PART A

1

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main() {
    long num_steps = 100000000; // Number of iterations for the
approximation
    double step = 1.0 / num_steps;
    double pi = 0.0;

    // Start parallel region
    #pragma omp parallel
    {
        double sum = 0.0;
        int i;
        #pragma omp for
        for (i = 0; i < num_steps; i++) {
            double x = (i + 0.5) * step;
            sum += 4.0 / (1.0 + x * x);
        }

        // Use a critical section to safely update the shared variable
`pi`
        #pragma omp critical
        {
            pi += sum;
        }
    }

    // Multiply the result by the step size to get the final value of
pi
    pi *= step;

    printf("Calculated value of PI: %.15f\n", pi);

    return 0;
}
```

2

```
#include <stdio.h>
#include <omp.h>

int main() {
    // Print basic OpenMP environment information
```

```
    printf("OpenMP Environment Information:\n");

    // Parallel region to get thread-specific information
    #pragma omp parallel
    {
        int thread_id = omp_get_thread_num();     // Get the thread ID
        int num_threads = omp_get_num_threads(); // Get the total
number of threads
        int num_procs = omp_get_num_procs();     // Get the number of
processors
        int max_threads = omp_get_max_threads(); // Get the maximum
number of threads
        int in_parallel = omp_in_parallel();     // Check if in
parallel region

        #pragma omp critical
        {
            printf("Thread ID: %d\n", thread_id);
            printf("  Total Threads: %d\n", num_threads);
            printf("  Number of Processors: %d\n", num_procs);
            printf("  Max Threads: %d\n", max_threads);
            printf("  In Parallel Region: %s\n", in_parallel ? "Yes" :
"No");
            printf("---------------------------------------\n");
        }
    }

    // Print outside the parallel region
    printf("Outside Parallel Region:\n");
    printf("  Max Threads: %d\n", omp_get_max_threads());
    printf("  Number of Processors: %d\n", omp_get_num_procs());
    printf("---------------------------------------\n");

    return 0;
}
```

3

```
#include <stdio.h>
#include <omp.h>

#define SIZE 1000  // Size of the arrays

int main() {
    int A[SIZE], B[SIZE], C[SIZE]; // Arrays for addition
    int i;

    // Initialize arrays A and B
    for (i = 0; i < SIZE; i++) {
        A[i] = i;
```

```
        B[i] = SIZE - i;
    }

    // Add arrays A and B in parallel to store the result in array C
    #pragma omp parallel for schedule(dynamic, 10)
    for (i = 0; i < SIZE; i++) {
        C[i] = A[i] + B[i];
    }

    // Print a few elements of the result array C
    printf("First 10 elements of the result array C:\n");
    for (i = 0; i < 10; i++) {
        printf("C[%d] = %d\n", i, C[i]);
    }

    return 0;
}
```

4

```
#include <stdio.h>
#include <omp.h>

#define SIZE 1000  // Size of the arrays

int main() {
    int A[SIZE], B[SIZE], Sum[SIZE], Product[SIZE];
    int i;

    // Initialize arrays A and B
    for (i = 0; i < SIZE; i++) {
        A[i] = i + 1;
        B[i] = (SIZE - i);
    }

    // Parallel region with work-sharing using sections
    #pragma omp parallel
    {
        #pragma omp sections
        {
            // Section for addition
            #pragma omp section
            {
                for (i = 0; i < SIZE; i++) {
                    Sum[i] = A[i] + B[i];
                }
                printf("Addition done by thread %d\n",
omp_get_thread_num());
            }
```

```
            // Section for multiplication
            #pragma omp section
            {
                for (i = 0; i < SIZE; i++) {
                    Product[i] = A[i] * B[i];
                }
                printf("Multiplication done by thread %d\n",
omp_get_thread_num());
            }
        }
    }

    // Print the first 10 results from both operations
    printf("First 10 elements of Sum array:\n");
    for (i = 0; i < 10; i++) {
        printf("Sum[%d] = %d\n", i, Sum[i]);
    }

    printf("\nFirst 10 elements of Product array:\n");
    for (i = 0; i < 10; i++) {
        printf("Product[%d] = %d\n", i, Product[i]);
    }

    return 0;
}
```

5

```
#include <stdio.h>
#include <omp.h>

#define N 500  // Dimensions of the matrices (N x N)

int main() {
    int A[N][N], B[N][N], C[N][N];
    int i, j, k;

    // Initialize matrices A and B with some values
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            A[i][j] = i + j;
            B[i][j] = i - j;
        }
    }

    // Initialize matrix C to 0
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            C[i][j] = 0;
        }
```

