National University of Computer and Emerging Sciences

Islamabad Campus

High Performance Computing with GPUs (CS4110)

Final Examination

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

Total Time (Hrs): 3
Total Marks: 100
Total Questions: 7

Date: May 21, 2025

_____ _____ _____

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

Do not write below this line.

1. **Solve all questions.**
2. **Clearly mention the question number as well as the part number of each question.**
3. **The source code should be properly indented.**
4. **Simply providing the answer is not enough to get the marks. Justify your answers.**
5. **Solve all questions and all parts of each question in order to get a bonus of 2 marks.**

[CLO 1: Describe the terms related to HPC systems, GPU architecture and GPU prog. models.]

Q1: **[4+4+4+4 = 16 marks]**

Answer each of the following questions in as few words as you can. Your answer will be graded based on completeness, correctness, and conciseness.

A) Give two potential disadvantages associated with increasing the amount of work done in each CUDA thread, such as loop unrolling techniques, using fewer threads in total?

B) Provide two advantages of a programmer-managed memory, such as the CUDA shared memory, over a hardware managed cache?

C) How to achieve block-level and device-level synchronization in CUDA?

D) Provide one advantage and one disadvantage of using Tensor core instead of cuda core in a GPU architecture.

SOLUTION

A)

Increased register pressure

Lower occupancy

B)

Lower latency access

Predictable control of data reuse

C)

Block-level: __syncthreads()

Device-level: cudaDeviceSynchronize()

D)

Advantage: Higher throughput for matrix ops

Disadvantage: Requires specific data layout and types

[CLO 2: Identify application hotspots based on the application profile using realistic workload.]

Q2: **[2+4+1+1+2 = 10 marks]**

Given a GPU profiling output that shows the following:

Kernel A: 75% execution time, 10% memory bandwidth usage, 100% occupancy
Kernel B: 20% execution time, 80% memory bandwidth usage, 60% occupancy
Kernel C: 5% execution time, 40% memory bandwidth usage, 30% occupancy

A) Which kernel is the hotspot?
B) What does the profiling data suggest about potential bottlenecks (compute-bound vs memory-bound)?
C) Suggest one optimization for Kernel A.
D) Suggest one optimization for Kernel B.
E) If a kernel has high occupancy and high memory bandwidth utilization, it is guaranteed to be a hotspot? Yes/No? Why?

SOLUTION

A)

Kernel A as it has 75% execution time

B)

Kernel A: Compute-bound

Kernel B: Memory-bound

Kernel C: Underutilized (low occupancy, moderate memory usage)

C)

Improve instruction-level parallelism or math optimization

D)

Optimize memory access patterns (e.g., coalescing, caching)

E)

No. High occupancy and bandwidth don't mean long execution time; it may still be a small percentage of total runtime.

[CLO 3: Develop data-parallel solutions using appropriate programming models.]

Q3: **[3+5+2+4 = 14 marks]**

You are given a 1D float array z of length N stored on the GPU. This array represents the output of a neural network layer. You need to work on the development of a CUDA program to apply the ReLU activation function to each element of z, in place. The ReLU function is defined as:

$$ReLU(x)=max(0,x)$$

A) Write a CUDA kernel **relu_activation(float* z, int N)** that applies ReLU in-place to each element using GPU parallelism.

B) In the host code:

- Allocate and initialize memory for z on the host and device.
- Launch the kernel with appropriate block/grid dimensions.
- Properly copy data and print the output wherever required.
- Ensure your kernel handles the case when N is not divisible by the block size.

C) Will there be any advantage of using shared memory in this application? If yes, which data-structure should be mapped to shared memory?

D) What changes do you need to make if you are supposed to use unified memory? Only rewrite the lines that which will change.

