** k) / math.factorial(k)
        if k % 4 == 0:
            real += coeff
        elif k % 4 == 1:
            imag += coeff
        elif k % 4 == 2:
            real -= coeff
        else:
            imag -= coeff

    return real, imag
```

This version defines correctness.

9.6 GPU Kernel: Term Computation

Each thread computes **one term**.

```python
from numba import cuda
import math

@cuda.jit
def exp_series_terms(x, terms_real, terms_imag):
    k = cuda.grid(1)
    if k < terms_real.size:
        coeff = (x ** k) / math.factorial(k)
        r = k % 4

        if r == 0:
            terms_real[k] = coeff
            terms_imag[k] = 0.0
        elif r == 1:
            terms_real[k] = 0.0
            terms_imag[k] = coeff
        elif r == 2:
            terms_real[k] = -coeff
            terms_imag[k] = 0.0
        else:
            terms_real[k] = 0.0
            terms_imag[k] = -coeff
```

This kernel is intentionally simple.

9.7 Reduction of Series Terms

After term computation: - Sum real parts - Sum imaginary parts

Reduction strategies: - GPU reduction kernel (later chapters) - CPU reduction (acceptable here)

```
real_sum = terms_real_d.copy_to_host().sum()
imag_sum = terms_imag_d.copy_to_host().sum()
```

Correctness first. Optimization later.

9.8 Numerical Error Considerations

Problems: - Large factorials overflow - Power terms grow rapidly

Mitigations: - Limit N - Scale inputs - Prefer stable series forms

GPUs expose instability quickly.

9.9 When Series Fail on GPUs

Avoid series when: - Terms depend on previous terms - Cancellation dominates - High precision is required

In such cases, reformulate the problem.

9.10 Preparing for Geometry and Signals

Series evaluation underlies: - Fourier expansions - Curve generation - Signal synthesis

This chapter prepares the ground for geometry and signal projects.

Chapter Summary

You now understand: - How to parallelize series computation - How to handle complex terms - Where numerical error arises - How GPUs change summation behavior

---

End of Chapter 9

Next file: **Phase 3 – Chapter 10: Coordinate Geometry and Transformations on the GPU** # Phase 3 — Chapter 10

Coordinate Geometry and Transformations on the GPU

This chapter introduces **coordinate geometry as a GPU workload**. Geometry is fundamentally data-parallel: each point can be computed independently. GPUs are therefore a natural fit when geometry is structured correctly.

10.1 Why Geometry Belongs on the GPU

Geometry problems often involve: - Large numbers of points - Identical transformations - Minimal branching

This maps perfectly to the GPU execution model: > One thread → one point

---

10.2 Coordinate Systems Review

We will work with three coordinate systems:

- Cartesian: (x, y)
- Polar: (r,)
- Parametric: (x(t), y(t))

GPU implication: - Each thread converts or transforms one coordinate

---

10.3 Cartesian to Polar Conversion

Mathematical definitions:

$$r = \sqrt{x^2 + y^2}, \qquad \theta = \text{atan2}(y, x)$$

10.4 CPU Reference Implementation

```python
import numpy as np
import math

def cartesian_to_polar_cpu(x, y):
    r = np.sqrt(x*x + y*y)
    theta = np.arctan2(y, x)
    return r, theta
```

This defines correctness.

10.5 GPU Kernel: Cartesian to Polar

```
from numba import cuda
import math

@cuda.jit
def cartesian_to_polar_kernel(x, y, r, theta):
    i = cuda.grid(1)
    if i < x.size:
        r[i] = math.sqrt(x[i]*x[i] + y[i]*y[i])
        theta[i] = math.atan2(y[i], x[i])
```

Each thread handles one point.

10.6 Parametric Curve Generation

Parametric curves:

$x(t) = f(t), \quad y(t) = g(t)$

Sampling strategy: - Each thread evaluates one value of t

10.7 GPU Kernel: Parametric Curve

```
@cuda.jit
def parametric_curve_kernel(t, x, y):
    i = cuda.grid(1)
    if i < t.size:
        x[i] = math.cos(t[i])
        y[i] = math.sin(t[i])
```

This generates a unit circle.

10.8 Affine Transformations

Affine transformation matrix:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

10.9 GPU Kernel: Affine Transform

```
@cuda.jit
def affine_transform_kernel(x, y, a, b, c, d, tx, ty):
    i = cuda.grid(1)
    if i < x.size:
        x_new = a*x[i] + b*y[i] + tx
        y_new = c*x[i] + d*y[i] + ty
        x[i] = x_new
        y[i] = y_new
```

This kernel forms the basis of geometry pipelines.

10.10 Numerical Considerations

Geometry kernels: - Are usually memory-bound - Accumulate floating-point error over transformations

Mitigations: - Normalize coordinates - Avoid repeated transforms when possible

---

Chapter Summary

You now understand: - How geometry maps to GPU threads - Coordinate system conversion on GPU - Parametric curve generation - Affine transformations in parallel

This chapter prepares the foundation for geometry-heavy projects.

End of Chapter 10

Next file: **Phase 3 – Chapter 11: Project 2 – Parametric & Lissajous Curve Engine** # Phase 3 — Chapter 11

Project 2: Parametric & Lissajous Curve Engine (GPU)

This project builds a **geometry generation engine** using the GPU. Unlike raster images, curves require *precise sampling*, *stable math*, and *clean CPU–GPU separation.* This project is a stepping stone toward simulation and signal processing.

11.1 Problem Definition

We want to generate large numbers of points for parametric curves:

$x(t) = f(t), \quad y(t) = g(t)$

and special cases like **Lissajous curves**:

$$x(t) = A\sin(at + \delta)$$
$$y(t) = B\sin(bt)$$

Each sample point is independent → ideal for GPUs.

11.2 CPU Reference Implementation

This version exists only for correctness.

