GENERAL EXECUTION STEPS:-

LOGGING INTO CUDA:-

1) Open PuTTY
2) Enter the IP **“172.16.13.15” [IP may/may not be changed during exam]**
3) Once you are inside the console, type the following credentials
   Username - cse7<your_section(in lowercase)><your_roll_number(01-61)>
   Password - sastra123
   [Note: Credentials may be changed during the exam]
4) After entering the credentials, type the following command
   ssh gnode3

CUDA PROGRAM CREATION, COMPILATION, AND EXECUTION:-

1) To create a program, type the following command
   vi <filename>.cu
2) Once you are inside the file, press **I** to go to INSERT MODE(edit mode)
3) After tying the code, press the following combo sequentially to *save the file*
   Esc(escape) + : (colon) -> then in prompt type *wq*
4) To compile a CUDA program, type the following command
   nvcc <filename>.cu
5) To run the file, use the following command,
   ./a.out

EXP 1 - DOT PRODUCT OF TWO VECTORS

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

__global__ void product(int* a, int* b, int* c, int size) {
    int t = threadIdx.x + blockIdx.x * blockDim.x;
    if (t < size) {
        c[t] = a[t] * b[t];
    }
}

__global__ void sum(int* c, int* partial_sums, int size) {
    extern __shared__ int sdata[];
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < size) {
        sdata[tid] = c[i];
    } else {
        sdata[tid] = 0;
    }
    __syncthreads();

    for (unsigned int s = blockDim.x / 2; s > 0; s >>= 1) {
        if (tid < s) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

    if (tid == 0) {
        partial_sums[blockIdx.x] = sdata[0];
    }
}

int main() {
    int *a, *b, *c, i, N;
    int size;
```

```
    printf("Enter the size of the arrays: ");
    scanf("%d", &N);
    size = N * sizeof(int);

    a = (int*)malloc(size);
    b = (int*)malloc(size);
    c = (int*)malloc(size);
    int ans = 0;

    int *d_a, *d_b, *d_c, *d_partial_sums;
    cudaMalloc((void**)&d_a, size);
    cudaMalloc((void**)&d_b, size);
    cudaMalloc((void**)&d_c, size);
    cudaMalloc((void**)&d_partial_sums, sizeof(int) * ((N + 255) / 256));

    printf("Enter array a:\n");
    for (i = 0; i < N; i++) {
        scanf("%d", &a[i]);
    }
    printf("Enter array b:\n");
    for (i = 0; i < N; i++) {
        scanf("%d", &b[i]);
    }

    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;

    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start);

    product<<<blocksPerGrid, threadsPerBlock>>>(d_a, d_b, d_c, N);

    sum<<<blocksPerGrid, threadsPerBlock, threadsPerBlock * sizeof(int)>>>(d_c,
d_partial_sums, N);
```

```
    int* partial_sums = (int*)malloc(sizeof(int) * blocksPerGrid);
    cudaMemcpy(partial_sums, d_partial_sums, sizeof(int) * blocksPerGrid,
cudaMemcpyDeviceToHost);

    ans = 0;
    for (i = 0; i < blocksPerGrid; i++) {
        ans += partial_sums[i];
    }

    cudaEventRecord(stop);
    cudaEventSynchronize(stop);

    float milliseconds = 0;
    cudaEventElapsedTime(&milliseconds, start, stop);
    cudaEventDestroy(start);
    cudaEventDestroy(stop);

    printf("DOT PRODUCT = %d\n", ans);
    printf("Elapsed time: %f ms\n", milliseconds);

    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
    cudaFree(d_partial_sums);
    free(a);
    free(b);
    free(c);
    free(partial_sums);