SOLUTION

A)

```
__global__ void relu_activation(float* z, int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N) {
        z[idx] = fmaxf(0.0f, z[idx]);
    }
}
```

B)

```
#include <stdio.h>
#include <cuda_runtime.h>
#define N 1023
#define BLOCK_SIZE 256

int main() {
    float *h_z, *d_z;

    // Allocate host memory and initialize
    h_z = (float*)malloc(N * sizeof(float));
    for (int i = 0; i < N; i++)
        h_z[i] = (i % 10) - 5;  // Some positive and negative values

    // Allocate device memory
    cudaMalloc(&d_z, N * sizeof(float));
    cudaMemcpy(d_z, h_z, N * sizeof(float), cudaMemcpyHostToDevice);

    // Launch kernel
    int gridSize = (N + BLOCK_SIZE - 1) / BLOCK_SIZE;
```

```
    relu_activation<<<gridSize, BLOCK_SIZE>>>(d_z, N);
    cudaDeviceSynchronize();

    // Copy back and print
    cudaMemcpy(h_z, d_z, N * sizeof(float), cudaMemcpyDeviceToHost);
    for (int i = 0; i < 10; i++)  // Print first 10 values
        printf("%f ", h_z[i]);
    printf("\n");

    // Cleanup
    cudaFree(d_z);
    free(h_z);

    return 0;
}
```

C)

No significant advantage of shared memory here. Shared memory benefits data reuse or inter-thread communication, which ReLU doesn't require. Each thread accesses its own element in z, with no reuse.

D)

```
// Instead of malloc + cudaMalloc do
cudaMallocManaged(&h_z, N * sizeof(float));

// Initialization stays same
for (int i = 0; i < N; i++)
    h_z[i] = (i % 10) - 5;

// No cudaMemcpy needed
// Remove: cudaMemcpy and still do cudaFree for unified memory
```

```
cudaFree(h_z);
```

[CLO 3: Develop data-parallel solutions using appropriate programming models.]

Q4: **[5+3+2 = 10 marks]**

The 2D heat equation is discretized as:

$$\mathbf{Tnew[i,j]=T[i,j]+\alpha\cdot(T[i+1,j]+T[i-1,j]+T[i,j+1]+T[i,j-1]-4\cdot T[i,j])}$$

Where:

- T[i,j] is the temperature at cell (i,j)
- α is the thermal diffusivity (a small float constant, e.g., 0.1)

A) Write a CUDA kernel using @cuda.jit that computes one timestep update for an internal cell in a 2D grid T, storing the result in T_new. Use 2D grid and block configuration (i.e., cuda.grid(2)). Skip boundary cells (i=0, i=N-1, j=0, j=N-1).

B) Write the host code in which you allocate T as a 2D NumPy array (e.g., 256x256) with float32 values. Launch the kernel with appropriate 2D blocks and grids. Copy result back to host to print the updated temperature at the end.

C) Will there be any advantage of using shared memory in this application? If yes, which data-structure should be mapped to shared memory?

SOLUTION

A)

```
@cuda.jit
def heat_step(T, T_new, alpha):
    i, j = cuda.grid(2)

    if 1 <= i < T.shape[0] - 1 and 1 <= j < T.shape[1] - 1:
        T_new[i, j] = T[i, j] + alpha * (
            T[i+1, j] + T[i-1, j] + T[i, j+1] + T[i, j-1] - 4 * T[i, j]
```

```
    )
```

B)

```
import numpy as np
from numba import cuda

# Parameters
N = 256
alpha = 0.1
TPB_X, TPB_Y = 16, 16  # Threads per block

# Allocate and initialize
T = np.random.rand(N, N).astype(np.float32)
T_new = np.zeros_like(T)

# Copy to device
d_T = cuda.to_device(T)
d_T_new = cuda.to_device(T_new)

# Define grid size
blockspergrid_x = (N + TPB_X - 1) // TPB_X
blockspergrid_y = (N + TPB_Y - 1) // TPB_Y
blockspergrid = (blockspergrid_x, blockspergrid_y)
threadsperblock = (TPB_X, TPB_Y)

# Launch kernel
heat_step[blockspergrid, threadsperblock](d_T, d_T_new, alpha)
cuda.synchronize()

# Copy result back
T_result = d_T_new.copy_to_host()
```

```
# Print a few values for verification
print(T_result[100:105, 100:105])
```

C)

Yes, shared memory can be beneficial here as each thread accesses neighboring values (T[i±1, j], T[i, j±1]), so many threads redundantly load overlapping data from global memory.