```python
import numpy as np
import math

def lissajous_cpu(t, A, B, a, b, delta):
    x = np.zeros_like(t)
    y = np.zeros_like(t)

    for i in range(len(t)):
        x[i] = A * math.sin(a * t[i] + delta)
        y[i] = B * math.sin(b * t[i])

    return x, y
```

11.3 Mapping the Problem to the GPU

Design decisions: - One thread → one value of `t` - No inter-thread communication - Pure arithmetic kernel

This keeps performance predictable.

11.4 GPU Kernel: Lissajous Curve

```python
from numba import cuda
import math

@cuda.jit
def lissajous_kernel(t, x, y, A, B, a, b, delta):
    i = cuda.grid(1)
    if i < t.size:
        x[i] = A * math.sin(a * t[i] + delta)
        y[i] = B * math.sin(b * t[i])
```

This kernel is: - Branch-free - Compute-heavy - Numerically stable

---

11.5 Kernel Launch

```
threads = 256
blocks = (t.size + threads - 1) // threads

x_d = cuda.device_array_like(t)
y_d = cuda.device_array_like(t)
t_d = cuda.to_device(t)

lissajous_kernel[blocks, threads](t_d, x_d, y_d, A, B, a, b, delta)
cuda.synchronize()
```

11.6 Validation

```
x_cpu, y_cpu = lissajous_cpu(t, A, B, a, b, delta)
x_gpu = x_d.copy_to_host()
y_gpu = y_d.copy_to_host()

assert np.allclose(x_cpu, x_gpu)
assert np.allclose(y_cpu, y_gpu)
```

Always validate before plotting.

11.7 Visualization (CPU-Side)

```
import matplotlib.pyplot as plt

plt.plot(x_gpu, y_gpu, '.', markersize=1)
plt.axis('equal')
plt.title('Lissajous Curve (GPU)')
plt.show()
```

GPU computes. CPU visualizes.

11.8 Performance Characteristics

Observations: - Minimal memory traffic - Heavy use of transcendental functions - Scales well with resolution

This is a **compute-bound** kernel.

11.9 Extensions and Experiments

Try: - Animate parameter changes - Generate multiple curves in one kernel - Add affine transformations - Explore numerical precision limits

---

Project Summary

You have built: - A GPU-based geometry generator - A reusable parametric kernel - A clean CPU–GPU pipeline

This project prepares you for simulation workloads.

End of Project 2

Next file: **Phase 4 – Chapter 12: Time-Stepped Simulations on the GPU** # Phase 4 — Chapter 12

Time-Stepped Simulations on the GPU

This chapter introduces **time-stepped simulations**, where system state evolves over time. Simulations are more demanding than geometry or series because **numerical stability and data dependencies** matter. GPUs excel here only when the problem is structured correctly.

12.1 What Makes Simulations Different

Unlike pure geometry: - State at time $t + \Delta t$ depends on time t - Errors accumulate over steps - Stability can dominate performance

GPU rule: > Parallelize over entities, not over time.

12.2 Discrete Time Integration

We simulate continuous systems by discretizing time.

Let: - Position: `x(t)` - Velocity: `v(t)` - Acceleration: `a(t)`

Basic update:

$$x_{n+1} = x_n + v_n\Delta t, \qquad v_{n+1} = v_n + a_n\Delta t$$

12.3 Euler Integration (Baseline)

Euler's method is simple but unstable for stiff systems.

CPU reference:

```
import numpy as np

def euler_step_cpu(x, v, a, dt):
    x_new = x + v * dt
    v_new = v + a * dt
    return x_new, v_new
```

This defines correctness.

12.4 Mapping Simulation to the GPU

Design decisions: - One thread → one particle - All particles update in parallel - Synchronization only between time steps (CPU-side)

We launch **one kernel per time step**.

12.5 GPU Kernel: Euler Step

```
from numba import cuda

@cuda.jit
def euler_step_kernel(x, v, a, dt):
    i = cuda.grid(1)
    if i < x.size:
        x[i] = x[i] + v[i] * dt
        v[i] = v[i] + a[i] * dt
```

Each thread updates one particle.

12.6 Time Loop (CPU-Controlled)

```
for step in range(num_steps):
    euler_step_kernel[blocks, threads](x_d, v_d, a_d, dt)
    cuda.synchronize()
```

Important: - GPU handles spatial parallelism - CPU controls time progression

---

12.7 Numerical Stability Concerns

Euler method: - Accumulates error quickly - Can explode for large `dt`

Symptoms: - Positions diverge - Energy grows unrealistically

Stability matters more than speed.

12.8 Verlet Integration (Improved)

Verlet update:

$x_{n+1} = 2x_n - x_{n-1} + a_n \Delta t^2$

This is more stable for physical systems.

12.9 GPU Kernel: Verlet Step

```python
@cuda.jit
def verlet_step_kernel(x, x_prev, a, dt):
    i = cuda.grid(1)
    if i < x.size:
        x_new = 2.0 * x[i] - x_prev[i] + a[i] * dt * dt
        x_prev[i] = x[i]
        x[i] = x_new
```

This kernel uses more memory but better stability.

12.10 When Simulations Scale Well on GPUs

Simulations scale well when: - Many entities - Local interactions - Uniform update rules

They scale poorly when: - Global coupling dominates - Time steps are adaptive per entity

---

Chapter Summary

You now understand: - How time-stepped simulations differ from geometry - How to structure simulation kernels - Why CPU controls time - Euler vs Verlet integration - Stability vs performance trade-offs

This chapter prepares you for full physical simulations.

End of Chapter 12

Next file: **Phase 4 – Chapter 13: Project 3 – N-Body Gravitational Simulation**

Phase 4 — Chapter 13

Project 3: N-Body Gravitational Simulation (GPU)

This project implements a **classical N-body gravitational simulation**, one of the most important benchmark problems in scientific computing. It stresses arithmetic intensity, memory access patterns, numerical stability, and GPU scaling behavior.

13.1 Problem Definition

Each particle attracts every other particle via Newton's law of gravitation:

$$\vec{F}_{ij} = G\frac{m_i m_j}{r_{ij}^2}\hat{r}_{ij}$$

Acceleration of particle i:

$$\vec{a}_i = \sum_{j \neq i} G\frac{m_j(\vec{r}_j - \vec{r}_i)}{|\vec{r}_j - \vec{r}_i|^3}$$

Computational complexity: - **O(N²)** per time step

Each particle's force computation is independent → GPU-suitable.

13.2 CPU Reference Implementation

This version defines correctness, not performance.

```
import numpy as np

def nbody_cpu(pos, vel, mass, G, dt):
    N = pos.shape[0]
    acc = np.zeros_like(pos)

    for i in range(N):
        for j in range(N):
            if i != j:
                r = pos[j] - pos[i]
                dist_sq = np.dot(r, r) + 1e-9
                inv_dist3 = 1.0 / (dist_sq * np.sqrt(dist_sq))
                acc[i] += G * mass[j] * r * inv_dist3

    vel += acc * dt
    pos += vel * dt

    return pos, vel
```

This is slow but unambiguous.

13.3 Mapping the Problem to the GPU

Design choices: - One thread → one particle - Each thread loops over all other particles - All force accumulation happens in registers

Time stepping is controlled by the CPU.

13.4 Naive GPU Kernel (Global Memory)