    return 0;
}
```

```
[cse7b27@gnode3 ~]$ nvcc ex1.cu
[cse7b27@gnode3 ~]$ ./a.out
Enter the size of the arrays: 5
Enter array a:
1 2 3 4 5
Enter array b:
6 5 8 9 10
DOT PRODUCT = 126
Elapsed time: 0.040032 ms
```

EXP 2 - MATRIX TRANSPOSE USING SHARED MEMORY

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

__global__ void mtrans_smem(int *a, int *b, int rows, int cols)
{
    __shared__ int tile[16][16];

    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int from = y * cols + x;

    if (x < cols && y < rows) {
        tile[threadIdx.y][threadIdx.x] = a[from];
    }

    __syncthreads();

    int bidx = threadIdx.y * blockDim.x + threadIdx.x;
    int irow = bidx / blockDim.y;
    int icol = bidx % blockDim.y;

    x = blockIdx.y * blockDim.y + icol;
    y = blockIdx.x * blockDim.x + irow;
    int to = y * rows + x;

    if (x < rows && y < cols) {
        b[to] = tile[threadIdx.x][threadIdx.y];
    }
```

```
}

int main()
{
    int rows, cols;
    printf("Enter number of rows: ");
    scanf("%d", &rows);
    printf("Enter number of columns: ");
    scanf("%d", &cols);

    int *a, *b;
    int size = rows * cols * sizeof(int);

    a = (int*)malloc(size);
    b = (int*)malloc(size);

    int *d_a, *d_b;
    cudaMalloc((void**)&d_a, size);
    cudaMalloc((void**)&d_b, size);

    printf("Enter the elements of the matrix:\n");
    for(int i = 0; i < rows * cols; i++) {
        scanf("%d", &a[i]);
    }

    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);

    dim3 block(16, 16);
    dim3 grid((cols + block.x - 1) / block.x, (rows + block.y - 1) / block.y);

    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start);

    mtrans_smem<<<grid, block>>>(d_a, d_b, rows, cols);

    cudaEventRecord(stop);
    cudaEventSynchronize(stop);
    float milliseconds = 0;
```

```
    cudaEventElapsedTime(&milliseconds, start, stop);
    cudaEventDestroy(start);
    cudaEventDestroy(stop);

    cudaDeviceSynchronize();
    cudaMemcpy(b, d_b, size, cudaMemcpyDeviceToHost);

    printf("Transposed matrix:\n");
    for(int i = 0; i < cols; i++) {
        for(int j = 0; j < rows; j++) {
            printf("\t%d", b[i * rows + j]);
        }
        printf("\n");
    }
    printf("Elapsed time : %f ms \n",milliseconds);
    cudaFree(d_a);
    cudaFree(d_b);
    free(a);
    free(b);

    cudaDeviceReset();
    return 0;
}
```

```
[cse7b27@gnode3 ~]$ nvcc ex2.cu
[cse7b27@gnode3 ~]$ ./a.out
Enter number of rows: 3
Enter number of columns: 2
Enter the elements of the matrix:
1 2
3 4
5 6
Transposed matrix:
        1       3       5
        2       4       6
Elapsed time : 0.006560 ms
```

EXP 3 - 1D STENCIL USING CONSTANT MEMORY

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

```
__constant__ int d_radius;

__global__ void stencilKernel(int *input, int *output, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < n) {
        int sum = 0;
        for (int offset = -d_radius; offset <= d_radius; ++offset) {
            int neighborIdx = idx + offset;
            if (neighborIdx >= 0 && neighborIdx < n) {
                sum += input[neighborIdx];
            }
        }
        output[idx] = sum;
    }
}

int main() {
    int n, radius;

    printf("Enter the size of the array: ");
    scanf("%d", &n);

    int *h_input = (int*)malloc(n * sizeof(int));
    int *h_output = (int*)malloc(n * sizeof(int));

    printf("Enter the elements of the array:\n");
    for (int i = 0; i < n; ++i) {
        scanf("%d", &h_input[i]);
    }

    printf("Enter the radius: ");
    scanf("%d", &radius);

    int *d_input, *d_output;
    cudaMalloc((void**)&d_input, n * sizeof(int));
    cudaMalloc((void**)&d_output, n * sizeof(int));

    cudaMemcpy(d_input, h_input, n * sizeof(int), cudaMemcpyHostToDevice);
```

```
    cudaMemcpyToSymbol(d_radius, &radius, sizeof(int));

    int threadsPerBlock = 256;
    int blocksPerGrid = (n + threadsPerBlock - 1) / threadsPerBlock;

    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start);

    stencilKernel<<<blocksPerGrid, threadsPerBlock>>>(d_input, d_output, n);

    cudaEventRecord(stop);
    cudaEventSynchronize(stop);
    float milliseconds = 0;
    cudaEventElapsedTime(&milliseconds, start, stop);
    cudaEventDestroy(start);
    cudaEventDestroy(stop);

    cudaMemcpy(h_output, d_output, n * sizeof(int), cudaMemcpyDeviceToHost);

    printf("Result array:\n");
    for (int i = 0; i < n; ++i) {
        printf("%d ", h_output[i]);
    }
    printf("\n");

    printf("Elapsed time : %f ms",milliseconds);

    cudaFree(d_input);
    cudaFree(d_output);
    free(h_input);
    free(h_output);

    return 0;
}
```

```
[cse7b27@gnode3 ~]$ nvcc ex3.cu
[cse7b27@gnode3 ~]$ ./a.out
Enter the size of the array: 6
Enter the elements of the array:
1 2 3 4 5 6
Enter the radius: 3
Result array:
10 15 21 21 20 18
Elapsed time : 0.039776 ms[cse7b27@gnode3 ~]$
```

