B3 : OpenMP

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

int main() {
    int A[100][100], B[100][100], C[100][100], D[100][100];
    int X = 5, Y = 10;
    int N = 10;

#pragma omp parallel private(A, B) shared(C, D) firstprivate(X, Y)
    {
#pragma omp single copyprivate(A, B)
        {
            printf("Thread ID is : %d\n", omp_get_thread_num());
            for (int i = 0; i < N; i++) {
                for (int j = 0; j < N; j++) {
                    A[i][j] = rand() % 100;
                    B[i][j] = rand() % 100;
                }
            }
        }

#pragma omp for schedule(runtime) collapse(2)
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
#pragma omp critical
            {
                C[i][j] = A[i][j] + B[i][j];
            }
        }
    }

#pragma omp for schedule(runtime) collapse(2)
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if (C[i][j] % 2 != 0) {
                D[i][j] = C[i][j] - X;
            }
            else {
                D[i][j] = C[i][j] + Y;
            }
        }
    }

#pragma omp single
    {
        printf("A, B\n");
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                printf("%d,%d  ", A[i][j], B[i][j]);
            }
            printf("\n");
        }

        printf("C, D\n");
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                printf("%d,%d  ", C[i][j], D[i][j]);
            }
            printf("\n");
        }
```

```
    }
    }
    return 0;
}
```

output ::

Thread ID is : 1

A, B

41,67 34,0 69,24 78,58 62,64 5,45 81,27 61,91 95,42 27,36

91,4 2,53 92,82 21,16 18,95 47,26 71,38 69,12 67,99 35,94

3,11 22,33 73,64 41,11 53,68 47,44 62,57 37,59 23,41 29,78

16,35 90,42 88,6 40,42 64,48 46,5 90,29 70,50 6,1 93,48

29,23 84,54 56,40 66,76 31,8 44,39 26,23 37,38 18,82 29,41

33,15 39,58 4,30 77,6 73,86 21,45 24,72 70,29 77,73 97,12

86,90 61,36 55,67 55,74 31,52 50,50 41,24 66,30 7,91 7,37

57,87 53,83 45,9 9,58 21,88 22,46 6,30 13,68 0,91 62,55

10,59 24,37 48,83 95,41 2,50 91,36 74,20 96,21 48,99 68,84

81,34 53,99 18,38 0,88 27,67 28,93 48,83 7,21 10,17 13,14

C, D

108,118 34,44 93,88 136,146 126,136 50,60 108,118 152,162 137,132 63,58

95,90 55,50 174,184 37,32 113,108 73,68 109,104 81,76 166,176 129,124

14,24 55,50 137,132 52,62 121,116 91,86 119,114 96,106 64,74 107,102

51,46 132,142 94,104 82,92 112,122 51,46 119,114 120,130 7,2 141,136

52,62 138,148 96,106 142,152 39,34 83,78 49,44 75,70 100,110 70,80

48,58 97,92 34,44 83,78 159,154 66,76 96,106 99,94 150,160 109,104

176,186 97,92 122,132 129,124 83,78 100,110 65,60 96,106 98,108 44,54

144,154 136,146 54,64 67,62 109,104 68,78 36,46 81,76 91,86 117,112

69,64 61,56 131,126 136,146 52,62 127,122 94,104 117,112 147,142 152,162

115,110 152,162 56,66 88,98 94,104 121,116 131,126 28,38 27,22 27,22

B3 – Cuda

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <malloc.h>
#include <stdio.h>
```

```
#include <stdlib.h>

__global__ void sumkernel(int* A, int* B, int* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    int j = blockDim.y * blockIdx.y + threadIdx.y;
    C[i * N + j] = A[i * N + j] + B[i * N + j];
}

__global__ void sumrowkernel(int* C, int* D, int N)
{
    int i = blockDim.y * blockIdx.y + threadIdx.y;
    int sum = 0;
    for (int j = 0; j < N; j++) {
        sum += C[i * N + j];
    }
    if (sum > 10) {
        for (int j = 0; j < N; j++) {
            D[i * N + j] = 0;
        }
    }
    else {
        for (int j = 0; j < N; j++) {
            D[i * N + j] = 1;
        }
    }
}

int main()
{
    int N = 3;
    int* A, * B, * C, * D, * dev_A, * dev_B, * dev_C, * dev_D;
    A = (int*)malloc(N * N * sizeof(int));
    B = (int*)malloc(N * N * sizeof(int));
    C = (int*)malloc(N * N * sizeof(int));
    D = (int*)malloc(N * N * sizeof(int));

    for (int i = 0; i < N * N; i++) {
        A[i] = rand() % 10;
        B[i] = rand() % 10;
    }

    A[0] = 1;
    A[1] = 1;
    A[2] = 1;
    B[0] = 1;
    B[1] = 1;
    B[2] = 1;

    cudaMalloc((void**)&dev_A, N * N * sizeof(int));
    cudaMalloc((void**)&dev_B, N * N * sizeof(int));
    cudaMalloc((void**)&dev_C, N * N * sizeof(int));
    cudaMalloc((void**)&dev_D, N * N * sizeof(int));

    cudaMemcpy(dev_A, A, N * N * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_B, B, N * N * sizeof(int), cudaMemcpyHostToDevice);

    dim3 threads(N, N, 1);
    dim3 blocks(1, 1, 1);

    sumkernel << <blocks, threads >> > (dev_A, dev_B, dev_C, N);

    cudaMemcpy(C, dev_C, N * N * sizeof(int), cudaMemcpyDeviceToHost);
```

```
    sumrowkernel << <blocks, threads >> > (dev_C, dev_D, N);

    cudaMemcpy(D, dev_D, N * N * sizeof(int), cudaMemcpyDeviceToHost);

    printf("A, B\n");
    for (int i = 0; i < N * N; i++) {
        printf("%d,%d  ", A[i], B[i]);
    }
    printf("\n");

    printf("C, D\n");
    for (int i = 0; i < N * N; i++) {
        printf("%d,%d  ", C[i], D[i]);
    }
    printf("\n");

    cudaFree(dev_A);
    cudaFree(dev_B);
    cudaFree(dev_C);
    cudaFree(dev_D);
    free(A);
    free(B);
    free(C);
    free(D);

    return 0;
}
```