T (in fact a tile of T) should be loaded into shared memory.

[CLO 4: Analyze the performance of an application running on an HPC system to improve compute and memory performance.]

Q5: **[6+4 = 10 marks]**

You are optimizing a CUDA kernel on an NVIDIA GPU with the following characteristics:

- Each Streaming Multiprocessor (SM) supports a maximum of 2048 active threads and 64 active warps.
- Each block you launch has 256 threads.
- The kernel uses 32 registers per thread.
- Each SM has a total of 65536 registers.
- Your kernel uses 4 KB of shared memory per block, and each SM has 48 KB shared memory available.

A) Calculate the maximum number of thread blocks that can reside on a single SM, considering the most limiting resource (threads, registers, or shared memory).
B) Based on the result, calculate the occupancy of the kernel as a percentage of the maximum number of threads per SM.

SOLUTION

GPU SM Limits and Kernel Usage Summary:

SM limits:

2048 threads per SM

64 warps per SM

65536 registers per SM

48 KB (49,152 bytes) shared memory per SM

Kernel usage per block:

256 threads → 256 / 32 = 8 warps

32 registers per thread → 256 × 32 = 8192 registers per block

4 KB (4096 bytes) shared memory per block

A) Find max resident blocks per SM per resource:

1. Thread limit: Max blocks by threads = 2048/256 = 8

2. Register limit: Max blocks by registers = 65536/8192 = 8

3. Shared memory limit: Max blocks by shared memory = 49152 / 4096 = 12

Most limiting resource = threads or registers (both result in 8 blocks max).

So the max blocks per SM = 8

B)

Each block has 256 threads

For 8 blocks: 8 × 256 = 2048 threads total

Occupancy=2048/2048=100%

[CLO 4: Analyze the performance of an application running on an HPC system to improve compute and memory performance.]

Q6: **[1+1+1+3+3+5 = 14 marks]**

For the vector addition kernel and the corresponding kernel launch code, answer each of the sub-questions below.

```
1    __global__
2    void vecAddKernel(float* A_d, float* B_d, float* C_d, int n)
3    {
4            int i = threadIdx.x + blockDim.x * blockIdx.x;
5            if(i<n) C_d[i] = A_d[i] + B_d[i];
6    }
7
8    int vectAdd(float* A, float* B, float* C, int n)
```

```
9   {
10          // assume size has been set to the actual length of
11          // arrays A, B, and C
12          int size = n * sizeof(float);
13
14          cudaMalloc((void **) &A_d, size);
15          cudaMalloc((void **) &B_d, size);
16          cudaMalloc((void **) &C_d, size);
17          cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
18          cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);
19          vecAddKernel<<<ceil(n/256), 256>>>(A_d, B_d, C_d, n);
20          cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);
21  }
```

A) Assume that the size of A, B, and C is 1000 elements. How many thread blocks will be generated?

B) Assume that the size of A, B, and C is 1000 elements. How many warps are there in each block?

C) Assume that the size of A, B, and C is 1000 elements. How many threads will be created in the grid?

D) Assume that the size of A, B, and C is 1000 elements. Is there any control divergence during the execution of the kernel? If so, identify the line number of the statement that causes the control divergence. Explain why or why not.

E) Assume that the size of A, B, and C is 768 elements. Is there any control divergence during the execution of the kernel? If so, identify the line number of the statement that causes the control divergence. Explain why or why not.

F) As discussed in class, data structure padding can be used to eliminate control divergence. Assuming that we will keep the host data structure size the same but pad the device data structure. Declare and initialize a new variable padded_size in line 13 and make some minor changes to statements in lines 14, 15, 16 and 19 to eliminate control divergence during the execution of the kernel. Assume that random input values to floating point addition operations will not cause any errors or exceptions.

SOLUTION

A)

blocks = ceil(1000 / 256) = 4

B)

Wraps = 256 / 32 = 8

C)

As Threads per block = 256

And Blocks = 4

So Threads = 256×4 = 1024

D)

Yes, control divergence occurs at line 5 due to the if statement. We launch 1024 threads but only need 1000. So threads with i = 1000 - 1023 will not execute the operation.

