National University of Computer and Emerging Sciences
Islamabad Campus

High Performance Computing with GPUs (CS4110)

Course Instructor(s):
Dr. Imran Ashraf
Section(s): A, B, C

Final Exam

Total Time (Min):	**105**
Total Marks:	**54**
Total Questions:	**4**

Date: Dec 23, 2025

Roll No **Course Section** **Student Signature**

Do not write below this line.

Q1 **[Marks = 12]**

[Marks = 3]

A. Kernel-level speedup are given

- Kernel A: 10×
- Kernel B: 5×
- Kernel C: 1× (no acceleration)

[Marks = 3]

B. Overall application speedup (Amdahl's Law)
Fractional times and speedups:

- A: (0.5 / 10 = 0.05)
- B: (0.3 / 5 = 0.06)
- C: (0.2 / 1 = 0.20)

Total normalized time after acceleration: 0.05 + 0.06 + 0.20 = 0.31

Overall speedup:
S = 1 / 0.31 = 3.23x

[Marks = 3]

C. New execution time after GPU acceleration
Original total time = 100 s
Tnew = 100 x 0.31 = 31s

OR: A = 5s, B = 6s, C = 20s => Tnew = 31 s)

[Marks = 3]

D. Bottleneck: Kernel C
After acceleration, Kernel C dominates the runtime, contributing 20 s out of 31 s (~65%). Even though Kernels A and B were accelerated, the serial kernel limits overall speedup, consistent with Amdahl's Law. Further performance gains require accelerating or parallelizing Kernel C.

Q2 **[Marks = 12]**

[Marks = 3]

A. Arithmetic Intensity (AI)

Operations per element: 1 multiply + 1 add = 2 FLOPs per element

Memory accesses per element (single-precision = 4 bytes): 16 bytes per element

Read A[i], B[i], D[i] : 3 x 4 B = 12 B,

Write C[i] = 4 B

AI = FLOPs / Bytes transferred =2/16 = 0.125 FLOPs/byte

[Marks = 3]

B. Memory-bound performance limit

Perf = Bandwidth x AI = 500 GB/s x 0.125 = 62.5 GFLOPS

[Marks = 3]

C. Compute-bound performance limit

Perf = 10 TFLOPS = 10000 GFLOPS

[Marks = 3]

D. Memory-bound or compute-bound?

Memory-bound limit: 62.5 GFLOPS

Compute-bound limit: 10,000 GFLOPS

Since the achievable performance is far below the compute peak and capped by memory bandwidth. Therefore, **the kernel is memory-bound.**

Q3 **[Marks = 15]**

```
// Optimized Reduction Kernel
__global__ void sumReduce(int *input, int *results, int n) {
    // Dynamic shared memory allocation
    extern __shared__ int sdata[];

    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;

    // Load data into shared memory (Coalesced access)
    sdata[tid] = (i < n) ? input[i] : 0;
    __syncthreads();

    // Tree-based reduction in shared memory
    // Using strided approach to avoid warp divergence, bank
conflicts
    for (unsigned int s = blockDim.x / 2; s > 0; s >>= 1) {
        if (tid < s) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

    // First thread of each block writes the block's partial sum to
global memory
    if (tid == 0) {
        results[blockIdx.x] = sdata[0];
    }
}

int main() {
    const int N = 1 << 20;
    const int TPB = 256;
    const int BPG = (N + TPB - 1) / TPB;

    int *data, *partialSums;
    int deviceId;
    cudaGetDevice(&deviceId);
```

```
    // 1. Unified Memory Allocation
    cudaMallocManaged(&data, N * sizeof(int));
    cudaMallocManaged(&partialSums, BPG * sizeof(int));

    // Initialize data
    for (int i = 0; i < N; i++) data[i] = 1;

    // Memory Prefetching (Performance Optimization)
    cudaMemPrefetchAsync(data, N * sizeof(int), deviceId);

    // Kernel Launch with Shared Memory size specification
    sumReduce<<<BPG, TPB, TPB*sizeof(int)>>>(data, partialSums, N);

    // Synchronize
    cudaDeviceSynchronize();

    // Prefetch Results back to CPU
    cudaMemPrefetchAsync(partialSums, BPG * sizeof(int),
cudaCpuDeviceId);

    // Final Sum on Host
    long long finalSum = 0;
    for (int i = 0; i < BPG; i++) {
        finalSum += partialSums[i];
    }

    std::cout << "GPU Accelerated Sum: " << finalSum << std::endl;

    cudaFree(data);
    cudaFree(partialSums);

    return 0;
}
```

Q4 **[Marks = 15]**

A)

Each warp = 32 threads × 4 bytes/thread = 128 bytes total data.

B)

Global memory burst size = 64 bytes.

To read 128 bytes → 128 / 64 = 2 memory transactions.

Therefore, 2 bursts per warp when perfectly aligned.

C)

If array A starts 16 bytes off from a 64-byte boundary, some threads in the warp will access data spanning three 64-byte segments instead of two. Hardware must issue 3 transactions to serve 128 bytes of data due to overlap across boundaries. Misalignment increases the number of transactions from 2 to 3, reducing coalescing efficiency.

D)

- Ensure arrays are aligned to 128-byte boundaries.
- Access memory with consecutive thread indices (e.g., `A[idx]` pattern).

In short, use aligned, contiguous, warp-friendly access patterns.

E)

Each warp reads 128 bytes using 2 × 64-byte transactions. If 100 million warps execute per second:

Theoretical read bandwidth = 100 x 10^6 warps/s x 128 bytes/warp

= 12.8 x 10^9 bytes/s

= 12.8 GB/s