Output ::

A, B

1,1 1,1 1,1 8,8 2,4 5,5 1,7 1,1 5,2

C, D

2,1 2,1 2,1 16,0 6,0 10,0 8,0 2,0 7,0

A2-Cuda

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>

__global__ void elementWiseMultiply(int* A, int* B, int* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    int j = blockDim.y * blockIdx.y + threadIdx.y;
    C[i * N + j] = A[i * N + j] * B[i * N + j];
}

__global__ void columnSumThreshold(int* C, int* D, int N)
{
    int j = blockDim.x * blockIdx.x + threadIdx.x;
    int sum = 0;
```

```
    for (int i = 0; i < N; i++) {
        sum += C[i * N + j];
    }
    if (sum > 10) {
        for (int i = 0; i < N; i++) {
            D[i * N + j] = 0;
        }
    }
    else {
        for (int i = 0; i < N; i++) {
            D[i * N + j] = 1;
        }
    }
}

int main()
{
    int N = 3;
    int* A, * B, * C, * D, * dev_A, * dev_B, * dev_C, * dev_D;
    A = (int*)malloc(N * N * sizeof(int));
    B = (int*)malloc(N * N * sizeof(int));
    C = (int*)malloc(N * N * sizeof(int));
    D = (int*)malloc(N * N * sizeof(int));

    for (int i = 0; i < N * N; i++) {
        A[i] = rand() % 10;
        B[i] = rand() % 10;
    }

    cudaMalloc((void**)&dev_A, N * N * sizeof(int));
    cudaMalloc((void**)&dev_B, N * N * sizeof(int));
    cudaMalloc((void**)&dev_C, N * N * sizeof(int));
    cudaMalloc((void**)&dev_D, N * N * sizeof(int));

    cudaMemcpy(dev_A, A, N * N * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_B, B, N * N * sizeof(int), cudaMemcpyHostToDevice);

    dim3 threads(N, N, 1);
    dim3 blocks(1, 1, 1);