```
from numba import cuda
import math

@cuda.jit
def nbody_naive_kernel(pos, vel, mass, G, dt):
    i = cuda.grid(1)
    N = pos.shape[0]

    if i < N:
        ax = 0.0
        ay = 0.0

        xi = pos[i, 0]
        yi = pos[i, 1]

        for j in range(N):
            if i != j:
                dx = pos[j, 0] - xi
                dy = pos[j, 1] - yi
                dist_sq = dx*dx + dy*dy + 1e-9
                inv_dist3 = 1.0 / (dist_sq * math.sqrt(dist_sq))
                ax += G * mass[j] * dx * inv_dist3
                ay += G * mass[j] * dy * inv_dist3

        vel[i, 0] += ax * dt
        vel[i, 1] += ay * dt
        pos[i, 0] += vel[i, 0] * dt
        pos[i, 1] += vel[i, 1] * dt
```

This kernel is correct but memory-heavy.

13.5 Kernel Launch and Time Loop

```
threads = 128
blocks = (N + threads - 1) // threads

for step in range(num_steps):
    nbody_naive_kernel[blocks, threads](pos_d, vel_d, mass_d, G, dt)
    cuda.synchronize()
```

13.6 Validation Against CPU

```
pos_cpu, vel_cpu = nbody_cpu(pos.copy(), vel.copy(), mass, G, dt)
pos_gpu = pos_d.copy_to_host()

assert np.allclose(pos_cpu, pos_gpu, rtol=1e-4, atol=1e-5)
```

Floating-point tolerance is required.

13.7 Performance Bottleneck Analysis

Naive kernel characteristics: - O(N^2) global memory loads - Poor cache reuse - Memory bandwidth bound

We fix this with **shared memory tiling**.

13.8 Optimized GPU Kernel (Shared Memory Tiling)

```
@cuda.jit
def nbody_tiled_kernel(pos, vel, mass, G, dt):
    shared_pos = cuda.shared.array((128, 2), dtype=cuda.float64)
    shared_mass = cuda.shared.array(128, dtype=cuda.float64)

    i = cuda.grid(1)
    tid = cuda.threadIdx.x
    N = pos.shape[0]

    if i < N:
        xi = pos[i, 0]
        yi = pos[i, 1]
        ax = 0.0
        ay = 0.0

        for tile in range(0, N, cuda.blockDim.x):
            j = tile + tid
            if j < N:
                shared_pos[tid, 0] = pos[j, 0]
                shared_pos[tid, 1] = pos[j, 1]
                shared_mass[tid] = mass[j]
            cuda.syncthreads()

            for k in range(cuda.blockDim.x):
                dx = shared_pos[k, 0] - xi
                dy = shared_pos[k, 1] - yi
                dist_sq = dx*dx + dy*dy + 1e-9
                inv_dist3 = 1.0 / (dist_sq * math.sqrt(dist_sq))
                ax += G * shared_mass[k] * dx * inv_dist3
                ay += G * shared_mass[k] * dy * inv_dist3
            cuda.syncthreads()

        vel[i, 0] += ax * dt
        vel[i, 1] += ay * dt
        pos[i, 0] += vel[i, 0] * dt
        pos[i, 1] += vel[i, 1] * dt
```

This dramatically reduces global memory traffic.

13.9 Performance Scaling

Observations: - Naive kernel slows quickly as N grows - Tiled kernel scales significantly better - Compute dominates over memory

This is a classic GPU optimization success case.

13.10 Numerical Stability

Important considerations: - Softening term prevents singularities - Time step size affects energy conservation - Verlet integration can improve stability

Stability must be tested, not assumed.

Project Summary

You have implemented: - A full N-body simulation - CPU baseline and validation - Naive and optimized GPU kernels - Shared memory tiling

This is a milestone project.

End of Project 3

Next file: **Phase 5 – Chapter 14: Data Structures and Reductions on the GPU** # Phase 5 — Chapter 14

Data Structures and Reductions on the GPU

This chapter introduces **data structures and collective operations** on GPUs. Unlike CPUs, GPUs do not support flexible pointer-heavy structures efficiently. Instead, performance comes from **structured arrays, reductions, and controlled atomic operations**.

14.1 Why Data Structures Are Hard on GPUs

CPU data structures rely on: - Pointers - Dynamic memory - Branch-heavy logic

GPU constraints: - No dynamic allocation inside kernels - Memory access must be regular - Branch divergence is expensive

Rule: > On GPUs, algorithms must adapt to the hardware.

14.2 The Reduction Problem

Reduction computes a single value from many:

$S = \sum_{i=0}^{N-1} a_i$

Other reductions: - min / max - dot product - norm

Naive GPU approach (wrong): - One thread does all work

Correct approach: - Parallel, hierarchical reduction

14.3 CPU Reference Reduction

```python
import numpy as np

def reduce_sum_cpu(a):
    s = 0.0
    for x in a:
        s += x
    return s
```

This defines correctness.

14.4 Naive GPU Reduction (Atomic)

```python
from numba import cuda

@cuda.jit
def reduce_atomic_kernel(a, out):
    i = cuda.grid(1)
    if i < a.size:
        cuda.atomic.add(out, 0, a[i])
```

This works but scales poorly due to contention.

14.5 Shared Memory Block Reduction

Each block reduces its chunk locally.

```
from numba import float32

@cuda.jit
def reduce_block_kernel(a, partial):
    shared = cuda.shared.array(256, dtype=float32)

    tid = cuda.threadIdx.x
    i = cuda.grid(1)

    if i < a.size:
        shared[tid] = a[i]
    else:
        shared[tid] = 0.0
    cuda.syncthreads()

    stride = cuda.blockDim.x // 2
    while stride > 0:
        if tid < stride:
            shared[tid] += shared[tid + stride]
        cuda.syncthreads()
        stride //= 2

    if tid == 0:
        partial[cuda.blockIdx.x] = shared[0]
```

Final reduction happens on CPU or via another kernel.

14.6 Histogram Computation

Histograms count occurrences per bin.

Challenges: - Multiple threads update same bin

Solution: - Atomic operations

```
@cuda.jit
def histogram_kernel(data, hist):
    i = cuda.grid(1)
    if i < data.size:
        cuda.atomic.add(hist, data[i], 1)
```

Atomic contention limits performance.

14.7 Prefix Sum (Scan)

Prefix sum:

$out[i] = \sum_{k=0}^{i} a_k$

Used in: - Sorting - Graph algorithms - Compaction

GPU scans are complex and multi-stage.

We introduce the concept; full implementation comes later.

14.8 When Atomics Are Acceptable

Atomics are acceptable when: - Contention is low - Bins are many - Accuracy matters more than speed

Avoid atomics in tight inner loops when possible.

14.9 GPU-Friendly Data Structures

Preferred: - Flat arrays - Struct-of-arrays (SoA)

Avoid: - Linked lists - Trees - Pointer chasing

Chapter Summary

You now understand: - Why data structures differ on GPUs - Reduction patterns - Atomic operations - Histogram and scan concepts

This chapter prepares you for algorithm-heavy GPU systems.

End of Chapter 14

Next file: **Phase 5 – Chapter 15: Project 4 – GPU Histogram and Statistical Engine**

Phase 5 — Chapter 15

Project 4: GPU Histogram & Statistical Engine

This project builds a **data-analysis engine** on the GPU. Unlike simulations or geometry, data analytics stresses **reductions, atomics, and numerical correctness**. The goal is to design a pipeline that is correct first, then scalable.

15.1 Problem Definition

Given a large dataset, compute: - Histogram - Mean - Variance - Standard deviation

These operations appear in: - Data science - Signal processing - Scientific analysis

---

15.2 CPU Reference Implementation

This version defines correctness.