```c
    }

    // Parallelized matrix multiplication
    #pragma omp parallel for private(i, j, k) shared(A, B, C)
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            for (k = 0; k < N; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }

    // Print a small section of the result matrix for verification
    printf("Result matrix C (5x5):\n");
    for (i = 0; i < 5; i++) {
        for (j = 0; j < 5; j++) {
            printf("%d ", C[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

6

```c
#include <stdio.h>
#include <omp.h>

int main() {
    int x = 10;  // Variable to demonstrate firstprivate
    int thread_id;

    printf("Initial value of x: %d\n", x);

    // Start parallel region
    #pragma omp parallel firstprivate(x) private(thread_id)
    {
        thread_id = omp_get_thread_num();
        x += thread_id;  // Each thread modifies its own private copy
of x

        printf("Thread %d: x = %d\n", thread_id, x);
    }

    // x remains unchanged outside the parallel region
    printf("Value of x after parallel region: %d\n", x);

    return 0;
}
```

7

```c
#include <stdio.h>
#include <omp.h>

#define SIZE 1000  // Size of the vector

int main() {
    int vector[SIZE];
    int i;
    long sum = 0;  // Variable to store the sum of elements

    // Initialize the vector with values
    for (i = 0; i < SIZE; i++) {
        vector[i] = i + 1;  // Fill vector with values 1 to SIZE
    }

    // Parallel region to calculate the sum using reduction
    #pragma omp parallel for reduction(+:sum)
    for (i = 0; i < SIZE; i++) {
        sum += vector[i];  // Add elements to sum
    }

    printf("Sum of all elements in the vector: %ld\n", sum);

    return 0;
}
```

PART B

1

```
%%cuda
#include <stdio.h>
// CUDA kernel to add two numbers
__global__ void addNumbers(int *a, int *b, int *c) {
*c = *a + *b; // Add the two numbers
}
int main() {
int a = 5, b = 7; // Numbers to add
int c; // Result of addition
int *d_a, *d_b, *d_c; // Device pointers
// Allocate memory on the device (GPU)
cudaMalloc((void **)&d_a, sizeof(int));
cudaMalloc((void **)&d_b, sizeof(int));
cudaMalloc((void **)&d_c, sizeof(int));
// Check if memory allocation was successful
if (d_a == NULL || d_b == NULL || d_c == NULL) {
printf("Failed to allocate device memory.\n");
return -1;
```

```
}
// Copy input data from host to device
cudaMemcpy(d_a, &a, sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(d_b, &b, sizeof(int), cudaMemcpyHostToDevice);
// Launch the kernel (1 block, 1 thread)
addNumbers<<<1, 1>>>(d_a, d_b, d_c);
// Synchronize device to ensure kernel execution completes
cudaDeviceSynchronize();
// Copy the result from device to host
cudaMemcpy(&c, d_c, sizeof(int), cudaMemcpyDeviceToHost);
// Print the result
printf("The sum of %d and %d is %d\n", a, b, c);
// Free device memory
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
return 0;
}
```

2

```
%%cuda
#include <stdio.h>
#include <cuda_runtime.h>
#define SIZE 64
__global__ void addVector(int *a, int *b, int *c) {
int tid = blockIdx.x * blockDim.x + threadIdx.x;
if (tid < SIZE) {
c[tid] = a[tid] + b[tid];
}
}
void randomNumberAssigner(int *array, int size) {
for (int i = 0; i < size; i++) {
array[i] = rand() % 100;
}
}
int main() {
int *a, *b, *c;
int *d_a, *d_b, *d_c;
printf("In main function\n");
a = (int*) malloc(SIZE * sizeof(int));
b = (int*) malloc(SIZE * sizeof(int));
c = (int*) malloc(SIZE * sizeof(int));
randomNumberAssigner(a, SIZE);
randomNumberAssigner(b, SIZE);
cudaMalloc((void**) &d_a, SIZE * sizeof(int));
cudaMalloc((void**) &d_b, SIZE * sizeof(int));
cudaMalloc((void**) &d_c, SIZE * sizeof(int));
cudaMemcpy(d_a, a, SIZE * sizeof(int), cudaMemcpyHostToDevice);
```

```
cudaMemcpy(d_b, b, SIZE * sizeof(int), cudaMemcpyHostToDevice);
addVector<<<2,32>>>(d_a, d_b, d_c);
cudaMemcpy(c, d_c, SIZE * sizeof(int), cudaMemcpyDeviceToHost);
for (int i = 0; i < SIZE; i++) {
printf("%d + %d = %d\n", a[i], b[i], c[i]);
}
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
free(a);
free(b);
free(c);
return 0;
}
```