EXP 4 - PRE-ORDER TREE TRAVERSAL

```
#include <iostream>
#include <cuda_runtime.h>

#define N 1024 // Maximum number of nodes
#define THREADS_PER_BLOCK 32

__global__ void preorder_traversal_kernel(int *tree, int *output, int *output_idx, int
num_nodes) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < num_nodes && tree[idx] != -1) {
        int stack[N];
        int top = -1;

        // Initialize stack with root node if it's the first thread in the block
        if (threadIdx.x == 0) {
            stack[++top] = idx;
        }

        while (top >= 0) {
            int current_node = stack[top--];
            int output_position = atomicAdd(output_idx, 1);

            // Output the current node
            output[output_position] = tree[current_node];

            // Push right and then left child to stack
            int right_child = 2 * current_node + 2;
```

```
            int left_child = 2 * current_node + 1;

            if (right_child < num_nodes && tree[right_child] != -1) {
                stack[++top] = right_child;
            }
            if (left_child < num_nodes && tree[left_child] != -1) {
                stack[++top] = left_child;
            }
        }
    }
}

void preorder_traversal_cuda(int *tree, int *output, int num_nodes) {
    int *d_tree, *d_output, *d_output_idx;

    // Allocate memory on GPU
    cudaMalloc((void**)&d_tree, num_nodes * sizeof(int));
    cudaMalloc((void**)&d_output, num_nodes * sizeof(int));
    cudaMalloc((void**)&d_output_idx, sizeof(int));

    // Copy tree to GPU
    cudaMemcpy(d_tree, tree, num_nodes * sizeof(int), cudaMemcpyHostToDevice);

    // Initialize output array on GPU
    cudaMemset(d_output, -1, num_nodes * sizeof(int));

    // Initialize output index on GPU
    int initial_idx = 0;
    cudaMemcpy(d_output_idx, &initial_idx, sizeof(int), cudaMemcpyHostToDevice);

    // Start timing
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start, 0);

    // Launch kernel with multiple threads
    int blocks = (num_nodes + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;
    preorder_traversal_kernel<<<blocks, THREADS_PER_BLOCK>>>(d_tree, d_output,
d_output_idx, num_nodes);
```

```
    // Stop timing
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);

    float elapsedTime;
    cudaEventElapsedTime(&elapsedTime, start, stop);

    // Copy the result back to the host
    cudaMemcpy(output, d_output, num_nodes * sizeof(int), cudaMemcpyDeviceToHost);

    // Free GPU memory
    cudaFree(d_tree);
    cudaFree(d_output);
    cudaFree(d_output_idx);

    // Destroy CUDA events
    cudaEventDestroy(start);
    cudaEventDestroy(stop);

    // Output elapsed time
    std::cout << "Time for the kernel execution: " << elapsedTime << " ms" << std::endl;
}

int main() {
    int num_nodes;

    std::cout << "Enter the number of nodes in the binary tree: ";
    std::cin >> num_nodes;

    int *tree = new int[num_nodes];
    std::cout << "Enter the elements of the binary tree in level-order (use -1 for null
nodes):" << std::endl;
    for (int i = 0; i < num_nodes; i++) {
        std::cin >> tree[i];
    }

    int *output = new int[num_nodes];

    preorder_traversal_cuda(tree, output, num_nodes);
```

```
    std::cout << "Preorder Traversal: ";
    for (int i = 0; i < num_nodes; i++) {
        if (output[i] != -1)
            std::cout << output[i] << " ";
    }
    std::cout << std::endl;

    delete[] tree;
    delete[] output;

    return 0;
}
```

```
[cse7b27@gnode3 ~]$ nvcc ex4.cu
[cse7b27@gnode3 ~]$ ./a.out
Enter the number of nodes in the binary tree: 7
Enter the elements of the binary tree in level-order (use -1 for null nodes):
1 2 3 4 5 6 7
Time for the kernel execution: 0.215296 ms
Preorder Traversal: 1 2 4 5 3 6 7
```