```python
import numpy as np

def stats_cpu(data, num_bins):
    hist = np.zeros(num_bins, dtype=np.int64)
    for x in data:
        hist[x] += 1

    mean = data.mean()
    var = ((data - mean) ** 2).mean()
    std = np.sqrt(var)

    return hist, mean, var, std
```

15.3 Data Assumptions

For simplicity: - Data is integer-valued - Values lie in `[0, num_bins)`

These assumptions are common in practice (e.g. pixel intensities).

15.4 GPU Kernel: Histogram

```python
from numba import cuda

@cuda.jit
def histogram_kernel(data, hist):
    i = cuda.grid(1)
    if i < data.size:
        cuda.atomic.add(hist, data[i], 1)
```

This kernel is simple but contention-heavy.

15.5 Kernel Launch and Validation

```python
hist_d = cuda.device_array(num_bins, dtype=np.int64)
hist_d[:] = 0
```

```
data_d = cuda.to_device(data)

threads = 256
blocks = (data.size + threads - 1) // threads

histogram_kernel[blocks, threads](data_d, hist_d)
cuda.synchronize()

hist_gpu = hist_d.copy_to_host()
```

Validation:

```
hist_cpu, _, _, _ = stats_cpu(data, num_bins)
assert np.array_equal(hist_cpu, hist_gpu)
```

15.6 GPU Mean Computation (Reduction)

We reuse reduction patterns.

```
mean_gpu = data_d.copy_to_host().mean()
```

This is acceptable here. GPU-only reductions come later.

15.7 GPU Variance Considerations

Variance formula:

$\sigma^2 = E[x^2] - (E[x])^2$

This is numerically sensitive.

Safer two-pass approach: 1. Compute mean 2. Compute squared deviations

---

15.8 Numerical Accuracy

Pitfalls: - Integer overflow - Catastrophic cancellation

Mitigations: - Use float64 for accumulation - Validate against CPU

15.9 Performance Discussion

Observations: - Histogram is atomic-bound - Mean/variance are memory-bound - GPU helps only for large datasets

This is expected.

15.10 Extensions

Try: - Shared-memory histograms per block - Multi-stage reductions - Streaming large datasets

---

Project Summary

You have built: - A GPU histogram engine - A statistical analysis pipeline - CPU–GPU validation discipline

This project prepares you for algorithmic GPU systems.

End of Project 4

Next file: **Phase 6 – Chapter 16: Hybrid CPU–GPU System Design**

Phase 6 — Chapter 16

Hybrid CPU–GPU System Design

Real-world GPU programs are **not GPU-only**. Successful systems split responsibilities between CPU and GPU based on *control flow*, *data movement*, and *parallel efficiency*. This chapter teaches how to design such hybrid systems deliberately.

16.1 The Core Principle

> CPUs control. GPUs compute.

Trying to move control-heavy logic onto the GPU almost always degrades performance and correctness.

16.2 Strengths of CPU vs GPU

CPU Strengths

- Complex branching
- Dynamic data structures
- I/O and file systems
- Orchestration and scheduling

GPU Strengths

- Massive data parallelism
- Uniform computation
- High arithmetic throughput

Good systems respect this division.

16.3 Data Movement Is the Hidden Cost

Host Device transfers: - Are expensive - Scale poorly - Can dominate runtime

Rule: > Move data as little as possible.

16.4 A Canonical Hybrid Pipeline

Typical pattern:

1. CPU loads and preprocesses data
2. CPU transfers data to GPU
3. GPU performs heavy computation
4. CPU postprocesses and visualizes results

This pattern appears in every serious GPU application.

16.5 Control Loops Belong on the CPU

Example: simulation loop

```
for step in range(num_steps):
    kernel[blocks, threads](state_d)
    cuda.synchronize()
```

Time, stopping conditions, and adaptive logic remain on the CPU.

16.6 Batch vs Streaming Workloads

Batch Workloads

- Large datasets
- One-time computation
- GPU-friendly

Streaming Workloads

- Continuous data arrival
- Smaller chunks
- CPU–GPU coordination required

Design must match workload type.

16.7 Asynchronous Execution (Conceptual)

GPUs can: - Launch kernels asynchronously - Overlap computation and data transfer

Numba supports this partially via streams (covered later).

16.8 Error Handling Strategy

GPU kernels: - Cannot throw Python exceptions

CPU responsibilities: - Validate inputs - Check outputs - Handle failures gracefully

Never trust GPU output blindly.

16.9 Testing Hybrid Systems

Testing strategy: 1. Test CPU components alone 2. Test GPU kernels independently 3. Test integration last

This isolates bugs effectively.

16.10 Design Case Study

Example: Image processing pipeline - CPU loads image - GPU applies filters - CPU saves result

This structure maximizes throughput and clarity.

Chapter Summary

You now understand: - Why hybrid design is mandatory - How to split work between CPU and GPU - Why data movement dominates performance - How to structure real GPU systems

This chapter transitions the book from algorithms to systems.

End of Chapter 16

Next file: **Phase 7 – Chapter 17: Performance Optimization Techniques** # Phase 7 — Chapter 17

Performance Optimization Techniques (Without Myths)

This chapter explains **how to optimize CUDA programs correctly**. GPU optimization is often misunderstood and approached through folklore. Here, we replace myths with first-principles reasoning.

17.1 The Optimization Order (Non-Negotiable)

Always optimize in this order:

1. Correctness
2. Algorithmic complexity
3. Memory access patterns
4. Parallelism and occupancy
5. Instruction-level tuning

Optimizing in any other order wastes time.

17.2 Measure Before You Optimize

Never guess performance.

Required steps: - Establish CPU baseline - Measure GPU kernel time correctly - Identify the dominant cost

If you cannot explain where time goes, stop.

17.3 Occupancy: What It Is and What It Is Not

Occupancy is: - The ratio of active warps to maximum possible warps per SM

Occupancy is **not**: - A guarantee of high performance - A target to maximize blindly

Many kernels reach peak performance below maximum occupancy.

17.4 Choosing Block Size Intelligently

Rules of thumb: - Start with 128 or 256 threads per block - Ensure block size is a multiple of warp size (32) - Prefer simpler choices over micro-tuning

Bad idea: > Randomly trying dozens of block sizes.

17.5 Memory Access Optimization

Key goals: - Coalesced global memory access - Minimize global memory loads - Reuse data via shared memory

Remember: > One global memory load saved is worth many arithmetic operations.

17.6 Shared Memory Tiling Pattern

Canonical pattern:

