Lab-6 Programs in CUDA

Q1 – 1D Convolution

```
// Title  : 1D Convolution using CUDA
// Author : Aditya Sinha
// Date   : 20/02/2026

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

__global__ void convolution1D(const int *input, const int *mask, int
*output, int n, int m) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n) {
        int radius = m / 2;
        int sum = 0;
        for (int j = 0; j < m; ++j) {
            int inputIndex = idx + j - radius;
            if (inputIndex >= 0 && inputIndex < n) {
                sum += input[inputIndex] * mask[j];
            }
        }
        output[idx] = sum;
    }
}

static void convolution1DHost(const int *input, const int *mask, int
*output, int n, int m) {
    int radius = m / 2;
    for (int idx = 0; idx < n; ++idx) {
        int sum = 0;
        for (int j = 0; j < m; ++j) {
            int inputIndex = idx + j - radius;
            if (inputIndex >= 0 && inputIndex < n) {
                sum += input[inputIndex] * mask[j];
            }
        }
        output[idx] = sum;
    }
}

static int convolution1DCuda(const int *h_input, const int *h_mask, int
*h_output, int n, int m) {
    int inputBytes = n * (int)sizeof(int);
    int maskBytes = m * (int)sizeof(int);

    int *d_input = NULL;
    int *d_mask = NULL;
    int *d_output = NULL;
```

```
    cudaError_t err = cudaMalloc((void **)&d_input, inputBytes);
    if (err != cudaSuccess) {
        return 1;
    }
    err = cudaMalloc((void **)&d_mask, maskBytes);
    if (err != cudaSuccess) {
        cudaFree(d_input);
        return 1;
    }
    err = cudaMalloc((void **)&d_output, inputBytes);
    if (err != cudaSuccess) {
        cudaFree(d_input);
        cudaFree(d_mask);
        return 1;
    }

    err = cudaMemcpy(d_input, h_input, inputBytes, cudaMemcpyHostToDevice);
    if (err != cudaSuccess) {
        cudaFree(d_input);
        cudaFree(d_mask);
        cudaFree(d_output);
        return 1;
    }
    err = cudaMemcpy(d_mask, h_mask, maskBytes, cudaMemcpyHostToDevice);
    if (err != cudaSuccess) {
        cudaFree(d_input);
        cudaFree(d_mask);
        cudaFree(d_output);
        return 1;
    }

    int threads = 256;
    int blocks = (n + threads - 1) / threads;

    convolution1D<<<blocks, threads>>>(d_input, d_mask, d_output, n, m);
    err = cudaGetLastError();
    if (err != cudaSuccess) {
        cudaFree(d_input);
        cudaFree(d_mask);
        cudaFree(d_output);
        return 1;
    }
    err = cudaDeviceSynchronize();
    if (err != cudaSuccess) {
        cudaFree(d_input);
        cudaFree(d_mask);
        cudaFree(d_output);
        return 1;
    }

    err = cudaMemcpy(h_output, d_output, inputBytes,
cudaMemcpyDeviceToHost);
    if (err != cudaSuccess) {
```

```
        cudaFree(d_input);
        cudaFree(d_mask);
        cudaFree(d_output);
        return 1;
    }

    cudaFree(d_input);
    cudaFree(d_mask);
    cudaFree(d_output);
    return 0;
}

int main() {
    int n = 0;
    int m = 0;
    scanf("%d", &n);
    scanf("%d", &m);

    if (n <= 0 || m <= 0) {
        return 1;
    }

    int inputBytes = n * (int)sizeof(int);
    int maskBytes = m * (int)sizeof(int);

    int *h_input = (int *)malloc(inputBytes);
    int *h_mask = (int *)malloc(maskBytes);
    int *h_output = (int *)malloc(inputBytes);

    if (h_input == NULL || h_mask == NULL || h_output == NULL) {
        free(h_input);
        free(h_mask);
        free(h_output);
        return 1;
    }

    for (int i = 0; i < n; ++i) {
        scanf("%d", &h_input[i]);
    }
    for (int i = 0; i < m; ++i) {
        scanf("%d", &h_mask[i]);
    }

    if (convolution1DCuda(h_input, h_mask, h_output, n, m) != 0) {
        convolution1DHost(h_input, h_mask, h_output, n, m);
    }

    for (int i = 0; i < n; ++i) {
        printf("%d", h_output[i]);
        if (i + 1 < n) {
            printf(" ");
        }
    }
    printf("\n");
```

```
    free(h_input);
    free(h_mask);
    free(h_output);

    return 0;
}
```

Output :

```
sinhaa@MIT-ICT-LAB5-27:~/Desktop/College/Sem6/PCAP/Lab/Lab6$ ./q1
3
4
1 2 3
3 4 65 7
79 155 206
```