    elementWiseMultiply << <blocks, threads >> > (dev_A, dev_B, dev_C, N);

    cudaMemcpy(C, dev_C, N * N * sizeof(int), cudaMemcpyDeviceToHost);

    columnSumThreshold << <blocks, N >> > (dev_C, dev_D, N);

    cudaMemcpy(D, dev_D, N * N * sizeof(int), cudaMemcpyDeviceToHost);

    printf("A, B\n");
    for (int i = 0; i < N * N; i++) {
        printf("%d,%d  ", A[i], B[i]);
    }
    printf("\n");

    printf("C, D\n");
    for (int i = 0; i < N * N; i++) {
        printf("%d,%d  ", C[i], D[i]);
    }
    printf("\n");

    cudaFree(dev_A);
    cudaFree(dev_B);
    cudaFree(dev_C);
```

```
    cudaFree(dev_D);
    free(A);
    free(B);
    free(C);
    free(D);

    return 0;
}
```

A2 – MPI

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

#define N 4 // Size of the matrix

int main(int argc, char** argv) {
    int rank, size;
    int matrix[N][N] = {
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12},
        {13, 14, 15, 16}
    };
    int modified_matrix[N][N];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (rank == 0) {
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                modified_matrix[i][j] = matrix[i][j];
            }
        }
    }

    MPI_Bcast(modified_matrix, N * N, MPI_INT, 0, MPI_COMM_WORLD);

    int local_row[N];
    MPI_Scatter(modified_matrix, N, MPI_INT, local_row, N, MPI_INT, 0,
MPI_COMM_WORLD);

    for (int i = 0; i < N; i++) {
        if (local_row[i] % 2 == 0) {
            local_row[i] = 1;
        }
        else {
            local_row[i] = 0;
        }
    }

    MPI_Gather(local_row, N, MPI_INT, modified_matrix, N, MPI_INT, 0,
MPI_COMM_WORLD);

    if (rank == 0) {
        printf("The final matrix after replacement:\n");
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                printf("%d ", modified_matrix[i][j]);
```

```
            }
            printf("\n");
        }
    }

    MPI_Finalize();
    return 0;
}
```

Vector Elementtwise

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>

#define N 770
#define THREADS_PER_BLOCK 256

_global_ void calculateForce(float* acc, float* mass, float* force) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < N) {
        force[idx] = acc[idx] * mass[idx];
    }
}

int main() {
    float* acc, * mass, * force;
    float* dev_acc, * dev_mass, * dev_force;

    // Allocate memory for acceleration, mass, and force vectors on host
    acc = (float*)malloc(N * sizeof(float));
    mass = (float*)malloc(N * sizeof(float));
    force = (float*)malloc(N * sizeof(float));

    // Initialize acceleration and mass vectors
    for (int i = 0; i < N; ++i) {
        acc[i] = 9.81f; // Example value for acceleration (gravity)
        mass[i] = 1.0f; // Example value for mass
    }

    // Allocate memory for acceleration, mass, and force vectors on device
    cudaMalloc((void**)&dev_acc, N * sizeof(float));
    cudaMalloc((void**)&dev_mass, N * sizeof(float));
    cudaMalloc((void**)&dev_force, N * sizeof(float));

    // Copy acceleration and mass vectors from host to device
    cudaMemcpy(dev_acc, acc, N * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_mass, mass, N * sizeof(float), cudaMemcpyHostToDevice);

    // Calculate number of blocks needed
    int numBlocks = (N + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;

    // Launch kernel to calculate forces
    calculateForce << <numBlocks, THREADS_PER_BLOCK >> > (dev_acc, dev_mass,
dev_force);

    // Copy force vector from device to host
    cudaMemcpy(force, dev_force, N * sizeof(float), cudaMemcpyDeviceToHost);
```

```
    // Print the forces (optional)
    printf("Forces:\n");
    for (int i = 0; i < N; ++i) {
        printf("Particle %d: Force = %.2f\n", i, force[i]);
    }

    // Free device memory
    cudaFree(dev_acc);
    cudaFree(dev_mass);
    cudaFree(dev_force);

    // Free host memory
    free(acc);
    free(mass);
    free(force);

    return 0;
}
```