E)

No. Total threads in the grid will be 768 == size of the arrays, the last warp will not have divergence as all 32 threads will take the same path.

F)

To eliminate divergence, pad the device arrays so that total number of threads = multiple of block size (i.e., no thread needs if(i < n) check). Assume n = 1000. Nearest multiple of 256 ≥ 1000 = 1024

So we can declared size of required padding as

```
int padded_size = ((n + 255) / 256) * 256;

// updated memory allocation considering padded size
cudaMalloc((void**)&A_d, padded_size * sizeof(float));
cudaMalloc((void**)&B_d, padded_size * sizeof(float));
cudaMalloc((void**)&C_d, padded_size * sizeof(float));

// Zero out the padding
cudaMemset(A_d + n, 0, (padded_size - n) * sizeof(float));
cudaMemset(B_d + n, 0, (padded_size - n) * sizeof(float));
```

```
// updated kernel launch configuration
vecAddKernel<<<padded_size / 256, 256>>>(A_d, B_d, C_d, padded_size);
```

Now update the kernel to remove the check

```
__global__ void vecAddKernel(float* A_d, float* B_d, float* C_d, int n)
{
   int i = threadIdx.x + blockDim.x * blockIdx.x;
   C_d[i] = A_d[i] + B_d[i];
}
```

Q7: **[1 x 26 = 26 marks]**

Select the correct option for each of the following questions. Fill the attached bubble sheet to get marks. Multiple marking, overwriting, unclear marking will result in zero marks in that part.

1. Which Numba decorator is used to compile a CUDA kernel for execution on the GPU?

A) @jit

B) @njit

C) @cuda.jit

D) @vectorize

2. In Numba CUDA programming, what does cuda.grid(1) return?

A) The thread's position within a block

B) The number of blocks in the grid

C) The global thread ID in 1D grid

D) The block index along the X dimension

3. Which memory space in CUDA has the lowest latency but is only shared among threads in the same block?

A) Global memory

B) Shared memory

C) Local memory

D) Constant memory

4. What is a key requirement when copying NumPy arrays to the GPU using Numba?

A) They must be 64-bit integers

B) They must be 2D arrays

C) They must be Fortran-ordered

D) They must have a compatible dtype like float32 or int32

5. Which of the following correctly launches a 2D CUDA kernel in Numba?

A) kernel[blockDim](args)

B) kernel[gridDim, blockDim](args)

C) kernel.launch(gridDim, blockDim, args)

D) kernel<<<gridDim, blockDim>>>(args)

6. What causes shared memory bank conflicts in CUDA?

A) Threads accessing registers simultaneously

B) Multiple threads accessing different addresses in the same memory bank

C) Misaligned global memory loads

D) Atomic operations in global memory

7. How many shared memory banks exist per SM in most NVIDIA GPUs?

A) 8

B) 16

C) 32

D) 64

8. Which of the following access patterns causes a **broadcast** and avoids a bank conflict?

A) Thread 0 accesses address 0, thread 1 accesses address 1

B) All threads access the same address

C) Threads access addresses with stride of 2

D) Threads access addresses spaced by 17

9. What is one effective way to avoid shared memory bank conflicts in a 2D array?

A) Use more threads per block

B) Use local memory instead of shared

C) Add padding to array dimensions

D) Switch to global memory access

10. Which CUDA API enables allocation of unified memory?

A) cudaMallocHost

B) cudaMallocManaged

C) cudaHostAlloc

D) cudaUnifiedAlloc

11. What is the effect of setting a block size larger than the maximum supported by the GPU?

A) Kernel runs slower

B) Kernel launch fails at runtime

C) Occupancy increases

D) Threads are serialized

12. Which RAPIDS component is used for GPU-accelerated linear regression, clustering, and PCA?

A) cuGraph

B) cuML

C) cuDF

D) cuIO

13. What is occupancy in the context of CUDA performance optimization?

A) Percentage of cache hits per block

B) Memory bandwidth utilization

C) Ratio of active warps to maximum supported warps per SM**

D) Number of blocks per grid

14. A kernel uses 72 registers per thread and 48 KB of shared memory. What is the most likely cause if it only launches 4 blocks per SM?