Q2 – Parallel Selection Sort

```
// Title  : Parallel Selection Sort using CUDA
// Author : Aditya Sinha
// Date   : 20/02/2026

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

__global__ void findBlockMin(const int *arr, int start, int n, int
*blockMinVal, int *blockMinIdx) {
    extern __shared__ int shared[];
    int *vals = shared;
    int *idxs = shared + blockDim.x;

    int tid = threadIdx.x;
    int global = start + blockIdx.x * blockDim.x + tid;

    if (global < n) {
        vals[tid] = arr[global];
        idxs[tid] = global;
    } else {
        vals[tid] = INT_MAX;
        idxs[tid] = -1;
    }
    __syncthreads();

    for (int stride = blockDim.x / 2; stride > 0; stride >>= 1) {
        if (tid < stride) {
            int leftVal = vals[tid];
```

```
            int rightVal = vals[tid + stride];
            int leftIdx = idxs[tid];
            int rightIdx = idxs[tid + stride];

            if (rightVal < leftVal || (rightVal == leftVal && rightIdx >= 0
&& (leftIdx < 0 || rightIdx < leftIdx))) {
                vals[tid] = rightVal;
                idxs[tid] = rightIdx;
            }
        }
        __syncthreads();
    }

    if (tid == 0) {
        blockMinVal[blockIdx.x] = vals[0];
        blockMinIdx[blockIdx.x] = idxs[0];
    }
}

__global__ void swapElements(int *arr, int i, int j) {
    if (threadIdx.x == 0 && blockIdx.x == 0 && i != j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }
}

static void selectionSortHost(int *arr, int n) {
    for (int i = 0; i < n - 1; ++i) {
        int minIdx = i;
        for (int j = i + 1; j < n; ++j) {
            if (arr[j] < arr[minIdx]) {
                minIdx = j;
            }
        }
        if (minIdx != i) {
            int temp = arr[i];
            arr[i] = arr[minIdx];
            arr[minIdx] = temp;
        }
    }
}

static int selectionSortCuda(int *h_arr, int n) {
    int bytes = n * (int)sizeof(int);
    int *d_arr = NULL;
    cudaError_t err = cudaMalloc((void **)&d_arr, bytes);
    if (err != cudaSuccess) {
        return 1;
    }
    err = cudaMemcpy(d_arr, h_arr, bytes, cudaMemcpyHostToDevice);
    if (err != cudaSuccess) {
        cudaFree(d_arr);
        return 1;
```

```
    }

    int threads = 256;

    for (int start = 0; start < n - 1; ++start) {
        int remaining = n - start;
        int blocks = (remaining + threads - 1) / threads;

        int *d_blockMinVal = NULL;
        int *d_blockMinIdx = NULL;
        err = cudaMalloc((void **)&d_blockMinVal, blocks *
(int)sizeof(int));
        if (err != cudaSuccess) {
            cudaFree(d_arr);
            return 1;
        }
        err = cudaMalloc((void **)&d_blockMinIdx, blocks *
(int)sizeof(int));
        if (err != cudaSuccess) {
            cudaFree(d_blockMinVal);
            cudaFree(d_arr);
            return 1;
        }

        int sharedBytes = 2 * threads * (int)sizeof(int);
        findBlockMin<<<blocks, threads, sharedBytes>>>(d_arr, start, n,
d_blockMinVal, d_blockMinIdx);
        err = cudaGetLastError();
        if (err != cudaSuccess) {
            cudaFree(d_blockMinVal);
            cudaFree(d_blockMinIdx);
            cudaFree(d_arr);
            return 1;
        }

        int *h_blockMinVal = (int *)malloc(blocks * (int)sizeof(int));
        int *h_blockMinIdx = (int *)malloc(blocks * (int)sizeof(int));
        if (h_blockMinVal == NULL || h_blockMinIdx == NULL) {
            free(h_blockMinVal);
            free(h_blockMinIdx);
            cudaFree(d_blockMinVal);
            cudaFree(d_blockMinIdx);
            cudaFree(d_arr);
            return 1;
        }

        err = cudaMemcpy(h_blockMinVal, d_blockMinVal, blocks *
(int)sizeof(int), cudaMemcpyDeviceToHost);
        if (err != cudaSuccess) {
            free(h_blockMinVal);
            free(h_blockMinIdx);
            cudaFree(d_blockMinVal);
            cudaFree(d_blockMinIdx);
            cudaFree(d_arr);
```

```
            return 1;
        }
        err = cudaMemcpy(h_blockMinIdx, d_blockMinIdx, blocks *
(int)sizeof(int), cudaMemcpyDeviceToHost);
        if (err != cudaSuccess) {
            free(h_blockMinVal);
            free(h_blockMinIdx);
            cudaFree(d_blockMinVal);
            cudaFree(d_blockMinIdx);
            cudaFree(d_arr);
            return 1;
        }

        int minVal = INT_MAX;
        int minIdx = start;
        for (int b = 0; b < blocks; ++b) {
            int val = h_blockMinVal[b];
            int idx = h_blockMinIdx[b];
            if (idx >= 0 && (val < minVal || (val == minVal && idx <
minIdx))) {
                minVal = val;
                minIdx = idx;
            }
        }

        swapElements<<<1, 1>>>(d_arr, start, minIdx);
        err = cudaGetLastError();
        if (err != cudaSuccess) {
            free(h_blockMinVal);
            free(h_blockMinIdx);
            cudaFree(d_blockMinVal);
            cudaFree(d_blockMinIdx);
            cudaFree(d_arr);
            return 1;
        }
        err = cudaDeviceSynchronize();
        if (err != cudaSuccess) {
            free(h_blockMinVal);
            free(h_blockMinIdx);
            cudaFree(d_blockMinVal);
            cudaFree(d_blockMinIdx);
            cudaFree(d_arr);
            return 1;
        }

        free(h_blockMinVal);
        free(h_blockMinIdx);
        cudaFree(d_blockMinVal);
        cudaFree(d_blockMinIdx);
    }

    err = cudaMemcpy(h_arr, d_arr, bytes, cudaMemcpyDeviceToHost);
    if (err != cudaSuccess) {
        cudaFree(d_arr);
```

```
        return 1;
    }

    cudaFree(d_arr);
    return 0;
}

int main() {
    int n = 0;
    scanf("%d", &n);
    if (n <= 0) {
        return 1;
    }

    int bytes = n * (int)sizeof(int);
    int *h_arr = (int *)malloc(bytes);
    if (h_arr == NULL) {
        return 1;
    }

    for (int i = 0; i < n; ++i) {
        scanf("%d", &h_arr[i]);
    }

    if (selectionSortCuda(h_arr, n) != 0) {
        selectionSortHost(h_arr, n);
    }

    for (int i = 0; i < n; ++i) {
        printf("%d", h_arr[i]);
        if (i + 1 < n) {
            printf(" ");
        }
    }
    printf("\n");

    free(h_arr);
    return 0;
}
```

