Notebook 0: The Accelerated Python Ecosystem

1. Ecosystem Overview

Imagine your Python code strapping into a high-speed rocket—these libraries are the fuel that lights the engines:

- CuPy
  Think of CuPy as NumPy's twin who went to space camp. Its API mirrors NumPy almost exactly, but every array operation happens on your GPU, turning sluggish loops into warp-speed math.

- RAPIDS
  Remember pandas and scikit-learn? RAPIDS clones their interfaces and offloads the work to your GPU. DataFrames and ML pipelines zip through terabytes of data in the blink of an eye.

- Warp
  Want to build physics simulations or graphics kernels that compile on the fly? Warp is your playground. Write Python that's JIT-compiled into blazing-fast code, fully differentiable and ready to plug into frameworks like PyTorch.

Self-Assessment: Ecosystem Overview
(Answer sheet is available in the end of the document)

1. Which library gives you a drop-in NumPy replacement for GPU arrays?
   A) Warp
   B) CuPy
   C) Numba
   D) RAPIDS

2. RAPIDS is best described as:
   A) A graphics-only JIT compiler
   B) A suite of GPU-accelerated data science tools
   C) A cloud service for Python
   D) A CPU multithreading library

2. Beyond the Big Three

But wait—there's more in this Python cosmos! Once you've mastered the main engines, try these boosters:

- CUDA Python
  Directly script against NVIDIA's CUDA platform without leaving Python.

- nvmath-python
  Tap into CUDA-X math libraries—FFT, BLAS, and more—from Python.

- Numba
  A JIT compiler that can target CPU or GPU, accelerating annotated Python and NumPy code seamlessly.

- cuPyNumeric
  For massive scale: Legate's distributed array library that fans out work over many GPUs or nodes.

Self-Assessment: Other Libraries
(Answer sheet is available in the end of the document)

3. Which library lets you call low-level CUDA APIs directly from Python?
   A) cuPyNumeric
   B) nvmath-python
   C) CUDA Python
   D) RAPIDS

4. Numba differs from CuPy mainly because it:
   A) Only works on CPUs
   B) Requires no Python code
   C) Can JIT-compile arbitrary Python functions for CPU and GPU
   D) Is limited to array operations

3. Introduction to the Workshop Lab

Time to get your hands dirty! In our notebook:

- Markdown cells hold explanations, links, and tips.

- Code cells run Python or shell commands.

- Prefix a cell with ! to run system tools.

python

```
print("Hello, GPU World!!!")
```

bash

```
!nvidia-smi   # See your GPU's stats, like 'top' for NVIDIA
!lscpu        # Check your CPU type
```

When you're ready for a clean slate, restart the kernel:

python

```
import IPython
app = IPython.Application.instance()
app.kernel.do_shutdown(True)
```

Self-Assessment: Lab Environment
(Answer sheet is available in the end of the document)

5. What does a leading ! in a notebook cell do?
    A) Marks it as markdown
    B) Runs a shell command
    C) Comments out the line
    D) JIT-compiles the code

6. To completely reset all variables and imports in your notebook, you should:
    A) Close the browser tab
    B) Run app.kernel.do_shutdown(True)
    C) Type exit() in a code cell
    D) Delete all cells

Notebook 1: Introduction to CuPy

1. NumPy vs CuPy: A Tale of Two Arrays

Think of NumPy as your trusty bicycle—dependable for everyday errands. CuPy is the same bike, but fitted with rocket boosters so you can zip around at GPU speeds.

- NumPy stores data in numpy.ndarray on your CPU's memory.
- CuPy gives you a cupy.ndarray that lives on the GPU, so all your array math can run in parallel on hundreds or thousands of cores.