```
shared = cuda.shared.array(BLOCK_SIZE, dtype=float32)

# Load tile
shared[tid] = global_data[idx]
cuda.syncthreads()

# Compute using shared data
```

Used in: - Convolution - Matrix multiplication - N-body simulation

---

17.7 Avoiding Branch Divergence

Divergence occurs when threads in a warp take different branches.

Mitigation strategies: - Replace conditionals with math where possible - Separate kernels for different cases - Pre-filter data on CPU

17.8 Register Pressure

Too many registers per thread: - Reduce active warps - Lower occupancy

Symptoms: - Slower kernels after "optimization"

Balance register use carefully.

17.9 When Optimization Does Not Matter

Do not optimize when: - Kernel runtime is negligible - Data transfer dominates - Problem size is small

Optimization effort should match impact.

17.10 A Real Optimization Workflow

Example workflow: 1. Implement naive kernel 2. Measure runtime 3. Identify bottleneck 4. Apply one optimization 5. Measure again

Repeat only if improvement is meaningful.

Chapter Summary

You now understand: - The correct order of optimization - What occupancy really means - How to choose block sizes rationally - How memory access dominates performance - When optimization is pointless

This chapter completes your optimization foundation.

End of Chapter 17

Next file: **Phase 7 – Chapter 18: Multi-Dimensional Kernels and Tiling** # Phase 7 — Chapter 18

Multi-Dimensional Kernels and Tiling

Many real GPU problems are **not 1-dimensional**. Images, grids, matrices, and physical fields are naturally 2D or 3D. This chapter teaches how to correctly map multi-dimensional data to CUDA grids and how **tiling** unlocks performance.

18.1 Why Dimensionality Matters

Examples: - Images → 2D arrays - Video → 3D (x, y, time) - Physical fields → 2D / 3D grids

Flattening everything to 1D hides structure and often hurts performance.

18.2 CUDA Grid Dimensions

CUDA supports: - 1D grids - 2D grids - 3D grids

Thread indices:

```
x, y = cuda.grid(2)
x, y, z = cuda.grid(3)
```

Each dimension is independent.

18.3 2D Kernel Indexing

Canonical image kernel pattern:

```
@cuda.jit
def kernel_2d(img, out):
    x, y = cuda.grid(2)
    if x < img.shape[1] and y < img.shape[0]:
        out[y, x] = img[y, x]
```

Rule: > Always check bounds in every dimension.

18.4 Launch Configuration for 2D Kernels

```
threads = (16, 16)
blocks_x = (width + threads[0] - 1) // threads[0]
blocks_y = (height + threads[1] - 1) // threads[1]
blocks = (blocks_x, blocks_y)
```

This is the standard pattern for images.

18.5 Memory Access Patterns in 2D

Row-major storage: - Consecutive elements vary fastest in x-direction

Implication: > Map `x` to contiguous memory when possible.

Incorrect mapping leads to uncoalesced access.

18.6 Tiling: The Core Idea

Tiling means: - Load a small block of data into shared memory - Reuse it for multiple computations

This reduces global memory traffic dramatically.

18.7 Example: 2D Convolution (Naive)

```
@cuda.jit
def conv2d_naive(img, kernel, out):
    x, y = cuda.grid(2)
    if x >= img.shape[1] or y >= img.shape[0]:
        return

    acc = 0.0
    for ky in range(-1, 2):
        for kx in range(-1, 2):
            ix = x + kx
            iy = y + ky
            if 0 <= ix < img.shape[1] and 0 <= iy < img.shape[0]:
                acc += img[iy, ix] * kernel[ky+1, kx+1]

    out[y, x] = acc
```

Correct but memory-heavy.

18.8 Tiled 2D Convolution (Shared Memory)

```
@cuda.jit
def conv2d_tiled(img, kernel, out):
    TILE = 16
    shared = cuda.shared.array((TILE+2, TILE+2), dtype=cuda.float32)

    tx = cuda.threadIdx.x
    ty = cuda.threadIdx.y
    x, y = cuda.grid(2)

    lx = tx + 1
    ly = ty + 1

    if x < img.shape[1] and y < img.shape[0]:
        shared[ly, lx] = img[y, x]
    else:
        shared[ly, lx] = 0.0

    cuda.syncthreads()

    if x < img.shape[1] and y < img.shape[0]:
        acc = 0.0
        for ky in range(3):
            for kx in range(3):
                acc += shared[ly+ky-1, lx+kx-1] * kernel[ky, kx]
        out[y, x] = acc
```

This pattern appears everywhere in image processing.

18.9 3D Kernels (Conceptual)

3D grids extend naturally:

```
x, y, z = cuda.grid(3)
```

Used in: - Volumetric data - 3D simulations - Medical imaging

The same tiling principles apply.

18.10 Common Mistakes

- Forgetting multi-dimensional bounds checks
- Choosing tile sizes that exceed shared memory
- Misaligned memory access

Tiling must respect hardware limits.

Chapter Summary

You now understand: - How to write 2D and 3D CUDA kernels - Proper grid and block configuration - Why tiling is critical for performance - How shared memory enables reuse

This chapter unlocks image processing and PDE solvers.

End of Chapter 18

Next file: **Phase 8 – Chapter 19: Advanced CUDA Features in Numba** # Phase 8 — Chapter 19

Advanced CUDA Features in Numba

This chapter covers **advanced CUDA concepts as exposed through Numba**. These features are powerful, but they are also easy to misuse. The goal here is not to use everything, but to know **what exists, when it helps, and when it does not**.

19.1 Atomic Operations (Deep Dive)

Atomics allow multiple threads to safely update the same memory location.

Supported atomic operations in Numba include: - add - max / min - compare-and-swap (limited)

Example:

```
@cuda.jit
def atomic_add_kernel(a, out):
    i = cuda.grid(1)
    if i < a.size:
        cuda.atomic.add(out, 0, a[i])
```

Atomics are correct but serialize access under contention.

19.2 When Atomics Are the Right Tool

Use atomics when: - Contention is naturally low - Exact correctness is required - Algorithm structure is simple

Avoid atomics when: - Inner loops are tight - Many threads hit the same location

---

19.3 CUDA Streams (Conceptual)

Streams allow: - Overlapping kernel execution - Overlapping memory transfer and compute

In Numba:

```
stream = cuda.stream()
kernel[blocks, threads, stream](args)
```

Streams are useful for **pipelined workloads**.

19.4 Asynchronous Memory Transfers

```
cuda.to_device(host_array, stream=stream)
```

Allows CPU and GPU to work concurrently.

Important: > Asynchronous does not mean faster by default.

19.5 Unified Memory

Unified memory allows the same array to be accessed by CPU and GPU.