Sample IO format :-

- Input:
  - `num_nodes = 7`
  - `tree = {1, 2, 3, 4, 5, 6, 7}`
  - This represents the following tree:

markdown   Copy code

```
    1
   / \
  2   3
 / \ / \
4  5 6  7
```

- Expected Output: `1 2 4 5 3 6 7`

EXP 5 - ODD-EVEN TRANSPOSITION SORT

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

__device__ int custom_max(int a, int b) {
    return a > b ? a : b;
}

__device__ int custom_min(int a, int b) {
    return a < b ? a : b;
}

__global__ void sort(int* a, int n) {
    int tid = threadIdx.x;

    for (int i = 0; i < n; i++) {

        if ((tid % 2 == 1) && (tid < n - 1)) {
            int t = a[tid + 1];
            a[tid + 1] = custom_max(a[tid], t);
            a[tid] = custom_min(a[tid], t);
        }
        __syncthreads();


        if ((tid % 2 == 0) && (tid < n - 1)) {
            int t = a[tid + 1];
            a[tid + 1] = custom_max(a[tid], t);
            a[tid] = custom_min(a[tid], t);
        }
        __syncthreads();
    }
}

int main() {
    int n;

    printf("Enter the size of the array: ");
    scanf("%d", &n);

    int *a = (int *)malloc(n * sizeof(int));
```

```
printf("Enter %d elements: ", n);
for (int i = 0; i < n; i++) {
    scanf("%d", &a[i]);
}

printf("ARRAY ELEMENTS BEFORE Sorting:");
for (int i = 0; i < n; i++) {
    printf(" %d", a[i]);
}
printf("\n");

int* d_a;
cudaMalloc((void**)&d_a, n * sizeof(int));
cudaMemcpy(d_a, a, n * sizeof(int), cudaMemcpyHostToDevice);

cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start);

sort<<<1, n>>>(d_a, n);
cudaDeviceSynchronize();

cudaEventRecord(stop);
cudaEventSynchronize(stop);
float milliseconds = 0;
cudaEventElapsedTime(&milliseconds, start, stop);
cudaEventDestroy(start);
cudaEventDestroy(stop);

cudaMemcpy(a, d_a, n * sizeof(int), cudaMemcpyDeviceToHost);

printf("ARRAY ELEMENTS AFTER Sorting:");
for (int i = 0; i < n; i++) {
    printf(" %d", a[i]);
}
printf("\n");

printf("Elapsed time : %f ms",milliseconds);
```

```
    cudaFree(d_a);
    free(a);
    return 0;
}
```

```
[cse7b27@gnode3 ~]$ nvcc ex5.cu
[cse7b27@gnode3 ~]$ ./a.out
Enter the size of the array: 8
Enter 8 elements: 1 2 3 4 5 6 7 8
ARRAY ELEMENTS BEFORE Sorting: 1 2 3 4 5 6 7 8
ARRAY ELEMENTS AFTER Sorting: 1 2 3 4 5 6 7 8
Elapsed time : 0.042688 ms[cse7b27@gnode3 ~]$
```

EXP 6 - QUICK SORT IN PARALLEL

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

__device__ void swap(int* a, int i, int j) {
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}

__device__ int partition(int* a, int left, int right) {
    int pivot = a[right];
    int i = left - 1;
    for (int j = left; j < right; j++) {
        if (a[j] < pivot) {
            i++;
            swap(a, i, j);
        }
    }
    swap(a, i + 1, right);
    return i + 1;
}

__global__ void quicksort(int* a, int* stack, int n) {
    int left, right, top, pivot;
```

```
    // Initialize stack
    if (threadIdx.x == 0) {
        stack[0] = 0;
        stack[1] = n - 1;
    }
    __syncthreads();

    top = 1;

    while (top >= 0) {
        if (threadIdx.x == 0) {
            right = stack[top--];
            left = stack[top--];
        }
        __syncthreads();

        if (left < right) {
            if (threadIdx.x == 0) {
                pivot = partition(a, left, right);
                stack[++top] = left;
                stack[++top] = pivot - 1;
                stack[++top] = pivot + 1;
                stack[++top] = right;
            }
        }
        __syncthreads();
    }
}

int main() {
    int n;
    printf("Enter the size of the array: ");
    scanf("%d", &n);

    int *h_a = (int*)malloc(n * sizeof(int));
    printf("Enter elements of the array:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &h_a[i]);
    }
```

```
    int *d_a, *d_stack;
    cudaMalloc((void**)&d_a, n * sizeof(int));
    cudaMalloc((void**)&d_stack, 2 * n * sizeof(int)); // Stack size for each thread

    cudaMemcpy(d_a, h_a, n * sizeof(int), cudaMemcpyHostToDevice);

    dim3 threadsPerBlock(1);
    dim3 blocksPerGrid(1);

    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    cudaEventRecord(start);
    quicksort<<<blocksPerGrid, threadsPerBlock>>>(d_a, d_stack, n);
    cudaEventRecord(stop);

    cudaMemcpy(h_a, d_a, n * sizeof(int), cudaMemcpyDeviceToHost);

    cudaEventSynchronize(stop);
    float milliseconds = 0;
    cudaEventElapsedTime(&milliseconds, start, stop);

    printf("Sorted array:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", h_a[i]);
    }
    printf("\nElapsed time: %f ms\n", milliseconds);

    cudaFree(d_a);
    cudaFree(d_stack);
    free(h_a);

    return 0;
}
```

