HPC: Numba Python

Sajina Pathak

May 2025

1 Objective

Learn Numba for high-performance computing using Python and CUDA.

2 Introduction

Numba is a just-in-time compiler for Python that makes numerical computing faster. The Just-In-Time (JIT) compiler is a component of the runtime environment that improves performance by compiling bytecode into native machine code at runtime.

Numba, a Python compiler from Anaconda that can compile Python code for execution on CUDA-capable GPUs, provides Python developers with an easy entry into GPU-accelerated computing. It allows rapid development with Python and the speed of compiled code targeting both CPUs and NVIDIA GPUs.

Functions Learned from the Numba Tutorial

- `cuda.to_device(data)`: Copies data from CPU (host) to GPU (device).
- `copy_to_host()`: Copies data from GPU (device) back to CPU (host).
- `device_array_like()`: Allocates a new, empty device array on the GPU with the same shape, data type, and strides as an existing array-like object.

What is a Fractal?

A fractal is a complex pattern built from simple mathematical rules. It can be infinitely detailed — like a pattern inside a pattern. For example, zooming into a snowflake and seeing the same shape over and over again is a fractal.

What is a Julia Set?

A Julia set is a type of fractal that comes from iterating the formula:

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

Each pixel is independent — allowing millions of them to be calculated in parallel on a GPU. This makes Julia sets ideal for GPU demonstrations.

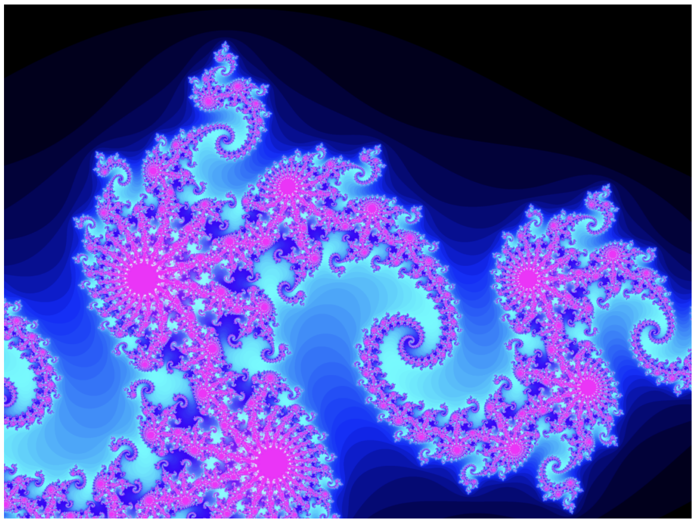


Figure 1: Julia Fractal

3 Problem Solutions

Problem 1: Basic Image Blur with Numba-CUDA

We rewrote the image blur kernel from Lab 4 in Python using Numba-CUDA.

- Removed explicit width/height by using `input_img.shape` inside the kernel.
- Used `cuda.grid(2)` and thread indexing to apply a 9×9 average blur.
- Applied the kernel to a real image ("Lenna") and compared the result with `scikit-image`'s Gaussian blur.

Conclusion: The 9×9 average blur implemented with a basic Numba-CUDA kernel correctly produced a smoothed image. The kernel operated without shared memory, directly accessing global memory, and was visually validated against a Gaussian blur. While performance can be improved using shared memory, the result confirms functional correctness of the CUDA indexing and averaging logic.

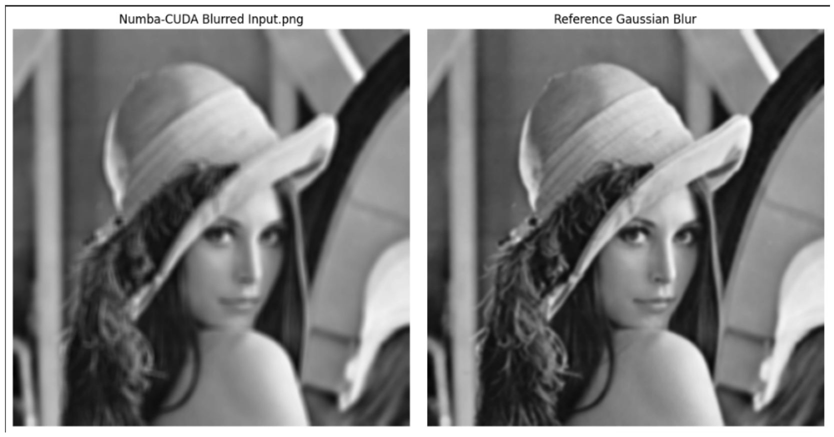


Figure 2: Visual comparison between the 9×9 average blur implemented with Numba-CUDA (left) and the Gaussian blur from `scikit-image` (right). Both filters produce a smoothing effect, validating the correctness of the GPU kernel.

