**To DO List**

Finish copying the cope into int(main)

Certainly! I’ll modify the code to keep the 2D Gaussian kernel for later use, but I’ll adapt the code for 1D smoothing as per your current task requirements. The 1D kernel will be added separately without modifying or removing the existing 2D kernel code.

Here's the updated code, preserving your original 2D kernel while adding the necessary elements for 1D smoothing:

```cpp

#include <iostream>

#include <vector>

#include <cuda\_runtime.h>

#include <device\_launch\_parameters.h>

#include <random> // To generate random float values

using namespace std;

// Keep the 5x5 Gaussian kernel for later use in 2D smoothing.

constexpr int RADIUS\_2D = 2;

constexpr float KERNEL\_2D[] = {

0.003765, 0.015019, 0.023792, 0.015019, 0.003765,

0.015019, 0.059912, 0.094907, 0.059912, 0.015019,

0.023792, 0.094907, 0.150342, 0.094907, 0.023792,

0.015019, 0.059912, 0.094907, 0.059912, 0.015019,

0.003765, 0.015019, 0.023792, 0.015019, 0.003765,

};

// Define a 1D Gaussian kernel for 1D smoothing.

constexpr int RADIUS\_1D = 2;

constexpr float KERNEL\_1D[] = {

0.0625, 0.25, 0.375, 0.25, 0.0625

};

// GPU kernel for 1D smoothing

\_\_global\_\_ void smooth1D(float\* values\_in, float\* values\_out, int N)

{

int idx = threadIdx.x + blockIdx.x \* blockDim.x;

// Prevent out-of-bounds access

if (idx >= N) return;

// Temporary variable to hold the weighted sum

float tmp = 0.0f;

// Apply the 1D smoothing using the 1D Gaussian kernel

for (int k = -RADIUS\_1D; k <= RADIUS\_1D; ++k)

{

int neighbor\_idx = min(max(idx + k, 0), N - 1); // Clamp to boundary

int kernel\_idx = k + RADIUS\_1D; // Translate kernel index to [0, 2\*RADIUS\_1D]

tmp += values\_in[neighbor\_idx] \* KERNEL\_1D[kernel\_idx];

}

// Store the smoothed value

values\_out[idx] = tmp;

}

int main(int argc, char\*\* argv)

{

// Set up parameters for the 1D smoothing task

const int N = 1024; // Size of the vector

vector<float> h\_values\_in(N); // Input values

vector<float> h\_values\_out(N); // Output values

// Initialize the input vector with random float values between 0 and 1

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_real\_distribution<> dis(0.0, 1.0);

for (int i = 0; i < N; ++i)

{

h\_values\_in[i] = static\_cast<float>(dis(gen));

}

// Allocate device memory

float\* d\_values\_in = nullptr;

float\* d\_values\_out = nullptr;

cudaMalloc((void\*\*)&d\_values\_in, N \* sizeof(float));

cudaMalloc((void\*\*)&d\_values\_out, N \* sizeof(float));

// Copy input values to the device

cudaMemcpy(d\_values\_in, h\_values\_in.data(), N \* sizeof(float), cudaMemcpyHostToDevice);

// Define block and grid dimensions for 1D smoothing

const int THREADS\_PER\_BLOCK = 256;

const int BLOCKS = (N + THREADS\_PER\_BLOCK - 1) / THREADS\_PER\_BLOCK;

// Launch the 1D smoothing kernel

smooth1D <<<BLOCKS, THREADS\_PER\_BLOCK>>>(d\_values\_in, d\_values\_out, N);

// Copy the results back to the host

cudaMemcpy(h\_values\_out.data(), d\_values\_out, N \* sizeof(float), cudaMemcpyDeviceToHost);

// Print out the smoothed values

cout << "Smoothed 1D values:" << endl;

for (int i = 0; i < N; ++i)

{

cout << h\_values\_out[i] << " ";

if (i % 10 == 9) cout << endl; // Print 10 values per line

}

cout << endl;

// Clean up GPU memory

cudaFree(d\_values\_in);

cudaFree(d\_values\_out);

cout << "1D Smoothing Completed" << endl;

return 0;

}

```

### Key Changes and Additions:

1. \*\*Kept the Original 5x5 Gaussian Kernel (2D Kernel)\*\*:

- Retained the `KERNEL\_2D` constant array in the code, which is meant for 2D smoothing.

2. \*\*Added a 1D Gaussian Kernel\*\*:

- Added `KERNEL\_1D` as a separate array for 1D smoothing operations. This kernel has 5 elements, which is consistent with the Gaussian weights.

3. \*\*Created a Separate `smooth1D` Kernel\*\*:

- The `smooth1D` kernel is defined specifically to handle 1D vector smoothing using the `KERNEL\_1D`. It calculates smoothed values using neighboring elements in the 1D array.

4. \*\*Main Function Adapted for 1D Smoothing\*\*:

- The `main` function now generates a vector of random float values, allocates GPU memory, and uses the `smooth1D` kernel to perform the smoothing.

- The results are copied back to the host and printed to the console.

### Future 2D Smoothing:

When you need to perform 2D smoothing later, you can switch back to using the `KERNEL\_2D` and adapt the existing `smooth` kernel you have. This will be easy since both the 1D and 2D kernel definitions are preserved in the code.

If you'd like further modifications or explanations, feel free to ask!