

***LAB ASSIGNMENT 3***

|  |  |
| --- | --- |
| **Name :** | Noor Fatima |
| **Registration No:** | SP23-BCS-109 |
| **Course:** | Parallel And Distributing Computing |
| **Teacher:** | Akhzar Nazir |
| **Date:** | 2-10-25 |

1. **Memory Allocation:**

!pip install -q cupy-cuda12x

import numpy as np

import cupy as cp

from cupy.cuda import runtime

print("CUDA device count:", runtime.getDeviceCount())

N = 1024

dtype = np.int32

nbytes = N \* np.dtype(dtype).itemsize

# Host (CPU) arrays

h\_A = np.arange(N, dtype=dtype) # A[i] = i

h\_B = (np.arange(N, dtype=dtype) \* 2) # B[i] = 2\*i

print("host A[:8] =", h\_A[:8])

print("host B[:8] =", h\_B[:8])

# Device allocation (cudaMalloc equivalent)

d\_A\_ptr = runtime.malloc(nbytes)

d\_B\_ptr = runtime.malloc(nbytes)

print("Allocated device pointers:", d\_A\_ptr, d\_B\_ptr)

# Define memcpy kind constants

cudaMemcpyHostToDevice = 1

cudaMemcpyDeviceToHost = 2

# Copy host -> device

runtime.memcpy(d\_A\_ptr, h\_A.ctypes.data, nbytes, cudaMemcpyHostToDevice)

runtime.memcpy(d\_B\_ptr, h\_B.ctypes.data, nbytes, cudaMemcpyHostToDevice)

# Verify: copy back to host

h\_A\_back = np.empty\_like(h\_A)

h\_B\_back = np.empty\_like(h\_B)

runtime.memcpy(h\_A\_back.ctypes.data, d\_A\_ptr, nbytes, cudaMemcpyDeviceToHost)

runtime.memcpy(h\_B\_back.ctypes.data, d\_B\_ptr, nbytes, cudaMemcpyDeviceToHost)

print("after device copy-back A[:8] =", h\_A\_back[:8])

print("after device copy-back B[:8] =", h\_B\_back[:8])

runtime.free(d\_A\_ptr)

runtime.free(d\_B\_ptr)

print("Freed device memory.")

**2. Kernels:**

import numpy as np

import cupy as cp

from cupy.cuda import runtime

N = 1024

dtype = np.int32

# Host arrays

h\_A = np.arange(N, dtype=dtype)

h\_B = np.arange(N, dtype=dtype) \* 2

# Device arrays

d\_A = cp.asarray(h\_A) # allocate & copy to device

d\_B = cp.asarray(h\_B)

d\_C = cp.zeros(N, dtype=dtype)

d\_D = cp.zeros(N, dtype=dtype)

# CUDA kernel source code

kernel\_code = r"""

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

void kernel1(const int \*A, const int \*B, int \*C, int N) {

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

if (i < N) {

C[i] = A[i] + B[i];

}

}

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

void kernel2(const int \*C, int \*D, int N) {

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

if (i < N) {

D[i] = C[i] \* C[i];

}

}

"""

# Compile kernels

module = cp.RawModule(code=kernel\_code)

kernel1 = module.get\_function("kernel1")

kernel2 = module.get\_function("kernel2")

# Thread hierarchy

threads\_per\_block = 32

blocks = (N + threads\_per\_block - 1) // threads\_per\_block

# Launch kernel1 then kernel2 serially (default stream)

kernel1((blocks,), (threads\_per\_block,), (d\_A, d\_B, d\_C, N))

kernel2((blocks,), (threads\_per\_block,), (d\_C, d\_D, N))

h\_C = d\_C.get()

h\_D = d\_D.get()

print("First 10 results of C (A+B):", h\_C[:10])

print("First 10 results of D (C\*C):", h\_D[:10])

**3. Streams:**

import numpy as np

import cupy as cp

N = 1024

dtype = np.int32

# Host arrays

h\_A = np.arange(N, dtype=dtype)

h\_B = np.arange(N, dtype=dtype) \* 2

# Device arrays

d\_A = cp.asarray(h\_A)

d\_B = cp.asarray(h\_B)

d\_C = cp.zeros(N, dtype=dtype)

d\_D = cp.zeros(N, dtype=dtype)

# Separate kernels as RawKernel

kernel1\_src = r'''

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

void kernel1(const int \*A, const int \*B, int \*C, int N) {

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

if (i < N) {

C[i] = A[i] + B[i];

}

}

'''

kernel2\_src = r'''

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

void kernel2(const int \*C, int \*D, int N) {

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

if (i < N) {

D[i] = C[i] \* C[i];