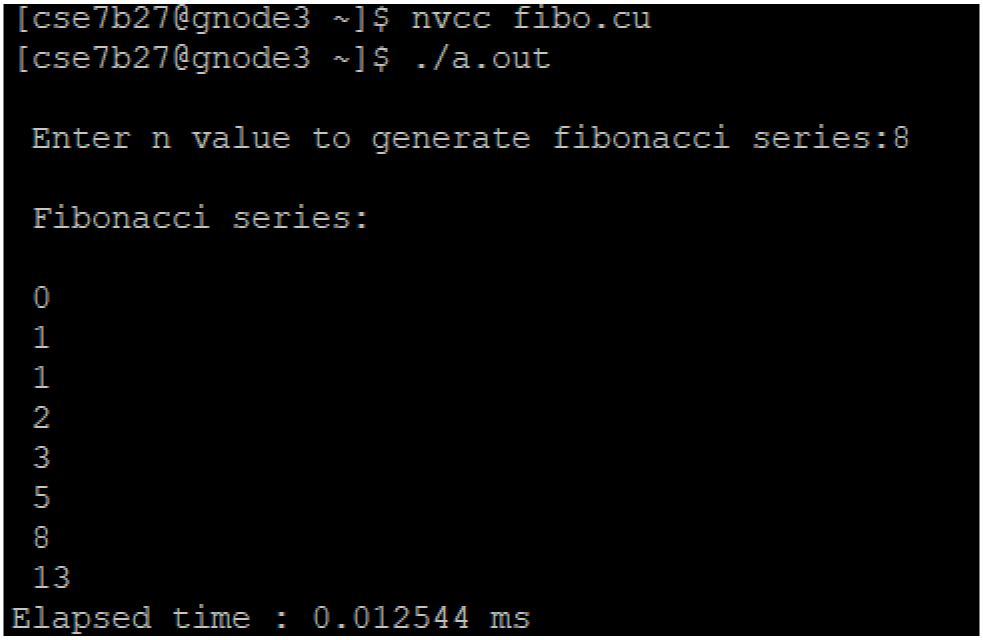
```
[cse7b27@gnode3 ~]$ nvcc quicksort.cu
[cse7b27@gnode3 ~]$ ./a.out
Enter the size of the array: 6
Enter elements of the array:
8 1 9 7 2 3
Sorted array:
1 2 3 7 8 9
Elapsed time: 0.042432 ms
```

ADDITIONAL EXPERIMENTS:-

PRINTING FIBONACCI SERIES IN PARALLEL(USING BINET'S FORMULA)

```
#include<stdio.h>
#include<cuda.h>
#define N 200

__global__ void fibonacci(int *a)
{
   int i = threadIdx.x;
   a[0] = 0;
   if(i>0)
   {
     a[i] = round((powf((1+sqrtf(5))/2,i) - powf((1-sqrtf(5))/2,i))/sqrtf(5));
   }
}

int main()
{
   int n,a[N];
   int *d_n,*d_a;

   cudaMalloc((void**)&d_n,sizeof(int));
   cudaMalloc((void**)&d_a,N*sizeof(int));

   printf("\n Enter n value to generate fibonacci series:");
   scanf("%d",&n);

   cudaMemcpy(d_n,&n,sizeof(int),cudaMemcpyHostToDevice);
```

```
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start);

    fibonacci<<<1,n>>>(d_a);

    cudaEventRecord(stop);
    cudaEventSynchronize(stop);

    float milliseconds = 0;
    cudaEventElapsedTime(&milliseconds, start, stop);
    cudaEventDestroy(start);
    cudaEventDestroy(stop);

    cudaMemcpy(a,d_a,N*sizeof(int),cudaMemcpyDeviceToHost);

    printf("\n Fibonacci series: \n");
    for(int i=0;i<n;i++)
    {
      printf("\n %d",a[i]);
    }
    printf("\n");
    printf("Elapsed time : %f ms\n",milliseconds);
    return 0;
}
```