Problem 2: Shared Memory Optimized Image Blur

We improved the performance of the 9×9 image blur kernel by using shared memory tiling. This technique loads tiles (with border halos) into fast shared memory to reduce global memory access overhead. So we "tile" a portion of the image, including the 9×9 neighborhood, into shared memory once. All threads reuse that tile to reduce reads. The size of shared memory should be TILE WIDTH + 2*RADIUS in both dimensions.

Explanation: Why the Initial Shared Memory Blur Output Was Blocky

The first implementation of the shared memory blur kernel produced a blocky and overly blurred image. This was due to multiple issues in how the image tile was loaded and processed within shared memory. Below is a breakdown of the problems and their corresponding fixes.

- **Incorrect Shared Memory Indexing:** Threads were computing pixel positions incorrectly when copying from global memory into shared memory. As a result, pixels from the wrong parts of the image were loaded into the tile, leading to distorted smoothing when the blur was applied.
- **Halo Misalignment:** The "halo" — a 4-pixel padding required for the 9×9 blur — was misaligned or incomplete. Without correct halo loading, each thread's blur computation missed neighboring pixels at the tile boundaries, introducing tiling artifacts.
- **Strided Loading Bug:** The original code used stride-based loops to load shared memory:

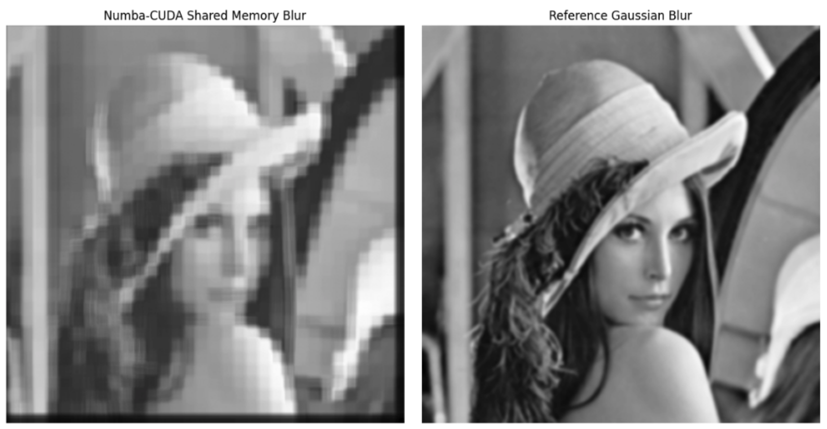


Figure 3: Output of the initial Numba-CUDA shared memory blur implementation (left) compared to the reference Gaussian blur (right). The left image exhibits severe blockiness and excessive blur due to incorrect shared memory tile loading and halo misalignment.

```
for j in range(ty, SHARED_SIZE, TILE_WIDTH):
    for i in range(tx, SHARED_SIZE, TILE_WIDTH):
        global_x = x - RADIUS + i
        global_y = y - RADIUS + j
        ...
```

This pattern caused some threads to skip pixels or go out of bounds during the tile load, making the shared memory data incomplete.

- **Visible Artifacts:** These bugs led to an output image with visible blocks and coarse averaging, resembling a low-resolution or pixelated effect.

Fix: Rewritten Shared Memory Indexing and Tile Load

To correct the shared memory tile loading and indexing, the following code was used:

```
# Coordinates in shared memory
shared_x = tx + RADIUS
shared_y = ty + RADIUS

# Load full tile including halo
for dy in range(-RADIUS, RADIUS + 1):
    for dx in range(-RADIUS, RADIUS + 1):
        global_x = x + dx
        global_y = y + dy
```

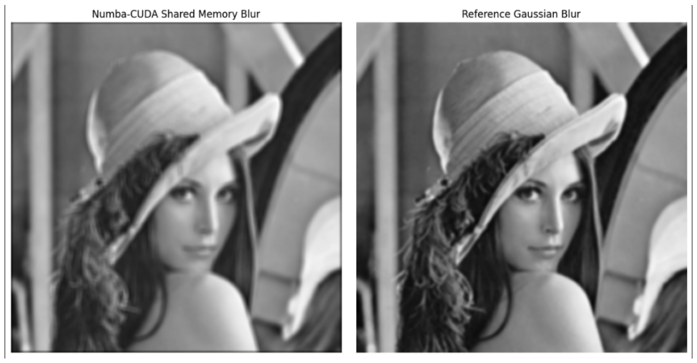


Figure 4: Numba-CUDA Shared Memory Blur (left) now closely matches the Reference Gaussian Blur (right) — with no block artifacts or pixelation

```
sx = shared_x + dx
sy = shared_y + dy

if 0 <= global_x < width and 0 <= global_y < height:
    shared_tile[sy, sx] = input_img[global_y, global_x]
else:
    shared_tile[sy, sx] = 0.0
```

This version:

- Properly maps shared memory positions relative to the thread's position.
- Fully loads a 24×24 tile (16×16 plus 4-pixel halo).
- Avoids skipping or repeating pixels using direct coordinate translation.

