High Performance Parallel Computing
Assignment: Filtering with CUDA
Part 1 — 1-D Filter

Objective

This handout walks through a working CUDA implementation of a 1-D filter (stencil / convolution), explains the concept of *halo elements* used to handle block boundaries in shared memory, and provides a small hand-checkable test run.

1 Part 1: 1-D Filter in CUDA (Reference Example)

A 1-D filter (also called a stencil or convolution operation) computes each output sample as a weighted combination of neighboring input samples:

$$y[i] = \sum_{k=-R}^{R} h[k]\, x[i+k]$$

where h is the filter kernel of radius R, x is the input signal, and y is the output signal. Below is a complete, working CUDA implementation.

```
#include <cstdio>
#include <cstdlib>
#include <cuda_runtime.h>

#define RADIUS      3
#define BLOCK_SIZE  256

// 1-D convolution kernel. Each thread computes one output element.
// Uses shared memory to cache the block's input tile plus halo regions.
__global__ void filter1D(const float *in, float *out, const float *h, int
    n)
{
    __shared__ float tile[BLOCK_SIZE + 2 * RADIUS];

    int gindex = blockIdx.x * blockDim.x + threadIdx.x;
    int lindex = threadIdx.x + RADIUS;

    // Load the center element
    if (gindex < n)
        tile[lindex] = in[gindex];
    else
        tile[lindex] = 0.0f;
```

```
22
23      // Load halo elements (left and right of the block)
24      if (threadIdx.x < RADIUS) {
25          int left = gindex - RADIUS;
26          tile[lindex - RADIUS] = (left >= 0) ? in[left] : 0.0f;
27
28          int right = gindex + BLOCK_SIZE;
29          tile[lindex + BLOCK_SIZE] = (right < n) ? in[right] : 0.0f;
30      }
31
32      __syncthreads();
33
34      if (gindex < n) {
35          float result = 0.0f;
36          for (int k = -RADIUS; k <= RADIUS; k++)
37              result += h[k + RADIUS] * tile[lindex + k];
38          out[gindex] = result;
39      }
40  }
41
42  int main(void)
43  {
44      const int n = 1 << 20;                  // 1M elements
45      const int kernelSize = 2 * RADIUS + 1;
46      size_t bytes = n * sizeof(float);
47
48      float *h_in  = (float *)malloc(bytes);
49      float *h_out = (float *)malloc(bytes);
50      float *h_kernel = (float *)malloc(kernelSize * sizeof(float));
51
52      for (int i = 0; i < n; i++) h_in[i] = (float)(rand() % 100);
53      for (int i = 0; i < kernelSize; i++) h_kernel[i] = 1.0f / kernelSize;
            // moving average
54
55      float *d_in, *d_out, *d_kernel;
56      cudaMalloc(&d_in,  bytes);
57      cudaMalloc(&d_out, bytes);
58      cudaMalloc(&d_kernel, kernelSize * sizeof(float));
59
60      cudaMemcpy(d_in, h_in, bytes, cudaMemcpyHostToDevice);
61      cudaMemcpy(d_kernel, h_kernel, kernelSize * sizeof(float),
            cudaMemcpyHostToDevice);
62
63      int gridSize = (n + BLOCK_SIZE - 1) / BLOCK_SIZE;
64      filter1D<<<gridSize, BLOCK_SIZE>>>(d_in, d_out, d_kernel, n);
65      cudaDeviceSynchronize();
66
67      cudaMemcpy(h_out, d_out, bytes, cudaMemcpyDeviceToHost);
68
69      printf("y[0] = %f, y[n-1] = %f\n", h_out[0], h_out[n - 1]);
70
71      cudaFree(d_in); cudaFree(d_out); cudaFree(d_kernel);
72      free(h_in); free(h_out); free(h_kernel);
73      return 0;
```

```
74 }
```

Points to note

- Each thread is responsible for exactly one output element.
- Shared memory is used to stage a tile of the input, including a *halo* of `RADIUS` elements on each side, so that neighboring threads do not repeatedly re-read global memory.
- Boundary conditions (indices outside $[0, n)$) are handled by zero-padding.
- `__syncthreads()` ensures the whole tile (including halos) is loaded before any thread starts computing its output.

Understanding Halo Elements

When a stencil operation (like a filter) is split across CUDA thread blocks, each block is responsible for computing a contiguous *core* range of output elements. However, computing an output element near the edge of a block's range requires *input* elements that belong to the **neighboring block**. These extra input elements — `RADIUS` elements just outside a block's own core range, on each side — are called **halo elements** (sometimes called *ghost* or *apron* cells).