Switching is often as simple as:

python

```python
# NumPy version
import numpy as np
A = np.random.randn(512, 512)
Q, R = np.linalg.qr(A)

# CuPy version
import cupy as cp
A = cp.random.randn(512, 512)
Q, R = cp.linalg.qr(A)
cp.cuda.Device().synchronize()  # Make sure we wait for the GPU
    to finish
```

Self-Assessment: Section 1
(Answer sheet is available in the end of the document)

7. Which array type lives on the GPU?
   A) numpy.ndarray
   B) cupy.ndarray
   C) list
   D) tuple

8. Why do we call cp.cuda.Device().synchronize() when benchmarking?
   A) To wait until all GPU work is done
   B) To free GPU memory
   C) To speed up the kernel launch
   D) To convert the array back to NumPy

2. CPU vs GPU Showdown

Time to pit your CPU bicycle against the GPU rocket! For a 4096×4096 matrix multiply:

Python

```
# NumPy (CPU)
from time import perf_counter
A = np.random.rand(4096,4096).astype(np.float32)
B = np.random.rand(4096,4096).astype(np.float32)
start = perf_counter()
C = np.matmul(A, B)
print("NumPy time:", perf_counter() - start)

# CuPy (GPU)
A = cp.random.rand(4096,4096).astype(cp.float32)
B = cp.random.rand(4096,4096).astype(cp.float32)
start = perf_counter()
C = cp.matmul(A, B)
cp.cuda.Device(0).synchronize()
print("CuPy time:", perf_counter() - start)
```

You'll see minutes of CPU time collapse to seconds on the GPU!

Self-Assessment: Section 2
(Answer sheet is available in the end of the document)

9. During timing, why must we run synchronize() on the GPU?
    A) To ensure the GPU finishes its work before we stop the clock
    B) To delete the arrays on the GPU
    C) To merge data across CPU and GPU

D) To precompile kernels

10. For large matrix multiplications, which wins?
    A) CuPy on the GPU
    B) NumPy on the CPU
    C) They tie every time
    D) It depends on the Python version

3. Speed-Boost Tricks: Overhead and Fusion

GPUs are fast—but launching thousands of tiny kernels or moving data back and forth can kill your speedup.

- Kernel Overhead
  The first time you call a CuPy function it must JIT-compile code (and create a CUDA context), so it's a bit slow up front. After that, calls are cached.

- Data Movement
  Sending arrays across the PCIe bus (host ↔ GPU) is orders of magnitude slower than GPU math. Minimize transfers by keeping data on the device when possible.

- Kernel Fusion
  Combine multiple operations into one kernel to cut launch overhead. Decorate small functions with @cp.fuse:

Python

```python
@cp.fuse
def fused_diff(x, y):
    return (x - y)**2
```

Self-Assessment: Section 3
(Answer sheet is available in the end of the document)

11. The first CuPy function call in a session is slow because of:
    A) JIT compilation & CUDA context setup
    B) Data transfer from CPU
    C) Kernel fusion overhead
    D) Python interpreter startup

12. Kernel fusion helps performance by:
    A) Preloading data to the CPU
    B) Reducing the number of separate kernel launches
    C) Increasing Python's garbage collection
    D) Disabling asynchronous execution

4. Advanced Features: Streams & Events

By default, all GPU ops run on the "default stream" in order. But you can create extra streams to overlap work:

Python

```python
stream1 = cp.cuda.Stream()
stream2 = cp.cuda.Stream()
event1  = cp.cuda.Event()
event2  = cp.cuda.Event()

with stream1:
    out1 = cp.linalg.inv(A_gpu)
    event1.record(stream1)

with stream2:
    out2 = cp.linalg.inv(A_gpu)
    event2.record(stream2)

# Wait for both to finish
event1.synchronize()
event2.synchronize()
```

- Streams let you queue multiple tasks concurrently.
- Events let you know exactly when a task in a stream completes—handy for fine-grained timing and coordination.

Self-Assessment: Section 4
(Answer sheet is available in the end of the document)

13. A CuPy stream represents:
    A) A sequence of GPU operations that execute in order
    B) A new type of array
    C) A CPU thread pool
    D) A memory buffer

14. CuPy events are used to:
    A) Mark when a GPU operation in a stream has finished
    B) Launch kernels automatically
    C) Transfer data from GPU to CPU
    D) Precompile functions

ANSWER KEY

Question	Answer
1	**B**
2	**B**
3	**C**
4	**C**
5	**B**
6	**B**
7	**B**
8	**A**
9	**A**
10	**A**
11	**A**
12	**B**
13	**A**
14	**A**