FINDING PRIME NUMBERS WITHIN A RANGE

```
#include <stdio.h>
#include <cuda.h>
#include <math.h>

__global__ void findprime(int *a, int n) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < 2)
      return;
    if (tid <= n) {
        for (int i = 2; i <= sqrtf((float)tid); i++) {
            if (tid % i == 0) {
                a[tid] = 0;
                return;
            }
        }
        a[tid] = 1;
    }
}

int main() {
    int n;
    int *a, *d_a;
```

```
printf("\nFind prime numbers from 1 to: ");
scanf("%d", &n);

a = (int*)malloc((n + 1) * sizeof(int));
cudaMalloc((void**)&d_a, (n + 1) * sizeof(int));

for (int i = 0; i <= n; i++) {
    a[i] = 1;
}

cudaMemcpy(d_a, a, (n + 1) * sizeof(int), cudaMemcpyHostToDevice);

int threadsPerBlock = 256;
int blocksPerGrid = (n + threadsPerBlock) / threadsPerBlock;

cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start);

findprime<<<blocksPerGrid, threadsPerBlock>>>(d_a, n);

cudaEventRecord(stop);
cudaEventSynchronize(stop);
float milliseconds = 0;
cudaEventElapsedTime(&milliseconds, start, stop);
cudaEventDestroy(start);
cudaEventDestroy(stop);

cudaMemcpy(a, d_a, (n + 1) * sizeof(int), cudaMemcpyDeviceToHost);

printf("\nPrime numbers are: ");
for (int i = 2; i <= n; i++) {
    if (a[i] == 1) {
        printf("\n %d", i);
    }
}
printf("\n");
printf("Elapsed time : %f ms\n", milliseconds);
```

```
    cudaFree(d_a);
    free(a);

    return 0;
}
```

```
[cse7b27@gnode3 ~]$ nvcc prime_no.cu
[cse7b27@gnode3 ~]$ ./a.out

Find prime numbers from 1 to: 100

Prime numbers are:
 2
 3
 5
 7
 11
 13
 17
 19
 23
 29
 31
 37
 41
 43
 47
 53
 59
 61
 67
 71
 73
 79
 83
 89
 97
Elapsed time : 0.034976 ms
```

VECTOR ADDITION IN PARALLEL

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

__global__ void vectorAdd(int *a, int *b, int *c, int n) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < n) {
        c[idx] = a[idx] + b[idx];
    }
}

int main() {
    int n;
    printf("Enter the size of the vectors: ");
    scanf("%d", &n);

    int size = n * sizeof(int);

    int *h_a = (int*)malloc(size);
    int *h_b = (int*)malloc(size);
    int *h_c = (int*)malloc(size);

    printf("Enter elements of vector A:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &h_a[i]);
    }

    printf("Enter elements of vector B:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &h_b[i]);
    }

    int *d_a, *d_b, *d_c;
    cudaMalloc((void**)&d_a, size);
    cudaMalloc((void**)&d_b, size);
    cudaMalloc((void**)&d_c, size);

    cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);

    int threadsPerBlock = 256;
```

```
    int blocksPerGrid = (n + threadsPerBlock - 1) / threadsPerBlock;

    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    cudaEventRecord(start);
    vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_a, d_b, d_c, n);
    cudaEventRecord(stop);

    cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost);

    cudaEventSynchronize(stop);
    float milliseconds = 0;
    cudaEventElapsedTime(&milliseconds, start, stop);

    printf("Resultant vector:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", h_c[i]);
    }
    printf("\nElapsed time: %f ms\n", milliseconds);

    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
    free(h_a);
    free(h_b);
    free(h_c);