Without halo elements, a thread near the boundary of a block would need to read from another block's shared memory, which CUDA does not allow directly. Instead, every block loads its own copy of the required halo elements *from global memory* into its shared-memory tile, so that all the data a block needs is self-contained once loading is complete.

The figure below illustrates this for `BLOCK_SIZE = 8` and `RADIUS = 3` (the same `RADIUS` used in the code above, with a smaller `BLOCK_SIZE` chosen only so the picture fits on the page).

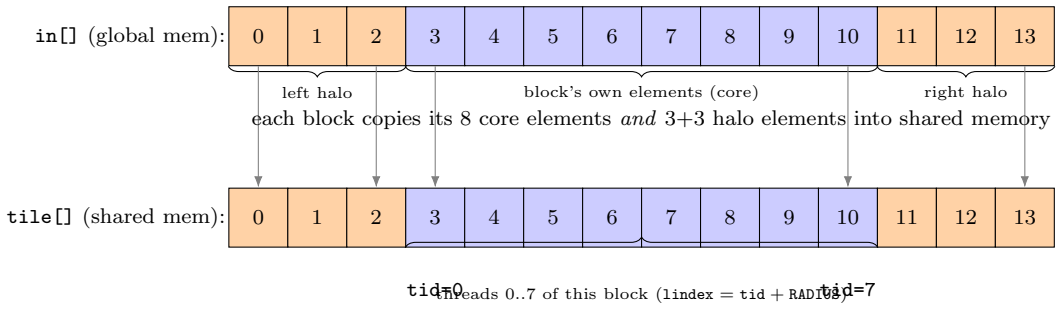

Figure 1: A block with `BLOCK_SIZE = 8` core elements (indices 3–10) also loads `RADIUS = 3` halo elements on each side (indices 0–2 and 11–13) from global memory into its shared-memory tile. Thread `tid` writes `tile[tid]` for the core element and, only if `tid < RADIUS`, additionally loads one left-halo and one right-halo element. After `__syncthreads()`, every thread can read the `2*RADIUS+1` neighborhood it needs entirely out of shared memory.

Test Run

To verify a 1-D filter implementation, it helps to run it on a small, hand-checkable example rather than the 1M-element random array used in `main()`. Consider $n = 10$, `RADIUS = 1`, and a 3-tap

averaging kernel $h = [0.25,\ 0.5,\ 0.25]$ applied to the input

$$x = [\,2,\ 4,\ 6,\ 8,\ 10,\ 12,\ 14,\ 16,\ 18,\ 20\,].$$

Because x is an arithmetic sequence and h is symmetric, every *interior* output equals the corresponding input value; only the last element is affected by zero-padding (there is no valid $x[10]$):

$$y[9] = 0.25\,x[8] + 0.5\,x[9] + 0.25 \cdot 0 = 0.25(18) + 0.5(20) = 14.5.$$

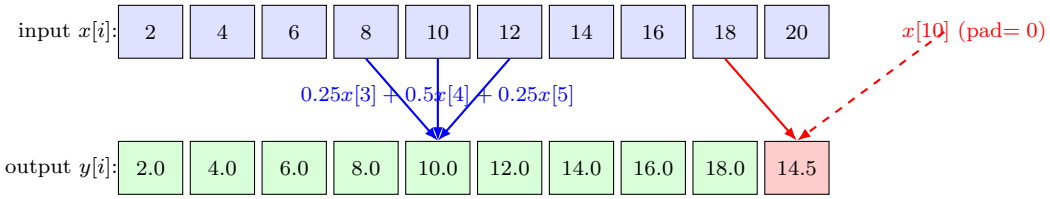

Figure 2: Small hand-verifiable test case: interior outputs simply reproduce the input (blue arrows show the 3-tap window computing $y[4]$), while the last output $y[9]$ is pulled down by the implicit zero-padding at the right boundary (red).

Running the CUDA kernel on this input (with `BLOCK_SIZE` ≥ 10 so everything fits in one block) should print:

```
x = [ 2.0  4.0  6.0  8.0 10.0 12.0 14.0 16.0 18.0 20.0 ]
y = [ 2.0  4.0  6.0  8.0 10.0 12.0 14.0 16.0 18.0 14.5 ]
```