}

}

'''

kernel1 = cp.RawKernel(kernel1\_src, "kernel1")

kernel2 = cp.RawKernel(kernel2\_src, "kernel2")

threads\_per\_block = 32

blocks = (N + threads\_per\_block - 1) // threads\_per\_block

# Create streams

stream1 = cp.cuda.Stream()

stream2 = cp.cuda.Stream()

# ---- Demonstrate race condition ----

# kernel1 writes C in stream1

kernel1((blocks,), (threads\_per\_block,),

(d\_A, d\_B, d\_C, N), stream=stream1)

# kernel2 uses C in stream2 (may overlap => race condition)

kernel2((blocks,), (threads\_per\_block,),

(d\_C, d\_D, N), stream=stream2)

stream1.synchronize()

stream2.synchronize()

h\_C = d\_C.get()

h\_D = d\_D.get()

print("Results with overlap (may be inconsistent):")

print("C[:10] =", h\_C[:10])

print("D[:10] =", h\_D[:10])

# ---- Correct fix: use event to enforce order ----

event = cp.cuda.Event()

kernel1((blocks,), (threads\_per\_block,), (d\_A, d\_B, d\_C, N), stream=stream1)

event.record(stream1)

# make kernel2 wait until kernel1 finishes

stream2.wait\_event(event)

kernel2((blocks,), (threads\_per\_block,), (d\_C, d\_D, N), stream=stream2)

stream1.synchronize()

stream2.synchronize()

h\_D\_fixed = d\_D.get()

print("\nResults after fixing with event (always correct):")

print("D[:10] =", h\_D\_fixed[:10])

• **Race condition**: In the first launch, kernel1 and kernel2 run independently. If kernel2 reads C before it’s fully written, results are corrupted.

• Fix: Use an event → event.record(stream1) + stream2.wait\_event(event) ensures kernel2 waits for kernel1.

**4.Synchronization:**

import numpy as np

import cupy as cp

N = 1024

dtype = np.int32

# Host arrays

h\_A = np.arange(N, dtype=dtype)

h\_B = np.arange(N, dtype=dtype) \* 2

# Device arrays

d\_A = cp.asarray(h\_A)

d\_B = cp.asarray(h\_B)

d\_C = cp.zeros(N, dtype=dtype)

d\_D = cp.zeros(N, dtype=dtype)

# Kernels

kernel1\_src = r'''

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

void kernel1(const int \*A, const int \*B, int \*C, int N) {

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

if (i < N) {

C[i] = A[i] + B[i];

}

}

'''

kernel2\_src = r'''

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

void kernel2(const int \*C, int \*D, int N) {

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

if (i < N) {

D[i] = C[i] \* C[i];

}

}

'''

kernel1 = cp.RawKernel(kernel1\_src, "kernel1")

kernel2 = cp.RawKernel(kernel2\_src, "kernel2")

threads\_per\_block = 32

blocks = (N + threads\_per\_block - 1) // threads\_per\_block

# -------- Case 1: WITH Synchronization --------

kernel1((blocks,), (threads\_per\_block,), (d\_A, d\_B, d\_C, N))

kernel2((blocks,), (threads\_per\_block,), (d\_C, d\_D, N))

# Synchronize before copying

cp.cuda.Device().synchronize()

h\_D\_sync = d\_D.get()

print("Case 1 (with synchronize):")

print("D[:10] =", h\_D\_sync[:10])

# -------- Case 2: WITHOUT Synchronization --------

kernel1((blocks,), (threads\_per\_block,), (d\_A, d\_B, d\_C, N))

kernel2((blocks,), (threads\_per\_block,), (d\_C, d\_D, N))

# ❌ No synchronize here

h\_D\_nosync = d\_D.get() # may fetch before GPU finishes

print("\nCase 2 (without synchronize):")

print("D[:10] =", h\_D\_nosync[:10])

• Case 1 (with synchronize) → always correct, because CPU waits for GPU.

• Case 2 (without synchronize) → may show incomplete/wrong results, especially on larger data or heavier kernels (on fast GPUs sometimes you “get lucky” and still see correct numbers).

**5. Thread Hierarchy:**

import numpy as np

import cupy as cp

N = 1024

dtype = np.int32

h\_A = np.arange(N, dtype=dtype)

h\_B = np.arange(N, dtype=dtype) \* 2

d\_A = cp.asarray(h\_A)

d\_B = cp.asarray(h\_B)

d\_C = cp.zeros(N, dtype=dtype)

# Kernel that stores thread/block index information into arrays

kernel\_src = r'''

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

void kernel1(const int \*A, const int \*B, int \*C, int \*tidArr, int \*bidArr, int N) {

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

if (i < N) {

C[i] = A[i] + B[i];

tidArr[i] = threadIdx.x; // record local thread index

bidArr[i] = blockIdx.x; // record block index

}

}

'''

kernel1 = cp.RawKernel(kernel\_src, "kernel1")