Output :

```
sinhaa@MIT-ICT-LAB5-27:~/Desktop/College/Sem6/PCAP/Lab/Lab6$ ./q2
5
1 6 4 2 1
1 1 2 4 6
```

Q3 – Odd-Even Transposition Sort

```
// Title  : Odd-Even Transposition Sort using CUDA
// Author : Aditya Sinha
// Date   : 20/02/2026

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

__global__ void oddEvenPhase(int *arr, int n, int phase) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int i = 2 * tid + phase;
    if (i + 1 < n) {
        int left = arr[i];
        int right = arr[i + 1];
        if (left > right) {
            arr[i] = right;
            arr[i + 1] = left;
        }
    }
}

static void oddEvenSortHost(int *arr, int n) {
    for (int iter = 0; iter < n; ++iter) {
        int start = iter % 2;
        for (int i = start; i + 1 < n; i += 2) {
            if (arr[i] > arr[i + 1]) {
                int temp = arr[i];
                arr[i] = arr[i + 1];
                arr[i + 1] = temp;
            }
        }
    }
}

static int oddEvenSortCuda(int *h_arr, int n) {
    int bytes = n * (int)sizeof(int);
    int *d_arr = NULL;
    cudaError_t err = cudaMalloc((void **)&d_arr, bytes);
    if (err != cudaSuccess) {
        return 1;
    }
    err = cudaMemcpy(d_arr, h_arr, bytes, cudaMemcpyHostToDevice);
    if (err != cudaSuccess) {
        cudaFree(d_arr);
        return 1;
    }

    int threads = 256;
    int pairs = n / 2;
    int blocks = (pairs + threads - 1) / threads;

    for (int iter = 0; iter < n; ++iter) {
        oddEvenPhase<<<blocks, threads>>>(d_arr, n, 0);
```

```
        err = cudaGetLastError();
        if (err != cudaSuccess) {
            cudaFree(d_arr);
            return 1;
        }
        err = cudaDeviceSynchronize();
        if (err != cudaSuccess) {
            cudaFree(d_arr);
            return 1;
        }
        oddEvenPhase<<<blocks, threads>>>(d_arr, n, 1);
        err = cudaGetLastError();
        if (err != cudaSuccess) {
            cudaFree(d_arr);
            return 1;
        }
        err = cudaDeviceSynchronize();
        if (err != cudaSuccess) {
            cudaFree(d_arr);
            return 1;
        }
    }

    err = cudaMemcpy(h_arr, d_arr, bytes, cudaMemcpyDeviceToHost);
    if (err != cudaSuccess) {
        cudaFree(d_arr);
        return 1;
    }

    cudaFree(d_arr);
    return 0;
}

int main() {
    int n = 0;
    scanf("%d", &n);
    if (n <= 0) {
        return 1;
    }

    int bytes = n * (int)sizeof(int);
    int *h_arr = (int *)malloc(bytes);
    if (h_arr == NULL) {
        return 1;
    }

    for (int i = 0; i < n; ++i) {
        scanf("%d", &h_arr[i]);
    }

    if (oddEvenSortCuda(h_arr, n) != 0) {
        oddEvenSortHost(h_arr, n);
    }
```

```
    for (int i = 0; i < n; ++i) {
        printf("%d", h_arr[i]);
        if (i + 1 < n) {
            printf(" ");
        }
    }
    printf("\n");

    free(h_arr);
    return 0;
}
```

Output :

```
sinhaa@MIT-ICT-LAB5-27:~/Desktop/College/Sem6/PCAP/Lab/Lab6$ ./q3
6
9 3 7 1 5 2
1 2 3 5 7 9
```