With this fix, the shared memory kernel produced an output visually identical to the reference Gaussian blur, confirming correctness.

Problem 3: Shared Memory Histogram Kernel

- Implemented a 256-bin histogram kernel using `Numba-CUDA`.
- Used per-block shared memory to reduce contention on global memory.
- Validated the correctness by comparing to NumPy's CPU histogram using `np.bincount`.

Conclusion: The shared-memory histogram kernel correctly accumulates bin counts from 10 million random `uint8` values. All results match NumPy's histogram exactly, validating the correctness of the GPU implementation. While the current grid configuration shows low GPU occupancy, the kernel logic is sound, and performance tuning can further optimize parallelism.

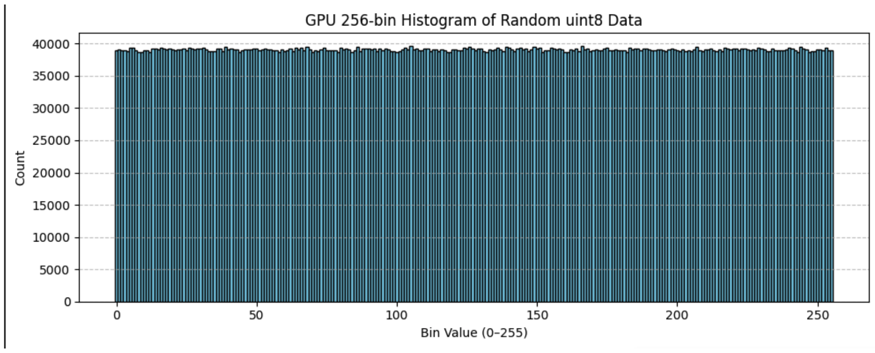
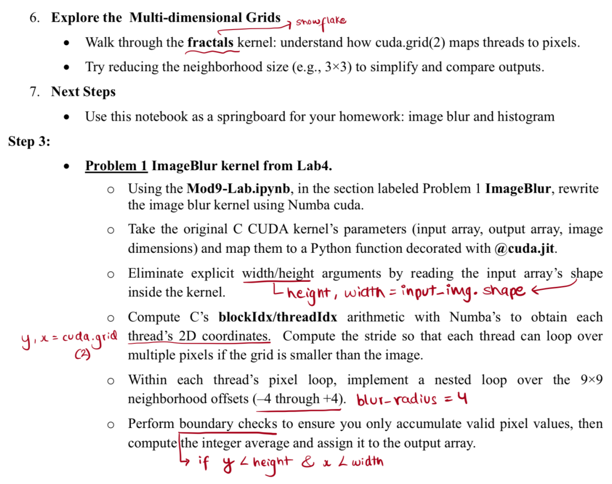


Figure 5: GPU-generated 256-bin histogram of 10 million random uint8 values using Numba-CUDA. The nearly uniform distribution confirms random input and correct binning.

4 References

- CUDA by Numba Examples (Towards Data Science)
- Numba NVIDIA CUDA Tutorial GitHub Repository
- Numba Official Documentation
- CUDA Toolkit - NVIDIA
- scikit-image Library