```
arr = cuda.managed_array(shape, dtype)
```

Pros: - Simpler code

Cons: - Less predictable performance - Hidden page migrations

Use with caution.

19.6 Constant Memory (Limited in Numba)

Constant memory: - Read-only - Cached - Broadcast-efficient

Numba support is limited, but small constant arrays can still help.

19.7 Limitations of Numba CUDA

Compared to CUDA C++: - Fewer optimization controls - Limited shared memory customization - No cooperative groups

Numba trades control for productivity.

19.8 When to Move Beyond Numba

Consider CUDA C++ when: - Absolute peak performance is required - You need advanced CUDA features - You maintain large, long-lived codebases

Numba is ideal for: - Research - Prototyping - Medium-scale systems

---

19.9 Advanced Feature Anti-Patterns

Avoid: - Using streams everywhere - Overusing unified memory - Mixing too many advanced features

Complexity is not performance.

Chapter Summary

You now understand: - How atomics really behave - How streams enable concurrency - Unified memory trade-offs - The limits of Numba CUDA - When to switch tools

This chapter completes advanced CUDA coverage.

End of Chapter 19

Next file: **Phase 9 – Chapter 20: Production-Grade GPU Systems** # Phase 9 — Chapter 20

Production-Grade GPU Systems

This chapter focuses on **turning GPU code into reliable systems**. Most GPU failures in real applications are not performance issues, but **correctness, reproducibility, and integration problems**. This chapter addresses those directly.

20.1 What "Production-Grade" Actually Means

Production-grade GPU code must be: - Correct - Deterministic (within tolerance) - Testable - Maintainable - Fallback-safe

Raw speed alone is not enough.

20.2 Determinism and Reproducibility

GPU execution is: - Parallel - Order-independent

This means: - Floating-point results may differ run-to-run

Acceptable practice: - Define numerical tolerances - Compare statistically, not bit-exactly

---

20.3 CPU Fallback Strategy

Every GPU system should have a CPU fallback.

Pattern:

```python
if cuda.is_available():
    run_gpu()
else:
    run_cpu()
```

Fallbacks enable: - Debugging - Testing - Deployment on non-GPU systems

---

20.4 Testing GPU Code

Testing levels:

1. Unit tests for CPU logic
2. Kernel-level validation against CPU
3. Integration tests for pipelines

Never test GPU kernels in isolation.

20.5 Error Handling and Failure Modes

GPU kernels: - Cannot raise Python exceptions

Therefore: - Validate inputs on CPU - Check outputs after execution - Guard against NaNs and infinities

---

20.6 Memory Management Discipline

Best practices: - Reuse device buffers - Avoid repeated allocations - Explicitly free when needed

```python
del device_array
```

Memory leaks on GPU are harder to detect.

20.7 Performance Regression Testing

Performance can regress silently.

Practice: - Track baseline timings - Alert on large regressions - Re-measure after driver updates

Performance is part of correctness.

20.8 Packaging GPU Systems

Considerations: - Driver compatibility - CUDA toolkit versions - Hardware capability checks

Document assumptions clearly.

20.9 Case Study: End-to-End GPU Pipeline

Example: - CPU loads data - GPU processes batches - CPU validates and stores results

This pattern appears in ML, simulations, and analytics.

Chapter Summary

You now understand: - What makes GPU code production-ready - How to ensure reproducibility - How to test and validate GPU systems - How to deploy responsibly

This chapter completes the **core technical content** of the book.

End of Chapter 20

Next file: **Appendix A – Exercises, Thought Problems, and Mental Models**

Appendix A — Exercises, Thought Problems, and Mental Models

This appendix turns the book from *readable* into *masterable*. The goal is not to test syntax, but to force correct GPU thinking. Exercises are ordered from conceptual to implementation-heavy.

A.1 How to Use This Appendix

Rules: - Do not optimize unless asked - Always write a CPU baseline first - Validate every GPU result - Measure only after correctness

If you skip these rules, you are practicing habits, not skills.

A.2 Mental Models You Must Internalize

Model 1: One Thread = One Data Element

If a thread needs global coordination, the design is wrong.

Model 2: Memory Is the Bottleneck

Always ask: > How many times does this kernel touch global memory?

Model 3: CPU Controls Time, GPU Controls Space

Time loops live on the CPU. Spatial parallelism lives on the GPU.

Model 4: Correctness Before Speed

A fast wrong answer is useless.

A.3 Conceptual Exercises (No Code)

1. Why does increasing occupancy not always increase performance?
2. Explain why reductions change floating-point results.
3. Why is a linked list a poor GPU data structure?
4. When does unified memory hurt performance?
5. Why are atomics dangerous inside inner loops?

Write answers in your own words.

A.4 Kernel Design Exercises (Short)

Exercise 1: Safe Indexing

Design a kernel that: - Squares an array - Never accesses out-of-bounds memory - Works for any array size

Explain why every line exists.

Exercise 2: Divergence Removal

Given a kernel with multiple branches: - Rewrite it to minimize divergence - Explain the trade-off

---

A.5 Reduction Exercises

Exercise 3: Sum Reduction

1. Write a CPU sum
2. Write a naive atomic GPU sum
3. Write a shared-memory block reduction
4. Compare numerical differences

Explain which version you trust and why.

A.6 Geometry Exercises

Exercise 4: Curve Engine

Generate: - A spiral - A hypotrochoid

Requirements: - GPU generates points - CPU plots - Parameters are configurable

---

A.7 Simulation Exercises

Exercise 5: Stability Test

Simulate a particle in a harmonic potential using: - Euler integration - Verlet integration

Compare: - Energy drift - Stability limits

Explain the results.

A.8 Data Structure Exercises

Exercise 6: Histogram Scaling

Implement: - Naive atomic histogram - Block-local histogram + merge

Measure contention effects.

A.9 System Design Exercises

Exercise 7: Pipeline Design

Design a GPU pipeline for: - Streaming sensor data - Batched processing

Explain: - CPU responsibilities - GPU responsibilities - Data transfer strategy

---

A.10 Capstone-Level Challenges

These are not quick exercises.

1. Implement a GPU image filter library
2. Build a real-time simulation viewer
3. Write a performance regression test suite

Each challenge should take days, not minutes.

A.11 Self-Assessment Checklist

You are ready for serious GPU work if you can: - Predict performance trends - Debug silent failures - Explain memory access patterns - Design hybrid systems

If not, revisit the chapters.

End of Appendix A

Next file: **Appendix B — CUDA Mental Models Cheat Sheet**

Appendix B — CUDA Mental Models Cheat Sheet

This appendix is a **compact, high-signal reference** meant to be revisited often. It condenses the entire book into mental models, rules, and patterns you should recall instantly when writing or reviewing CUDA–Numba code.

B.1 The One-Sentence GPU Model

> The CPU decides *what* and *when*. The GPU decides *how much at once*.

If a design violates this sentence, it is probably wrong.

B.2 Execution Model (Burn This In)

- One kernel launch → one grid
- One grid → many blocks
- One block → many warps
- One warp → 32 threads executing in lockstep

Reality check: > The warp, not the thread, is the execution unit.

B.3 Thread Responsibilities

Correct: - One thread handles one data element - Threads do independent work

Incorrect: - Threads coordinating global state - Threads depending on execution order

---

B.4 Memory Hierarchy (Order Matters)

Fast → Slow: 1. Registers (thread-local) 2. Shared memory (block-local) 3. Global memory (device-wide)

Rule: > Minimize global memory traffic before optimizing math.

B.5 Bounds Checks Are Not Optional

Every kernel must include:

```python
if idx < size:
    ...
```

Skipping bounds checks is not an optimization. It is a bug.

B.6 Synchronization Rules

- `cuda.syncthreads()` synchronizes **within a block only**
- All threads in a block must reach it
- There is no global synchronization inside kernels

If you think you need global sync, redesign.

B.7 Divergence Heuristics

Bad: - Deep branching inside warps

Better: - Replace branches with arithmetic - Separate kernels per case - Pre-filter on CPU

B.8 Reduction Rules

- Reductions change floating-point order
- Numerical differences are expected
- Trust tolerances, not exact matches

Never expect bit-identical CPU and GPU sums.

B.9 Atomics: The Honest Truth

Atomics are: - Correct - Slow under contention

Use atomics when: - Correctness > speed - Contention is naturally low

Avoid atomics in inner loops.

B.10 Occupancy Reality Check

High occupancy: - Hides memory latency

High occupancy does NOT guarantee: - High performance

Memory access patterns matter more.

B.11 Block Size Rules of Thumb

Start with: - 128 or 256 threads per block

Ensure: - Block size is multiple of 32

Avoid: - Blind tuning without measurement

B.12 Multi-Dimensional Mapping

For images: - `x` → columns (contiguous) - `y` → rows

Memory coalescing depends on correct axis mapping.

B.13 Tiling Pattern (Core Optimization)

Canonical structure:

```
load tile → syncthreads → compute → syncthreads
```

Used in: - Convolutions - Matrix ops - N-body simulations

B.14 CPU–GPU Responsibility Split

CPU: - Control flow - I/O - Time stepping - Error handling

GPU: - Bulk computation - Uniform math

Never mix these roles.

B.15 Debugging Checklist

If results look wrong: 1. Reduce problem size 2. Compare against CPU 3. Check bounds 4. Check synchronization 5. Check memory access patterns

Do not guess.

B.16 Performance Checklist

Before optimizing, ask: - Is the kernel memory-bound? - Is data transfer dominating? - Is the problem large enough?

If unsure, measure again.

B.17 When to Stop Optimizing

Stop when: - Kernel time is negligible - Bottleneck is elsewhere - Complexity outweighs gains

Clean code beats clever code.

B.18 Tool Choice Reminder

Use Numba CUDA when: - Productivity matters - Research or prototyping - Medium-scale systems

Use CUDA C++ when: - Absolute peak performance is required - Advanced CUDA features are needed

---

B.19 Final Mental Check

Before writing any kernel, ask:

1. What does one thread do?
2. Where does data live?
3. How often is global memory touched?
4. How is correctness validated?

If you cannot answer these, stop.

End of Appendix B

Next file: **Appendix C — Project Index, Extensions, and Reading Path**

Appendix C — Project Index, Extensions, and Reading Path

This appendix ties the entire book together. It maps **projects to concepts**, proposes **serious extensions**, and gives **clear reading paths** depending on your goals. Use this as your navigation and growth plan.

C.1 Complete Project Index

Project 1 — Mandelbrot & Julia Set Engine (Phase 2, Ch. 8)

Core concepts: - Complex arithmetic without Python `complex` - Iterative divergence tests - 2D grids and bounds checking - CPU validation and visualization

Skills proven: - Mapping math to GPU threads - Writing deterministic kernels - End-to-end CPU–GPU pipelines

---

Project 2 — Parametric & Lissajous Curve Engine (Phase 3, Ch. 11)

Core concepts: - Parametric geometry - One-thread–one-sample mapping - Compute-bound kernels

Skills proven: - Geometry generation at scale - Numerical stability awareness - Visualization separation

---

Project 3 — N-Body Gravitational Simulation (Phase 4, Ch. 13)

Core concepts: - $O(N^2)$ interactions - Shared-memory tiling - Numerical softening - Time stepping

Skills proven: - Performance bottleneck analysis - Shared-memory optimization - Simulation stability reasoning

---

Project 4 — GPU Histogram & Statistical Engine (Phase 5, Ch. 15)

Core concepts: - Atomics - Reductions - Numerical accuracy

Skills proven: - Data-analytic GPU pipelines - Correct use of atomics - Validation under contention

---

C.2 Project Extension Paths (Serious Work)

Each project can be extended into **research-grade or production-grade systems**.

Extensions for Project 1 (Fractals)

- Smooth coloring algorithms
- Orbit traps
- Double-precision zoom engine
- GPU-based coloring kernels
- Animated zoom pipelines using streams

Extensions for Project 2 (Geometry)

- Batch curve generation in one kernel
- Affine transform pipelines
- GPU-based curve rasterization
- Real-time animation loops

Extensions for Project 3 (N-Body)

- Barnes–Hut approximation
- Verlet integration upgrade
- Energy conservation analysis
- 3D extension
- GPU-only time stepping

Extensions for Project 4 (Statistics)

- Block-local histograms
- Multi-stage reductions
- Streaming data ingestion
- GPU percentile computation

C.3 Concept-to-Chapter Map

Use this table when revising.

Concept	Chapters
CUDA execution model	Phase 1
Memory hierarchy	Ch. 4, 17
Reductions	Ch. 9, 14, 15
Geometry	Ch. 10, 11
Simulation	Ch. 12, 13
Optimization	Ch. 17, 18
Production systems	Ch. 16, 20

C.4 Reading Paths by Goal

Path A — Learning CUDA Properly (Recommended)

1. Phase 1 (entire)
2. Phase 2 (Ch. 7–9)
3. Project 1
4. Phase 3
5. Project 2
6. Phase 4
7. Project 3
8. Phase 5
9. Project 4
10. Phase 6–9

Path B — Scientific Computing Focus

1. Phase 1
2. Phase 2
3. Project 1
4. Phase 4
5. Project 3
6. Phase 7
7. Phase 9

Path C — Data & Analytics Focus

1. Phase 1
2. Phase 5
3. Project 4
4. Phase 6
5. Phase 7

Path D — Revision / Interview Prep

1. Appendix B
2. Phase 1
3. Project 3
4. Phase 7
5. Phase 9

C.5 What This Book Enables You To Do

After completing this book, you should be able to: - Design GPU algorithms from scratch - Predict performance bottlenecks - Debug silent GPU failures - Build hybrid CPU–GPU systems - Read CUDA C++ code confidently

This is not entry-level knowledge.

C.6 Where to Go Next

Natural next steps: - CUDA C++ programming guide - OpenCL comparison - GPU-based PDE solvers - Real-time visualization engines

This book gives you the foundation to choose wisely.

End of Appendix C

Next (optional): **Capstone Project — End-to-End Scientific GPU System** # Capstone Project — End-to-End Scientific GPU System

GPU-Accelerated Particle Field Simulator & Analyzer

This capstone integrates **geometry, simulation, reductions, optimization, and production design** into one coherent system. It is intentionally large. Treat it as a real project, not a chapter.

CP.1 System Goal

Build a hybrid CPU–GPU system that: - Simulates a large particle field over time - Applies geometric transforms - Computes statistics and diagnostics - Produces visual outputs - Remains correct, testable, and scalable

This mirrors real scientific and engineering workloads.

CP.2 System Architecture

CPU responsibilities: - Configuration and parameter validation - Time stepping and control flow - I/O, logging, visualization

GPU responsibilities: - Particle state updates - Force computation - Geometry transforms - Reductions and statistics

Rule: > CPU orchestrates. GPU computes.

CP.3 Data Model (Struct-of-Arrays)

