Assignment # 4

PDC

Submitted by:

Abdullah Latif

SP23-BCS-005

Section (A)

Submitted on: **December 12, 2025**

Department of Computer Science

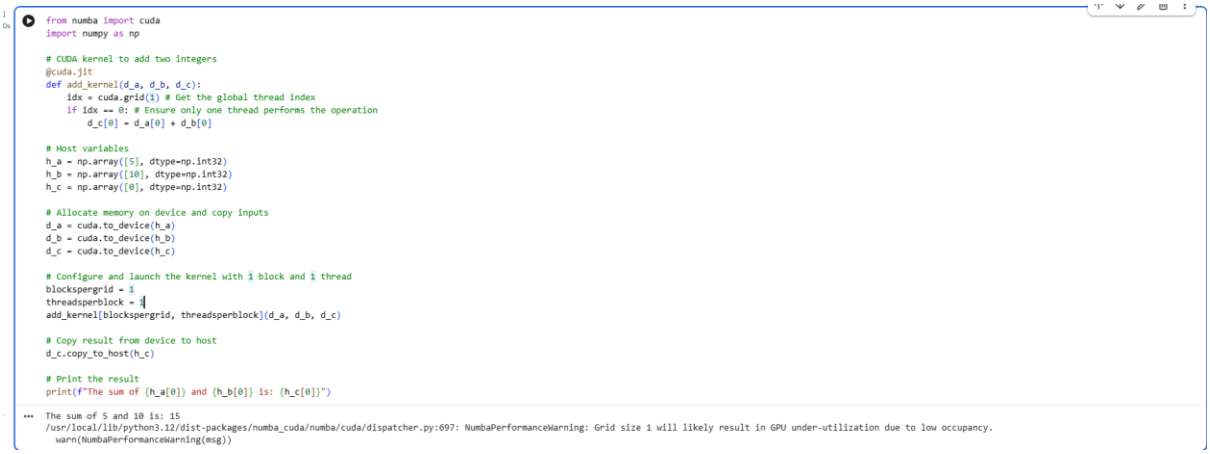
COMSATS University Islamabad

Lahore Campus

Question#01: CLO: <4> Implement parallel programming algorithms using appropriate parallel and distributed computing paradigms; Bloom Taxonomy Level: <Applying>

A. Write a CUDA program to add two integers using a device kernel. Allocate memory on the GPU for the two input integers and one result variable. Perform the addition on the GPU and copy the result back to the host for printing. Use only 1 block and 1 thread.

```
from numba import cuda
import numpy as np

# CUDA kernel to add two integers
@cuda.jit
def add_kernel(d_a, d_b, d_c):
    idx = cuda.grid(1) # Get the global thread index
    if idx == 0: # Ensure only one thread performs the operation
        d_c[0] = d_a[0] + d_b[0]

# Host variables
h_a = np.array([5], dtype=np.int32)
h_b = np.array([10], dtype=np.int32)
h_c = np.array([0], dtype=np.int32)

# Allocate memory on device and copy inputs
d_a = cuda.to_device(h_a)
d_b = cuda.to_device(h_b)
d_c = cuda.to_device(h_c)

# Configure and launch the kernel with 1 block and 1 thread
blockspergrid = 1
threadsperblock = 1
add_kernel[blockspergrid, threadsperblock](d_a, d_b, d_c)

# Copy result from device to host
d_c.copy_to_host(h_c)

# Print the result
print(f"The sum of {h_a[0]} and {h_b[0]} is: {h_c[0]}")
```



```
from numba import cuda
import numpy as np

# CUDA kernel to add two integers
@cuda.jit
def add_kernel(d_a, d_b, d_c):
    idx = cuda.grid(1) # Get the global thread index
    if idx == 0: # Ensure only one thread performs the operation
        d_c[0] = d_a[0] + d_b[0]

# Host variables
h_a = np.array([5], dtype=np.int32)
h_b = np.array([10], dtype=np.int32)
h_c = np.array([0], dtype=np.int32)

# Allocate memory on device and copy inputs
d_a = cuda.to_device(h_a)
d_b = cuda.to_device(h_b)
d_c = cuda.to_device(h_c)

# Configure and launch the kernel with 1 block and 1 thread
blockspergrid = 1
threadsperblock = 1
add_kernel[blockspergrid, threadsperblock](d_a, d_b, d_c)

# Copy result from device to host
d_c.copy_to_host(h_c)

# Print the result
print(f"The sum of {h_a[0]} and {h_b[0]} is: {h_c[0]}")
```

```
The sum of 5 and 10 is: 15
/usr/local/lib/python3.12/dist-packages/numba_cuda/numba/cuda/dispatcher.py:697: NumbaPerformanceWarning: Grid size 1 will likely result in GPU under-utilization due to low occupancy.
  warn(NumbaPerformanceWarning(msg))
```

B. Write a CUDA program to perform element-wise addition of two 2D integer matrices of size 16×16.

a. Use a 1D grid with 1 block and 256 threads.

