National University of Computer and Emerging Sciences

Islamabad Campus

High Performance Computing with GPUs (CS4110)

Sessional-I Exam

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

Total Time (Hrs): 1
Total Marks: 43
Total Questions: 6

Date: Feb 25, 2025

_____ _____ _____

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

Do not write below this line.

Attempt all the questions.

1. **Solve all questions and all parts of each question in order.**
2. **The source code should be properly indented.**
3. **Simply providing the answer is not enough to get the marks. You need to justify your answers.**
4. **Write the question number as well as the part number of each question.**

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

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

A scientific simulation runs on a **CPU-only system** in **100 seconds**. A portion **P** of the program can be offloaded to a **GPU**, while the remaining **(1 - P)** must still be executed on the **CPU**. The GPU can execute the parallel portion **50× faster** than the CPU. In the light of these details, answer the following questions.

A) If **60% of the program can be offloaded to the GPU**, calculate the **overall speedup**.

B) What percentage of the program **must be offloaded to the GPU** to achieve at least a **20× speedup**?

C) Suppose **kernel launch overhead and memory transfers** introduce an **additional 5% execution time overhead** in the parallel portion. How does this impact the expected speedup?

A)

To calculate the overall speedup, we use Amdahl's Law:

$S = 1 / [(1 - P) + (P / S\_GPU)]$

where:

- S is the overall speedup,
- P = 0.6 (fraction of the program offloaded to the GPU),
- S_GPU = 50 (speedup factor of the GPU),

- 1 - P = 0.4 (fraction of the program still running on the CPU).

So

S = 1 / [(1 - 0.6) + (0.6 / 50)] ≈ 2.43

Therefore, overall speedup is approximately 2.43×.

B)

To achieve at least a 20× speedup, we use Amdahl's Law: S = 1 / [(1 - P) + (P / S_GPU)]

with

- S = 20 (desired speedup),

- S_GPU = 50 (GPU speedup factor).

Solving for P: (1 - P) + (P / 50) = 1 / 20 gives P ≈ 0.9694 (96.94%)

So 96.94% of the program must be offloaded to the GPU to achieve a 20× speedup.

C)

To determine the impact of a 5% execution time overhead, we modify Amdahl's Law:

S = 1 / [(1 - P) + (P / (S_GPU * (1 - Overhead)))]

Given:

- S_GPU = 50,

- Overhead = 0.05 (5% extra execution time),

- P = 96.94% (previously calculated for 20× speedup).

Adjusting for overhead, S = 1 / [0.0306 + (0.9694 / 47.5)] ≈ 19.61

With a 5% overhead, the speedup reduces from 20× to approximately 19.61×.

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

Q2: **[4+4 marks]**

You are developing a CUDA program which squares each element of the input array. For an **array of size N**, where N = 1000000 and the block size is 256,

A) Provide CUDA code for kernel launch configuration (ONLY) that ensures all elements are processed.

B) Provide the complete kernel. Inside the kernel, prevent **out-of-bounds** access.

A)

```
int N = 1000000; // Size of the array
int blockSize = 256;
int gridSize = (N + blockSize - 1) / blockSize; // Compute the number of blocks
// Kernel launch
squareKernel<<<gridSize, blockSize>>>(d_input, d_output, N);
```

B)

```
__global__ void squareKernel(float* d_input, float* d_output, int N)
{
   int idx = blockIdx.x * blockDim.x + threadIdx.x;
   // Prevent out-of-bounds access
   if (idx < N)
   {
      d_output[idx] = d_input[idx] * d_input[idx];
   }
}
```

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

Q3: **[1+2 marks]**

Given the kernel launch:

```
myKernel<<<64, 512>>>(d_data);
```

A) How many total threads are launched?

B) Modify the launch configuration to maintain the same total threads but with 256 threads per block instead of 512.

A) Total Threads = Grid Size × Block Size = 64*512 = 32768

B) myKernel<<<128, 256>>>(d_data);

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

Q4: **[4 marks]**

A CUDA kernel is launched with:

```
dim3 blockSize(16, 16);
```

```
dim3 gridSize(32, 32);
myKernel<<<gridSize, blockSize>>>(d_data);
```

How would you compute a **unique 2D global index** (global_x, global_y) for each thread inside the kernel?

```
int global_x = blockIdx.x * blockDim.x + threadIdx.x;
int global_y = blockIdx.y * blockDim.y + threadIdx.y;
```

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

Q5: **[10 marks]**

Given the following host side of CUDA program. Point out the error(s) and fix them.

```
int main() {
  int N = 1000;
  int * h_array, * d_array;

  // Allocate memory on device
  cudaMalloc(d_array, N * sizeof(int));

  // Allocate memory on host
  h_array = (int * ) malloc(N * sizeof(int));

  // Copy data from host to device
  cudaMemcpy(h_array, d_array, N * sizeof(int), cudaMemcpyDeviceToHost);

  // Initialize host array
  for (int i = 0; i < N; i++) {
    h_array[i] = i;
  }

  // Launch kernel
  kernel << < (N / 256), 256 >>> (d_array, N);

  // Copy results back to host
  cudaMemcpy(d_array, h_array, N * sizeof(int), cudaMemcpyHostToDevice);

  // Free memory
  cudaFree(d_array);
  free(h_array);
```

```
 return 0;
}
```

Summary fixes:

1. Fixed `cudaMalloc` argument casting.
2. Corrected `cudaMemcpy` calls by swapping source and destination where needed.
3. Fixed kernel launch syntax and ensured full coverage of `N` elements.
4. Added cudaDeviceSynchronize.
5. Corrected `cudaMemcpy` direction for copying results back.

Fixed code

```
int main() {
 int N = 1000;
 int *h_array, *d_array;

 // Allocate memory on device
 cudaMalloc((void**)&d_array, N * sizeof(int));

 // Allocate memory on host
 h_array = (int*)malloc(N * sizeof(int));

 // Initialize host array
 for (int i = 0; i < N; i++) {
  h_array[i] = i;
 }

 // Copy data from host to device
 cudaMemcpy(d_array, h_array, N * sizeof(int), cudaMemcpyHostToDevice);
```

```
 // Launch kernel
 kernel<<<(N + 255) / 256, 256>>>(d_array, N);

// Ensure the kernel execution is completed before proceeding
cudaDeviceSynchronize();

 // Copy results back to host
 cudaMemcpy(h_array, d_array, N * sizeof(int), cudaMemcpyDeviceToHost);

 // Free memory
 cudaFree(d_array);
 free(h_array);

 return 0;
}
```

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

Q6: **[3+3 marks]**

Compare and contrast the following by providing at least 1 example for each.

- **A)** Homogeneous vs Heterogeneous computing
- **B)** Grid dimensions vs Block dimensions

A) Homogeneous vs Heterogeneous Computing

Homogeneous Computing: Uses a single type of processor for all computations. Example: A program running entirely on a multi-core CPU.

Heterogeneous Computing: Uses multiple types of processors (e.g., CPU + GPU) for different tasks. Example: Machine learning inference using a CPU for data preprocessing and a GPU for model execution.

B) Grid Dimensions vs Block Dimensions

Grid Dimensions: Represents the number of blocks in a CUDA kernel launch. Defines how work is distributed across blocks. Example: `dim3 grid(32, 32);` means a 2D grid of 32 × 32 blocks.
Block Dimensions: Represents the number of threads per block, defining parallelism within a block. Example: `dim3 block(16, 16);` means each block contains 16 × 16 = 256 threads.