```
# Device arrays
pos_x, pos_y      # positions
vel_x, vel_y      # velocities
mass              # masses
acc_x, acc_y      # accelerations
```

Why SoA: - Coalesced access - Independent updates - Lower register pressure

---

CP.4 CPU Reference (Correctness Baseline)

```
import numpy as np

def step_cpu(px, py, vx, vy, m, G, dt):
    N = len(px)
    ax = np.zeros_like(px)
    ay = np.zeros_like(py)
    for i in range(N):
        for j in range(N):
            if i != j:
                dx = px[j] - px[i]
                dy = py[j] - py[i]
                r2 = dx*dx + dy*dy + 1e-9
                invr3 = 1.0 / (r2 * np.sqrt(r2))
                ax[i] += G * m[j] * dx * invr3
                ay[i] += G * m[j] * dy * invr3

    vx += ax * dt
    vy += ay * dt
    px += vx * dt
    py += vy * dt

    return px, py, vx, vy
```

This is slow by design. It defines truth.

CP.5 GPU Kernels

CP.5.1 Force Computation (Tiled)

```
from numba import cuda
import math

@cuda.jit
def force_kernel(px, py, vx, vy, m, G, dt):
    TILE = 128
    sx = cuda.shared.array(TILE, dtype=cuda.float64)
    sy = cuda.shared.array(TILE, dtype=cuda.float64)
    sm = cuda.shared.array(TILE, dtype=cuda.float64)

    i = cuda.grid(1)
    tid = cuda.threadIdx.x
    N = px.size

    if i >= N:
        return

    xi = px[i]
    yi = py[i]
    axi = 0.0
    ayi = 0.0

    for base in range(0, N, TILE):
        j = base + tid
        if j < N:
            sx[tid] = px[j]
            sy[tid] = py[j]
            sm[tid] = m[j]
        cuda.syncthreads()

        for k in range(min(TILE, N - base)):
            dx = sx[k] - xi
            dy = sy[k] - yi
            r2 = dx*dx + dy*dy + 1e-9
            invr3 = 1.0 / (r2 * math.sqrt(r2))
            axi += G * sm[k] * dx * invr3
            ayi += G * sm[k] * dy * invr3
        cuda.syncthreads()

    vx[i] += axi * dt
    vy[i] += ayi * dt
    px[i] += vx[i] * dt
    py[i] += vy[i] * dt
```

CP.5.2 Geometry Transform Kernel

```
@cuda.jit
def rotate_kernel(px, py, angle):
    i = cuda.grid(1)
    if i < px.size:
        c = math.cos(angle)
        s = math.sin(angle)
        x = px[i]
        y = py[i]
        px[i] = c*x - s*y
        py[i] = s*x + c*y
```

CP.6 Statistics and Diagnostics

Kinetic Energy Reduction

```
@cuda.jit
def kinetic_energy(px, py, vx, vy, m, out):
    i = cuda.grid(1)
    if i < m.size:
        out[i] = 0.5 * m[i] * (vx[i]**2 + vy[i]**2)
```

Final reduction can be CPU or GPU.

CP.7 Simulation Loop (CPU-Controlled)

```
for step in range(steps):
    force_kernel[blocks, threads](px_d, py_d, vx_d, vy_d, m_d, G, dt)
    rotate_kernel[blocks, threads](px_d, py_d, angle)
    cuda.synchronize()
```

CPU controls time and termination.

CP.8 Validation Strategy

1. Run CPU for small N
2. Run GPU for same N
3. Compare with tolerances
4. Increase N gradually

Never skip validation.

CP.9 Performance Analysis

Expected behavior: - O(N^2) compute dominates - Shared memory improves scaling - GPU outperforms CPU beyond modest N

Measure, don't assume.

CP.10 Production Readiness Checklist

- CPU fallback available
- Bounds checks in all kernels
- Numerical tolerances defined
- Device memory reused
- Clear separation of concerns

If any box is unchecked, it is not done.

CP.11 Extension Ideas

- Barnes–Hut approximation
- 3D extension
- GPU-only reductions
- Streamed visualization
- Checkpointing and restart

Capstone Summary

You have designed: - A full hybrid GPU system - Multiple interacting kernels - A validation and analysis pipeline - A production-ready architecture

This capstone represents **professional-level GPU engineering**.

End of Capstone Project