|  |
| --- |
| PARALLEL DISTRIBUTED COMPUTING |
| ASSIGNMENT 3 |
| OMAR ISMAIL SP23-BCS-110 |

**QUESTION 1:**

* **INITIALIZATION:**

1. n = 1024
2. a\_np = np.arange(n, dtype=cp.int32)    #allocating a in cpu(host) memory.
3. b\_np = np.arange(n, dtype=cp.int32)
4. a = cp.asarray(a\_np)                   #allocation in gpu(device) memory.
5. b = cp.asarray(b\_np)
6. c = cp.empty\_like(a)
7. d = cp.empty\_like(a)
8. a[i] = i

b[i] = 2\*i

* **KERNELS:**

kernal\_1 = cp.RawKernel(r'''

extern "C" \_\_global\_\_

void add1(int\* a, int\* b, int\* c, int n){

   int i = blockDim.x \* blockIdx.x + threadIdx.x;

    if (i < n) {

      c[i] = a[i] + b[i];

    }

}''', 'add1')

kernal\_2 = cp.RawKernel(r'''

extern "C" \_\_global\_\_

void add2(int\* c, int\* d, int n){

   int i = blockDim.x \* blockIdx.x + threadIdx.x;

    if (i < n) {

      d[i] = c[i] \* c[i];

    }

}''', 'add2')

c\_cpu = c.get()

print("Element wise addition : ", c\_cpu[:20])

d\_cpu = d.get()                        #to transfer back to cpu from gpu we use get()

print("Element wise square : ", d\_cpu[:20])

* **STREAMS:**

stream\_1 = cp.cuda.Stream()

stream\_2 = cp.cuda.Stream()

with stream\_1:

  kernal\_1((block\_per\_grid, ), (thread\_per\_block, ), (a, b, c, n))

with stream\_2:

  kernal\_2((block\_per\_grid, ), (thread\_per\_block, ), (c, d, n))

stream\_1.synchronize()

stream\_2.synchronize()

c\_cpu = c.get()

print("Element wise addition : ", c\_cpu[:20])

d\_cpu = d.get()                        #to transfer back to cpu from gpu we use get()

print("Element wise square : ", d\_cpu[:20])

**Discussion:**

To prevent the both kernels from accessing the same output array, use a cp.Cuda.event that makes sure that when kernal 1 is complete, it tells that it is complete and there is a event.synchronize that tells the 2nd kernal if 1st kernal has done executing.

* **THREADING:**

**Thread = n , block = 1**

import cupy as cp

import numpy as np

kernel = cp.RawKernel(r'''

extern "C" \_\_global\_\_

void write\_thread\_ids(int\* out\_block, int\* out\_thread, int n) {

    int i = blockDim.x \* blockIdx.x + threadIdx.x;

    if (i < n) {

        out\_block[i] = blockIdx.x;

        out\_thread[i] = threadIdx.x;

    }

}''', 'write\_thread\_ids')

n = 1024

out\_thread = cp.empty(n, dtype=cp.int32)

out\_block = cp.empty(n, dtype=cp.int32)

kernel((1,), (n,), (out\_block, out\_thread, n))

# Copy back to host

out\_cpu\_block = out\_block.get()

print("Block IDs:", out\_cpu\_block)

out\_cpu\_thread = out\_thread.get()

print("Thread IDs:", out\_cpu\_thread)

**OUTPUT:**

Block IDs: [0 0 0 ... 0 0 0]

Thread IDs: [ 0 1 2 ... 1021 1022 1023]

1 block having 1024 threads.

**Thread = n/32, block = 32**

kernel((int(n/32),), (32,), (out\_block, out\_thread, n))

**Output:**

Block IDs: [ 0 0 0 ... 31 31 31]

Thread IDs: [ 0 1 2 ... 29 30 31]

32 Blocks having 32 threads.