    return 0;
}
```

```
[cse7b27@gnode3 ~]$ nvcc vectoradd.cu
[cse7b27@gnode3 ~]$ ./a.out
Enter the size of the vectors: 6
Enter elements of vector A:
1 2 0 6 5 2
Enter elements of vector B:
6 8 5 1 0 5
Resultant vector:
7 10 5 7 5 7
Elapsed time: 0.036800 ms
```

MATRIX MULTIPLICATION IN PARALLEL

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

// Kernel for matrix multiplication
__global__ void matrixMul(int *a, int *b, int *c, int N, int M, int P) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    if (row < N && col < P) {
        int sum = 0;
        for (int i = 0; i < M; i++) {
            sum += a[row * M + i] * b[i * P + col];
        }
        c[row * P + col] = sum;
    }
}

int main() {
    int N, M, P;
    printf("Enter the number of rows and columns for matrix A (rows cols): ");
    scanf("%d %d", &N, &M);
    printf("Enter the number of rows and columns for matrix B (rows cols): ");
    scanf("%d %d", &M, &P);

    // Check if the matrices can be multiplied
    if (M != M) {
        printf("Error: Number of columns in matrix A must be equal to number of rows in matrix B.\n");
        return -1;
```

```
}

int sizeA = N * M * sizeof(int);
int sizeB = M * P * sizeof(int);
int sizeC = N * P * sizeof(int);

int *h_a = (int*)malloc(sizeA);
int *h_b = (int*)malloc(sizeB);
int *h_c = (int*)malloc(sizeC);

printf("Enter elements of matrix A:\n");
for (int i = 0; i < N * M; i++) {
    scanf("%d", &h_a[i]);
}

printf("Enter elements of matrix B:\n");
for (int i = 0; i < M * P; i++) {
    scanf("%d", &h_b[i]);
}

int *d_a, *d_b, *d_c;
cudaMalloc((void**)&d_a, sizeA);
cudaMalloc((void**)&d_b, sizeB);
cudaMalloc((void**)&d_c, sizeC);

cudaMemcpy(d_a, h_a, sizeA, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, h_b, sizeB, cudaMemcpyHostToDevice);

dim3 threadsPerBlock(16, 16);
dim3 blocksPerGrid((P + threadsPerBlock.x - 1) / threadsPerBlock.x,
                   (N + threadsPerBlock.y - 1) / threadsPerBlock.y);

cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start);
matrixMul<<<blocksPerGrid, threadsPerBlock>>>(d_a, d_b, d_c, N, M, P);
cudaEventRecord(stop);
```

```
    cudaMemcpy(h_c, d_c, sizeC, cudaMemcpyDeviceToHost);

    cudaEventSynchronize(stop);
    float milliseconds = 0;
    cudaEventElapsedTime(&milliseconds, start, stop);

    printf("Resultant matrix C:\n");
    for (int i = 0; i < N * P; i++) {
        if (i % P == 0 && i != 0) printf("\n");
        printf("%d ", h_c[i]);
    }
    printf("\nElapsed time: %f ms\n", milliseconds);

    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
    free(h_a);
    free(h_b);
    free(h_c);

    return 0;
}
```

```
[cse7b27@gnode3 ~]$ nvcc matrixmul.cu
[cse7b27@gnode3 ~]$ ./a.out
Enter the number of rows and columns for matrix A (rows cols): 1 3
Enter the number of rows and columns for matrix B (rows cols): 3 1
Enter elements of matrix A:
1 2 3
Enter elements of matrix B:
1
2
3
Resultant matrix C:
14
Elapsed time: 0.038272 ms
```

SUM OF AN ARRAY USING PARALLEL REDUCTION

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

```
__global__ void reduceSum(int *input, int *output, int n) {
    extern __shared__ int sdata[];
    int tid = threadIdx.x;
    int i = blockIdx.x * blockDim.x + tid;
    sdata[tid] = (i < n) ? input[i] : 0;
    __syncthreads();

    for (int s = blockDim.x / 2; s > 0; s >>= 1) {
        if (tid < s) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

    if (tid == 0) {
        output[blockIdx.x] = sdata[0];
    }
}

int main() {
    int n;
    printf("Enter the size of the array: ");
    scanf("%d", &n);

    int size = n * sizeof(int);
    int *h_input = (int*)malloc(size);
    int *h_output = (int*)malloc(size);

    printf("Enter elements of the array:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &h_input[i]);
    }

    int *d_input, *d_output;
    cudaMalloc((void**)&d_input, size);
    cudaMalloc((void**)&d_output, size);

    cudaMemcpy(d_input, h_input, size, cudaMemcpyHostToDevice);

    int threadsPerBlock = 256;
```

```
    int blocksPerGrid = (n + threadsPerBlock - 1) / threadsPerBlock;

    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    cudaEventRecord(start);
    reduceSum<<<blocksPerGrid, threadsPerBlock, threadsPerBlock *
sizeof(int)>>>(d_input, d_output, n);
    cudaEventRecord(stop);

    cudaMemcpy(h_output, d_output, blocksPerGrid * sizeof(int),
cudaMemcpyDeviceToHost);

    int sum = 0;
    for (int i = 0; i < blocksPerGrid; i++) {
        sum += h_output[i];
    }

    cudaEventSynchronize(stop);
    float milliseconds = 0;
    cudaEventElapsedTime(&milliseconds, start, stop);

    printf("Sum of array elements: %d\n", sum);
    printf("Elapsed time: %f ms\n", milliseconds);

    cudaFree(d_input);
    cudaFree(d_output);
    free(h_input);
    free(h_output);

    return 0;
}
```

```
[cse7b27@gnode3 ~]$ nvcc sum_array.cu
[cse7b27@gnode3 ~]$ ./a.out
Enter the size of the array: 8
Enter elements of the array:
1 2 3 4 5 6 7 8
Sum of array elements: 36
Elapsed time: 0.037376 ms
```