Extra information for future self

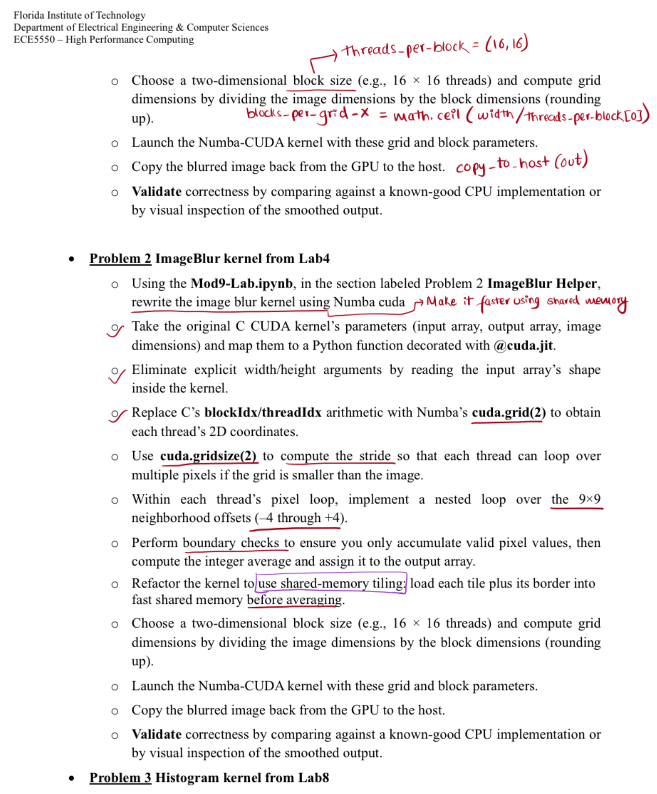


6. **Explore the Multi-dimensional Grids** → snowflake
   - Walk through the **fractals** kernel: understand how cuda.grid(2) maps threads to pixels.
   - Try reducing the neighborhood size (e.g., 3×3) to simplify and compare outputs.
7. **Next Steps**
   - Use this notebook as a springboard for your homework: image blur and histogram

Step 3:

- **Problem 1 ImageBlur kernel from Lab4.**
  - Using the **Mod9-Lab.ipynb**, in the section labeled Problem 1 **ImageBlur**, rewrite the image blur kernel using Numba cuda.
  - Take the original C CUDA kernel's parameters (input array, output array, image dimensions) and map them to a Python function decorated with **@cuda.jit**.
  - Eliminate explicit width/height arguments by reading the input array's shape inside the kernel. height, width = input_img. shape
  - Compute C's **blockIdx/threadIdx** arithmetic with Numba's to obtain each thread's 2D coordinates. Compute the stride so that each thread can loop over multiple pixels if the grid is smaller than the image. y, x = cuda.grid (2)
  - Within each thread's pixel loop, implement a nested loop over the 9×9 neighborhood offsets (–4 through +4). blur_radius = 4
  - Perform boundary checks to ensure you only accumulate valid pixel values, then compute the integer average and assign it to the output array. ↳ if $y < height$ & $x < width$

Figure 6: Understanding Problem 1



Florida Institute of Technology
Department of Electrical Engineering & Computer Sciences
ECE5550 – High Performance Computing

  - Choose a two-dimensional block size (e.g., 16 × 16 threads) and compute grid dimensions by dividing the image dimensions by the block dimensions (rounding up). → threads_per_block = (16,16) blocks_per_grid_x = math.ceil (width / threads_per_block[0])
  - Launch the Numba-CUDA kernel with these grid and block parameters.
  - Copy the blurred image back from the GPU to the host. copy_to_host (out)
  - **Validate** correctness by comparing against a known-good CPU implementation or by visual inspection of the smoothed output.

- **Problem 2 ImageBlur kernel from Lab4**
  - Using the **Mod9-Lab.ipynb**, in the section labeled Problem 2 **ImageBlur Helper**, rewrite the image blur kernel using Numba cuda → Make it faster using shared memory
  - ✓ Take the original C CUDA kernel's parameters (input array, output array, image dimensions) and map them to a Python function decorated with **@cuda.jit**.
  - ✓ Eliminate explicit width/height arguments by reading the input array's shape inside the kernel.
  - ✓ Replace C's **blockIdx/threadIdx** arithmetic with Numba's **cuda.grid(2)** to obtain each thread's 2D coordinates.
  - Use **cuda.gridsize(2)** to compute the stride so that each thread can loop over multiple pixels if the grid is smaller than the image.
  - Within each thread's pixel loop, implement a nested loop over the 9×9 neighborhood offsets (–4 through +4).
  - Perform boundary checks to ensure you only accumulate valid pixel values, then compute the integer average and assign it to the output array.
  - Refactor the kernel to use shared-memory tiling: load each tile plus its border into fast shared memory before averaging.
  - Choose a two-dimensional block size (e.g., 16 × 16 threads) and compute grid dimensions by dividing the image dimensions by the block dimensions (rounding up).
  - Launch the Numba-CUDA kernel with these grid and block parameters.
  - Copy the blurred image back from the GPU to the host.
  - **Validate** correctness by comparing against a known-good CPU implementation or by visual inspection of the smoothed output.
- **Problem 3 Histogram kernel from Lab8**

Figure 7: Understanding Problem 2