3

```
%%cuda
#include <stdio.h>
#include <cuda_runtime.h>
#define ROW 10
#define COL 10
__global__ void addMatrix(int *a, int *b, int *c) {
int row = blockIdx.y * blockDim.y + threadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.x;
if (row < ROW && col < COL) {
c[row * COL + col] = a[row * COL + col] + b[row * COL + col];
}
}
void randomNumberAssigner(int *array, int size) {
for (int i = 0; i < size; i++) {
array[i] = rand() % 10;
}
}
int main() {
int *a, *b, *c;
int *d_a, *d_b, *d_c;
int size = ROW * COL;
printf("In main function\n");
a = (int*) malloc(size * sizeof(int));
b = (int*) malloc(size * sizeof(int));
c = (int*) malloc(size * sizeof(int));
randomNumberAssigner(a, size);
randomNumberAssigner(b, size);
cudaMalloc((void**) &d_a, size * sizeof(int));
cudaMalloc((void**) &d_b, size * sizeof(int));
cudaMalloc((void**) &d_c, size * sizeof(int));
cudaMemcpy(d_a, a, size * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size * sizeof(int), cudaMemcpyHostToDevice);
```

```
dim3 threadsPerBlock(COL, ROW);
dim3 numBlocks(1,1);
// int tbp = 64, nb = ((ROW*COL)+tbp-1)/tbp;
addMatrix<<<threadsPerBlock, numBlocks>>>(d_a, d_b, d_c);
cudaMemcpy(c, d_c, size * sizeof(int), cudaMemcpyDeviceToHost);
printf("Matrix A:\n");
for (int i = 0; i < ROW; i++) {
for(int j = 0; j < COL; j++)
printf("%d\t", a[i*COL+j]);
printf("\n");
}
printf("Matrix B:\n");
for (int i = 0; i < ROW; i++) {
for(int j = 0; j < COL; j++)
printf("%d\t", b[i*COL+j]);
printf("\n");
}
printf("Matrix C:\n");
for (int i = 0; i < ROW; i++) {
for (int j = 0; j < COL; j++)
printf("%d\t", c[i*COL+j]);
printf("\n");
}
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
free(a);
free(b);
free(c);
return 0;
}
```

5

```
%%cuda
#include <stdio.h>
#include <cuda_runtime.h>
#define ROW_A 10
#define COL_A 5
#define ROW_B COL_A
#define COL_B 8
__global__ void matrixMul(int *a, int *b, int *c) {
int row = blockIdx.y * blockDim.y + threadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.x;
if (row < ROW_A && col < COL_B) {
int sum = 0;
for (int k = 0; k < COL_A; k++) {
sum += a[row * COL_A + k] * b[k * COL_B + col];
}
```

```
c[row * COL_B + col] = sum;
}
}
void randomNumberAssigner(int *array, int size) {
for (int i = 0; i < size; i++) {
array[i] = rand() % 10;
}
}
int main() {
int *a, *b, *c;
int *d_a, *d_b, *d_c;
int size_a = ROW_A * COL_A;
int size_b = ROW_B * COL_B;
int size_c = ROW_A * COL_B;
printf("In main function\n");
a = (int*) malloc(size_a * sizeof(int));
b = (int*) malloc(size_b * sizeof(int));
c = (int*) malloc(size_c * sizeof(int));
randomNumberAssigner(a, size_a);
randomNumberAssigner(b, size_b);
cudaMalloc((void**) &d_a, size_a * sizeof(int));
cudaMalloc((void**) &d_b, size_b * sizeof(int));
cudaMalloc((void**) &d_c, size_c * sizeof(int));
cudaMemcpy(d_a, a, size_a * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size_b * sizeof(int), cudaMemcpyHostToDevice);
//dim3 threadsPerBlock(COL, ROW);
//dim3 numBlocks(1,1);
int tbp = 32;
dim3 threadsPerBlock(tbp, tbp);
dim3 numBlocks((COL_B+threadsPerBlock.x-1)/threadsPerBlock.x,
(ROW_A+threadsPerBlock.y-1)/threadsPerBlock.y);
matrixMul<<<numBlocks, threadsPerBlock>>>(d_a, d_b, d_c);
cudaMemcpy(c, d_c, size_c * sizeof(int), cudaMemcpyDeviceToHost);
printf("Matrix A:\n");
for (int i = 0; i < ROW_A; i++) {
for(int j = 0; j < COL_A; j++)
printf("%d\t", a[i*COL_A+j]);
printf("\n");
}
printf("Matrix B:\n");
for (int i = 0; i < ROW_B; i++) {
for(int j = 0; j < COL_B; j++)
printf("%d\t", b[i*COL_B+j]);
printf("\n");
}
printf("Matrix C:\n");
for (int i = 0; i < ROW_A; i++) {
for (int j = 0; j < COL_B; j++)
printf("%d\t", c[i*COL_B+j]);
```

```
printf("\n");
}
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
free(a);
free(b);
free(c);
return 0;
}
```