# Arrays to store indexing info

d\_tid = cp.zeros(N, dtype=np.int32)

d\_bid = cp.zeros(N, dtype=np.int32)

# -------- Launch 1: <<<1, N>>> (1 block, N threads) --------

kernel1((1,), (N,), (d\_A, d\_B, d\_C, d\_tid, d\_bid, N))

cp.cuda.Device().synchronize()

C1 = d\_C.get()

tid1 = d\_tid.get()

bid1 = d\_bid.get()

print("Launch <<<1, N>>> (first 8 threads):")

for i in range(8):

print(f"i={i}, A={h\_A[i]}, B={h\_B[i]}, C={C1[i]}, threadIdx={tid1[i]}, blockIdx={bid1[i]}")

# Reset output arrays

d\_C.fill(0)

d\_tid.fill(0)

d\_bid.fill(0)

# -------- Launch 2: <<<N/32, 32>>> (32 blocks, 32 threads each) --------

blocks = N // 32

threads = 32

kernel1((blocks,), (threads,), (d\_A, d\_B, d\_C, d\_tid, d\_bid, N))

cp.cuda.Device().synchronize()

C2 = d\_C.get()

tid2 = d\_tid.get()

bid2 = d\_bid.get()

print("\nLaunch <<<N/32, 32>>> (first 8 threads):")

for i in range(8):

print(f"i={i}, A={h\_A[i]}, B={h\_B[i]}, C={C2[i]}, threadIdx={tid2[i]}, blockIdx={bid2[i]}")

Comparison

Case 1: <<<1, N>>>

Only 1 block (blockIdx.x = 0).

All threads are inside that block (threadIdx.x = 0..1023).

Global index = threadIdx.x.

Case 2: <<<N/32, 32>>>

32 blocks, each with 32 threads.

For example:

Block 0 has threadIdx.x = 0..31 → global indices 0..31.

Block 1 has threadIdx.x = 0..31 → global indices 32..63.

…

Global index = threadIdx.x + blockIdx.x \* blockDim.x.

Both produce the same C = A + B, but indexing math changes.

6. Bonus Challenge:

import numpy as np

import cupy as cp

N = 1024

dtype = np.int32

# Host arrays

h\_A = np.arange(N, dtype=dtype)

h\_B = np.arange(N, dtype=dtype) \* 2

# Device arrays

d\_A = cp.asarray(h\_A)

d\_B = cp.asarray(h\_B)

d\_C = cp.zeros(N, dtype=dtype)

d\_D = cp.zeros(N, dtype=dtype)

d\_sum = cp.zeros(1, dtype=dtype) # single integer for global sum

# Kernels

kernel1\_src = r'''

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

void kernel1(const int \*A, const int \*B, int \*C, int N) {

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

if (i < N) {

C[i] = A[i] + B[i];

}

}

'''

kernel2\_src = r'''

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

void kernel2(const int \*C, int \*D, int N) {

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

if (i < N) {

D[i] = C[i] \* C[i];

}

}

'''

reduction\_src = r'''

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

void reduceSum(const int \*D, int \*result, int N) {

extern \_\_shared\_\_ int sdata[];

int tid = threadIdx.x;

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

// Load data into shared memory

sdata[tid] = (i < N) ? D[i] : 0;

\_\_syncthreads();

// Perform reduction in shared memory

for (int s = blockDim.x / 2; s > 0; s >>= 1) {

if (tid < s) {

sdata[tid] += sdata[tid + s];

}

\_\_syncthreads();

}

// Add block's result to global sum

if (tid == 0) {

atomicAdd(result, sdata[0]);

}

}

'''

kernel1 = cp.RawKernel(kernel1\_src, "kernel1")

kernel2 = cp.RawKernel(kernel2\_src, "kernel2")

reduceSum = cp.RawKernel(reduction\_src, "reduceSum")

# Thread hierarchy

threads\_per\_block = 32

blocks = (N + threads\_per\_block - 1) // threads\_per\_block

# Step 1: Compute C = A + B

kernel1((blocks,), (threads\_per\_block,), (d\_A, d\_B, d\_C, N))

# Step 2: Compute D = C \* C

kernel2((blocks,), (threads\_per\_block,), (d\_C, d\_D, N))

# Step 3: Reduction (sum of all D)

reduceSum((blocks,), (threads\_per\_block,), (d\_D, d\_sum, N),

shared\_mem=threads\_per\_block \* np.dtype(dtype).itemsize)

cp.cuda.Device().synchronize()

h\_sum = d\_sum.get()[0]

print("Sum of all elements in D:", h\_sum)

# Verify on CPU

D\_cpu = (h\_A + h\_B) \*\* 2

print("CPU check (sum of D):", D\_cpu.sum())