PARALLEL LINEAR SEARCH

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

__global__ void parallelLinearSearch(int *arr, int *result, int target, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < n) {
        if (arr[idx] == target) {
            result[idx] = idx;
        } else {
            result[idx] = n;
        }
    }
}

int main() {
    int n, target;
    printf("Enter the size of the array: ");
    scanf("%d", &n);

    int *h_arr = (int*)malloc(n * sizeof(int));
    printf("Enter elements of the array:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &h_arr[i]);
    }

    printf("Enter the target value to search: ");
    scanf("%d", &target);

    int *d_arr, *d_result;
    int *h_result = (int*)malloc(n * sizeof(int));

    cudaMalloc((void**)&d_arr, n * sizeof(int));
    cudaMalloc((void**)&d_result, n * sizeof(int));
    cudaMemcpy(d_arr, h_arr, n * sizeof(int), cudaMemcpyHostToDevice);

    int blockSize = 256;
```

```
    int numBlocks = (n + blockSize - 1) / blockSize;

    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    cudaEventRecord(start);

    parallelLinearSearch<<<numBlocks, blockSize>>>(d_arr, d_result, target, n);

    cudaEventRecord(stop);
    cudaMemcpy(h_result, d_result, n * sizeof(int), cudaMemcpyDeviceToHost);

    cudaEventSynchronize(stop);
    float milliseconds = 0;
    cudaEventElapsedTime(&milliseconds, start, stop);

    int foundIndex = n;
    for (int i = 0; i < n; i++) {
        if (h_result[i] < foundIndex) {
            foundIndex = h_result[i];
        }
    }

    if (foundIndex < n) {
        printf("Element found at index %d\n", foundIndex);
    } else {
        printf("Element not found\n");
    }
    printf("Elapsed time: %f ms\n", milliseconds);

    cudaFree(d_arr);
    cudaFree(d_result);
    free(h_arr);
    free(h_result);

    return 0;
}
```

```
[cse7b27@gnode3 ~]$ nvcc linearsearch.cu
[cse7b27@gnode3 ~]$ ./a.out
Enter the size of the array: 5
Enter elements of the array:
1 0 5 6 10
Enter the target value to search: 6
Element found at index 3
Elapsed time: 0.032736 ms
```

MATRIX ADDITION

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

__global__ void matrixAddKernel(float *A, float *B, float *C, int numRows, int numCols) {
   int row = blockIdx.y * blockDim.y + threadIdx.y;
   int col = blockIdx.x * blockDim.x + threadIdx.x;

   if (row < numRows && col < numCols) {
      C[row * numCols + col] = A[row * numCols + col] + B[row * numCols + col];
   }
}

int main() {
   int numRows, numCols;

   printf("Enter the number of rows and columns for matrices: ");
   scanf("%d %d", &numRows, &numCols);

   size_t size = numRows * numCols * sizeof(float);

   float *h_A = (float*)malloc(size);
   float *h_B = (float*)malloc(size);
   float *h_C = (float*)malloc(size);

   printf("Enter the elements of matrix A:\n");
   for (int i = 0; i < numRows * numCols; ++i) {
      scanf("%f", &h_A[i]);
   }
```

```
    printf("Enter the elements of matrix B:\n");
    for (int i = 0; i < numRows * numCols; ++i) {
        scanf("%f", &h_B[i]);
    }

    float *d_A, *d_B, *d_C;
    cudaMalloc((void**)&d_A, size);
    cudaMalloc((void**)&d_B, size);
    cudaMalloc((void**)&d_C, size);

    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    dim3 dimBlock(16, 16);
    dim3 dimGrid((numCols + dimBlock.x - 1) / dimBlock.x, (numRows + dimBlock.y - 1) /
dimBlock.y);

    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start);

    matrixAddKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C, numRows, numCols);

    cudaEventRecord(stop);
    cudaEventSynchronize(stop);
    float milliseconds = 0;
    cudaEventElapsedTime(&milliseconds, start, stop);
    cudaEventDestroy(start);
    cudaEventDestroy(stop);

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    printf("Result matrix C:\n");
    for (int i = 0; i < numRows; i++) {
        for (int j = 0; j < numCols; j++) {
            printf("%f ", h_C[i * numCols + j]);
        }
        printf("\n");
    }
```

```
    printf("\n");
    printf("Elapsed time : %f ms\n",milliseconds);

    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);


    free(h_A);
    free(h_B);
    free(h_C);

    return 0;
}
```

```
[cse7b27@gnode3 ~]$ nvcc matrixadd.cu
[cse7b27@gnode3 ~]$ ./a.out
Enter the number of rows and columns for matrices: 3 4
Enter the elements of matrix A:
1 2 3 4
0 2 1 5
5 6 7 8
Enter the elements of matrix B:
0 0 1 2
1 2 10 15
9 5 8 6
Result matrix C:
1.000000 2.000000 4.000000 6.000000
1.000000 4.000000 11.000000 20.000000
14.000000 11.000000 15.000000 14.000000

Elapsed time : 0.036704 ms
```