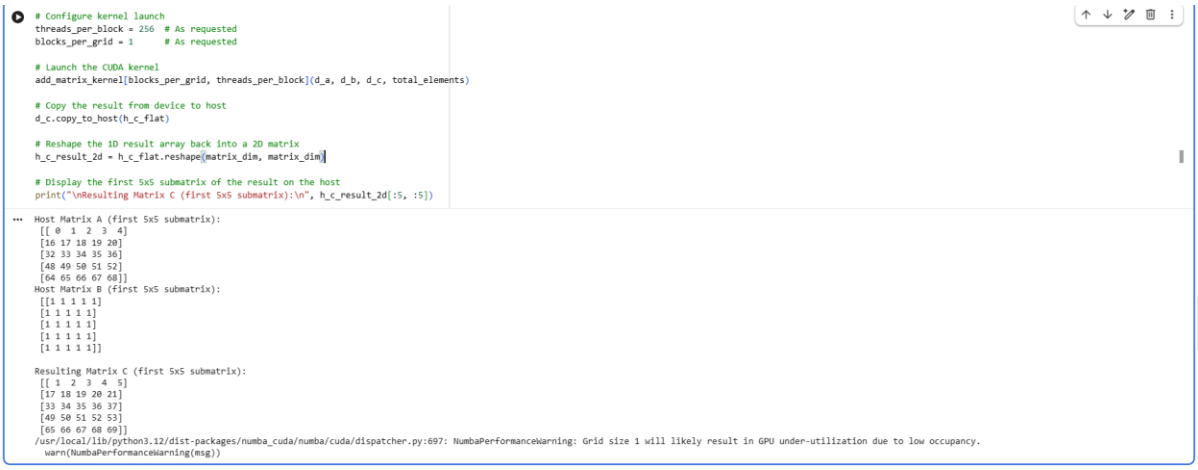
b. Flatten the 2D matrices into 1D arrays for CUDA processing.

c. Each thread should compute the sum of one matrix element.

d. Display the first 5×5 submatrix of the result on the host.

```
from numba import cuda
import numpy as np

# CUDA kernel to perform element-wise addition of two 1D arrays
@cuda.jit
def add_matrix_kernel(d_a, d_b, d_c, total_elements):
    idx = cuda.grid(1) # Get the global thread index
    if idx < total_elements:
        d_c[idx] = d_a[idx] + d_b[idx]

# Define matrix dimensions
matrix_dim = 16
total_elements = matrix_dim * matrix_dim # 16 * 16 = 256

# Host matrices initialization
# For demonstration, initialize A with values from 0 to 255
# and B with all ones.
h_a_2d = np.arange(total_elements, dtype=np.int32).reshape(matrix_dim,
matrix_dim)
h_b_2d = np.full((matrix_dim, matrix_dim), 1, dtype=np.int32)
h_c_2d = np.zeros((matrix_dim, matrix_dim), dtype=np.int32)

print("Host Matrix A (first 5x5 submatrix):\n", h_a_2d[:5, :5])
print("Host Matrix B (first 5x5 submatrix):\n", h_b_2d[:5, :5])
```

```
# Flatten the 2D matrices into 1D arrays for CUDA processing
h_a_flat = h_a_2d.flatten()
h_b_flat = h_b_2d.flatten()
h_c_flat = h_c_2d.flatten()

# Allocate memory on device and copy inputs
d_a = cuda.to_device(h_a_flat)
d_b = cuda.to_device(h_b_flat)
d_c = cuda.to_device(h_c_flat)

# Configure kernel launch
threads_per_block = 256  # As requested
blocks_per_grid = 1      # As requested

# Launch the CUDA kernel
add_matrix_kernel[blocks_per_grid, threads_per_block](d_a, d_b, d_c,
total_elements)

# Copy the result from device to host
d_c.copy_to_host(h_c_flat)

# Reshape the 1D result array back into a 2D matrix
h_c_result_2d = h_c_flat.reshape(matrix_dim, matrix_dim)

# Display the first 5x5 submatrix of the result on the host
print("\nResulting Matrix C (first 5x5 submatrix):\n",
h_c_result_2d[:5, :5])
```



C. Write a CUDA program to add two large integer arrays (e.g., size = 10,000). Use multiple blocks and threads to divide the work, and ensure your kernel properly handles the case when the array size is not a multiple of the number of threads per

block. Include bounds checking using if (index < n) inside the kernel. Print the first 10 elements of the result on the host.

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

# CUDA kernel to perform element-wise addition of two 1D arrays
@cuda.jit
def add_large_arrays_kernel(d_a, d_b, d_c, n):
    # Calculate global index for this thread
    idx = cuda.grid(1)

    # Perform bounds checking
    if idx < n:
        d_c[idx] = d_a[idx] + d_b[idx]

# Define array size
array_size = 10000

# Host arrays initialization
# For demonstration, initialize A with values from 0 to array_size-1
# and B with all ones.
h_a = np.arange(array_size, dtype=np.int32)
h_b = np.full(array_size, 1, dtype=np.int32)
h_c = np.zeros(array_size, dtype=np.int32)

print("Host Array A (first 10 elements):", h_a[:10])
print("Host Array B (first 10 elements):", h_b[:10])

# Allocate memory on device and copy inputs
d_a = cuda.to_device(h_a)
d_b = cuda.to_device(h_b)
d_c = cuda.to_device(h_c)

# Configure kernel launch
threads_per_block = 256 # A common choice, power of 2
# Calculate blocks_per_grid to cover all elements, rounding up
blocks_per_grid = math.ceil(array_size / threads_per_block)

print(f"\nLaunching kernel with {blocks_per_grid} blocks and
{threads_per_block} threads per block.")

# Launch the CUDA kernel
add_large_arrays_kernel[blocks_per_grid, threads_per_block](d_a, d_b,
d_c, array_size)
```

```
# Copy the result from device to host
d_c.copy_to_host(h_c)

# Display the first 10 elements of the result on the host
print("\nResulting Array C (first 10 elements):", h_c[:10])
```