A) Thread divergence

B) Register pressure or shared memory limits occupancy

C) Warp scheduler saturation

D) L2 cache congestion

15. What optimization explains the performance gain of loop unrolling inside a CUDA kernel with shared memory?

A) Avoids register spill

B) Reduces branch overhead and enables ILP (instruction-level parallelism)

C) Increases warp occupancy

D) Allows memory coalescing

16. On NVIDIA GPUs, the smallest execution unit that can be independently scheduled and issued is?

A) Thread

B) Warp

C) Thread block

D) SM

17. How are registers physically allocated in NVIDIA GPUs?

A) Per thread block

B) Per thread

C) Per warp

D) Globally shared across Sms

18. What is the function of the SFU (Special Function Unit) in NVIDIA SMs?

A) Handles memory transactions

B) Schedules warps with divergence

C) Executes transcendental operations (sin, exp, log)

D) Computes integer division only

19. In GPUs, what is the main advantage of using SIMT (Single Instruction, Multiple Thread) model?

A) Simplifies cache coherence

B) Reduces branching overhead

C) Allows massive parallelism with minimal control logic per thread

D) Improves double-precision performance

20. Why is the wmma API in CUDA used when programming with Tensor Cores?

A) To declare shared memory statically

B) To enable warp-level matrix multiply-accumulate using Tensor Cores

C) To align thread blocks to warps

D) To automatically tile memory accesses

21. In terms of performance, what is a major limitation when using Tensor Cores inefficiently?

A) Uses too much shared memory

B) Requires specific memory alignment and tiling for full utilization

C) Reduces warp occupancy

D) Increases register spilling

22. What is the tile size of matrix operands (A, B, C) in wmma APIs for FP16 on most GPUs?

A) 8×8

B) 16×16

C) 32×32

D) 64×64

23. Which of the following will prevent Tensor Cores from being used automatically in a matrix multiplication kernel?

A) Using shared memory

B) Using standard FP32 cublasSgemm with default math mode

C) Using NVIDIA A100

D) Compiling with -arch=sm_80

24. Which CUDA API would you use to measure kernel execution time?

A) cudaMemcpy

B) cudaEventRecord with cudaEventElapsedTime

C) cudaProfilerStart

D) cudaHostRegister

25. In CUDA-aware MPI, which of the following statements is true?

A) Device memory must be copied to host before sending via MPI

B) Only host pointers can be passed to MPI_Send

C) Device pointers can be used directly in MPI calls

D) CUDA streams are ignored in all MPI operations

26. What happens if you launch a kernel without setting the device in a multi-GPU environment?

A) All GPUs execute the kernel

B) No GPU executes the kernel

C) The kernel launches on the default device (usually device 0)

D) A runtime error occurs

SOLUTION

1. C: @cuda.jit
2. C: The global thread ID in 1D grid
3. B: Shared memory
4. D: They must have a compatible dtype like float32 or int32
5. B: kernelgridDim, blockDim
6. B: Multiple threads accessing different addresses in the same memory bank
7. C: 32
8. B: All threads access the same address
9. C: Add padding to array dimensions
10. B: cudaMallocManaged
11. B: Kernel launch fails at runtime
12. B: cuML
13. C: Ratio of active warps to maximum supported warps per SM
14. B: Register pressure or shared memory limits occupancy
15. B: Reduces branch overhead and enables ILP (instruction-level parallelism)
16. B: Warp
17. B: Per thread
18. C: Executes transcendental operations (sin, exp, log)
19. C: Allows massive parallelism with minimal control logic per thread
20. B: To enable warp-level matrix multiply-accumulate using Tensor Cores
21. B: Requires specific memory alignment and tiling for full utilization
22. B: 16×16
23. B: Using standard FP32 cublasSgemm with default math mode
24. B: cudaEventRecord with cudaEventElapsedTime
25. C: Device pointers can be used directly in MPI calls
26. C: The kernel launches on the default device (usually device 0)