6

```
%%cuda
#include <stdio.h>
#include <cuda_runtime.h>
#define ROW_A 2
#define COL_A 3
#define ROW_B COL_A
#define COL_B 2
__global__ void transpose(int *input, int *output, int rows, int cols)
{
int row = blockIdx.y * blockDim.y + threadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.x;
if (row < rows && col < cols) {
output[col * rows + row] = input[row * cols + col];
}
}
__global__ void matrixMul(int *a, int *b, int *c, int row_a, int
col_a, int col_b) {
int row = blockIdx.y * blockDim.y + threadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.x;
if (row < row_a && col < col_b) {
int sum = 0;
for (int k = 0; k < col_a; k++) {
sum += a[row * col_a + k] * b[k * col_b + col];
}
c[row * col_b + col] = sum;
}
}
int main() {
int a[ROW_A * COL_A] = {1, 2, 3, 4, 5, 6};
int b[ROW_B * COL_B] = {7, 8, 9, 10, 11, 12};
int c[ROW_A * COL_B], tc[ROW_A * COL_B];
int ta[COL_A * ROW_A], tb[COL_B * ROW_B];
int *d_a, *d_b, *d_c, *d_ta, *d_tb, *d_tc;
cudaMalloc(&d_a, ROW_A * COL_A * sizeof(int));
cudaMalloc(&d_b, ROW_B * COL_B * sizeof(int));
cudaMalloc(&d_c, ROW_A * COL_B * sizeof(int));
```

```
cudaMalloc(&d_ta, COL_A * ROW_A * sizeof(int));
cudaMalloc(&d_tb, COL_B * ROW_B * sizeof(int));
cudaMalloc(&d_tc, ROW_A * COL_B * sizeof(int));
cudaMemcpy(d_a, a, ROW_A * COL_A * sizeof(int),
cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, ROW_B * COL_B * sizeof(int),
cudaMemcpyHostToDevice);
dim3 threadsPerBlock(16, 16);
dim3 numBlocksA((COL_A + threadsPerBlock.x - 1) / threadsPerBlock.x,
(ROW_A + threadsPerBlock.y-1) / threadsPerBlock.y);
dim3 numBlocksB((COL_B + threadsPerBlock.x - 1) / threadsPerBlock.x,
(ROW_B + threadsPerBlock.y-1) / threadsPerBlock.y);
dim3 numBlocksC((COL_B + threadsPerBlock.x - 1) / threadsPerBlock.x,
(ROW_A + threadsPerBlock.y - 1) / threadsPerBlock.y);
transpose<<<numBlocksA, threadsPerBlock>>>(d_a, d_ta, ROW_A, COL_A);
transpose<<<numBlocksB, threadsPerBlock>>>(d_b, d_tb, ROW_B, COL_B);
matrixMul<<<numBlocksC, threadsPerBlock>>>(d_a, d_b, d_c, ROW_A,
COL_A, COL_B);
matrixMul<<<numBlocksC, threadsPerBlock>>>(d_ta, d_tb, d_tc, COL_A,
ROW_A, ROW_B);
cudaMemcpy(c, d_c, ROW_A * COL_B * sizeof(int),
cudaMemcpyDeviceToHost);
cudaMemcpy(tc, d_tc, ROW_A * COL_B * sizeof(int),
cudaMemcpyDeviceToHost);
//Verification
printf("Matrix A:\n");
for (int i = 0; i < ROW_A; i++){
for(int j = 0; j < COL_A; j++) {
printf("%d\t", a[i*COL_A+j]);
}
printf("\n");
}
printf("Matrix B:\n");
for (int i = 0; i < ROW_B; i++){
for(int j = 0; j < COL_B; j++) {
printf("%d\t", b[i*COL_B+j]);
}
printf("\n");
}
printf("Matrix C:\n");
for (int i = 0; i < ROW_A; i++){
for(int j = 0; j < COL_B; j++) {
printf("%d\t", c[i*COL_B+j]);
}
printf("\n");
}
printf("\nMatrix TC:\n");
for(int i = 0; i < ROW_A; i++) {
for (int j = 0; j < COL_B; j++){
```

```
printf("%d\t", tc[i * COL_B + j]);
}
printf("\n");
}
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
cudaFree(d_ta);
cudaFree(d_tb);
cudaFree(d_tc);
return 0;
}
```