Bhuvi Ghosh
60009210191

HPC Experiment-9

Aim:
To implement and compare a histogramming algorithm using both CPU and GPU (CUDA) approaches, and demonstrate performance improvement by leveraging GPU parallelism.

Theory:

A **histogram** is a graphical representation that organizes a group of data points into user-specified ranges. In computing histograms, we count how many elements fall into each bin.

The CPU implementation uses a loop to process elements sequentially, whereas the **GPU (CUDA)** implementation processes multiple elements in parallel using threads. This is especially useful for large datasets.

Key CUDA concepts involved:

- **Grid-stride loop**: Ensures all data elements are covered by allowing threads to iterate over data in strides.
- **Atomic operations**: Ensure safe concurrent updates to shared memory (e.g., incrementing the same histogram bin by multiple threads).

Formula to compute bin index:

python
CopyEdit
```
bin_number = int((element - xmin) / bin_width)
```

Where bin_width = (xmax - xmin) / nbins

Code:

CPU Implementation:

```
import numpy as np

def cpu_histogram(x, xmin, xmax, histogram_out):
    '''Increment bin counts in histogram_out, given histogram range [xmin, xmax).'''
    nbins = histogram_out.shape[0]
    bin_width = (xmax - xmin) / nbins

    for element in x:
```

```
        bin_number = np.int32((element - xmin)/bin_width)
        if 0 <= bin_number < nbins:
            histogram_out[bin_number] += 1
```

CUDA GPU Implementation:

```
from numba import cuda

@cuda.jit
def cuda_histogram(x, xmin, xmax, histogram_out):
    '''Increment bin counts in histogram_out, given histogram range [xmin, xmax).'''
    nbins = histogram_out.shape[0]
    bin_width = (xmax - xmin) / nbins

    start = cuda.grid(1)
    stride = cuda.gridsize(1)

    for i in range(start, x.shape[0], stride):
        element = x[i]
        bin_number = int((element - xmin) / bin_width)
        if 0 <= bin_number < nbins:
            cuda.atomic.add(histogram_out, bin_number, 1)
```

Running & Testing the Code:

```
x = np.random.normal(size=10000, loc=0, scale=1).astype(np.float32)
xmin = np.float32(-4.0)
xmax = np.float32(4.0)

histogram_out_cpu = np.zeros(shape=10, dtype=np.int32)
cpu_histogram(x, xmin, xmax, histogram_out_cpu)

d_x = cuda.to_device(x)
d_histogram_out = cuda.to_device(np.zeros(shape=10, dtype=np.int32))

blocks = 128
threads_per_block = 64

cuda_histogram[blocks, threads_per_block](d_x, xmin, xmax, d_histogram_out)

histogram_out_gpu = d_histogram_out.copy_to_host()
```

np.testing.assert_array_almost_equal(histogram_out_gpu, histogram_out_cpu, decimal=2)
print("Histogram (GPU):", histogram_out_gpu)
print("Histogram (CPU):", histogram_out_cpu)

In [16]:
```
from numba import cuda

@cuda.jit
def cuda_histogram(x, xmin, xmax, histogram_out):
    '''Increment bin counts in histogram_out, given histogram range [xmin, xmax).'''
    nbins = histogram_out.shape[0]
    bin_width = (xmax - xmin) / nbins

    start = cuda.grid(1)
    stride = cuda.gridsize(1)

    for i in range(start, x.shape[0], stride):
        element = x[i]
        bin_number = int((element - xmin) / bin_width)
        if 0 <= bin_number < nbins:
            cuda.atomic.add(histogram_out, bin_number, 1)
```

In [17]:
```
d_x = cuda.to_device(x)
d_histogram_out = cuda.to_device(np.zeros(shape=10, dtype=np.int32))

blocks = 128
threads_per_block = 64

cuda_histogram[blocks, threads_per_block](d_x, xmin, xmax, d_histogram_out)
```

In [18]:
```
# This assertion will fail until you correctly implement `cuda_histogram`
np.testing.assert_array_almost_equal(d_histogram_out.copy_to_host(), histogram_out, decimal=2)
```

Conclusion:

This lab demonstrates the effectiveness of GPU acceleration using CUDA for computing histograms. With the help of **grid-stride loops** and **atomic operations**, we can process large data arrays significantly faster than traditional CPU-based methods. This approach is scalable and ideal for high-performance computing tasks in data analysis, image processing, and scientific simulations.