# Kuwahara Filter Implementation Report

Le Chi Thanh

November 11, 2024

## 1 Introduction

The Kuwahara filter is an edge-preserving smoothing filter that reduces noise while maintaining edge clarity. This report details the CUDA-based implementation, optimizations, and performance analysis.

## 2 Implementation Details

### 2.1 RGB to HSV Conversion (Scatter Pattern)

The RGB to HSV conversion was implemented in Lab-work 8 (Scatter) so we just reuse the solution. The scatter pattern is used here because we're transforming data from an Array of Structures (AoS) format (where RGB values are packed together) to a Structure of Arrays (SoA) format (separate H, S, and V arrays).

```
@cuda.jit
def RGB2HSV(rgb_in, h_out, s_out, v_out):
    tidx = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
    tidy = cuda.threadIdx.y + cuda.blockIdx.y * cuda.blockDim.y

    if tidx < rgb_in.shape[1] and tidy < rgb_in.shape[0]:
        # GATHER: gather RGB values
        r = rgb_in[tidy, tidx, 0] / 255.0
        g = rgb_in[tidy, tidx, 1] / 255.0
        b = rgb_in[tidy, tidx, 2] / 255.0

        [... conversion code ...]

        # SCATTER: Write results to separate arrays
        h_out[tidy, tidx] = h
        s_out[tidy, tidx] = s
        v_out[tidy, tidx] = v
```

This conversion is necessary for the Kuwahara filter because we need the V (Value) component to calculate standard deviations in each window quadrant, while maintaining access to the original RGB values for final color averaging. The scatter approach helps optimize memory access patterns in subsequent kernel operations.

### 2.2 Filter without Shared Memory

The Kuwahara filter kernel operates on each pixel independently. It works by dividing the neighborhood around each pixel into four windows (quadrants), computing statistics for each quadrant, and selecting the quadrant with the lowest standard deviation to determine the output pixel value. The parallel patterns we use are:

- Scatter: Initial RGB to HSV conversion
- Gather: Reading neighborhood pixels for each quadrant
- Map: Calculating statistics for each quadrant
- Reduction: Finding minimum variance quadrant

For each pixel, we calculate it's new RGB values from it's surrounding 4 windows (called quadrant), following the lowest standard deviation conditions of the Kuwahara operator. The gather pattern here is a window-based gathering of pixel values, to deal with edge pixels we also have to implement boundary checking and non-coalesced memory access.

```
for quadrant in range(4):
    # GATHER pattern: Read neighborhood values
    if quadrant == 0:   # Top-left quadrant
        start_x, end_x = max(tidx - window_size, 0), tidx + 1
        start_y, end_y = max(tidy - window_size, 0), tidy + 1
    # ... other quadrants similarly

    # GATHER values from the quadrant
    for y in range(start_y, end_y):
        for x in range(start_x, end_x):
            v_val = v_in[y, x]      # Gather V channel
            rgb_vals = rgb_in[y, x]  # Gather RGB values
```

After that, for each quadrant we perform the local reduction to compute the new RGB values and standard deviation using V values (brightness) from HSV conversion.

```
# REDUCE pattern: Compute means and variance
    mean_v = sum_v / count  # Reduce to mean
    variance = (sum_v_squared / count) - (mean_v * mean_v)  # Reduce to variance
    std_dev = math.sqrt(max(variance, 0.0))

    # Reduce RGB sums to means
    means_rgb[quadrant][0] = sum_rgb[0] / count  # R mean
    means_rgb[quadrant][1] = sum_rgb[1] / count  # G mean
    means_rgb[quadrant][2] = sum_rgb[2] / count  # B mean
```

After the condition checking, we write the results to separate RGB channels:

```
# SCATTER pattern: Write to separate RGB channels
    rgb_out[tidy, tidx, 0] = means_rgb[chosen_quadrant][0]  # R
    rgb_out[tidy, tidx, 1] = means_rgb[chosen_quadrant][1]  # G
    rgb_out[tidy, tidx, 2] = means_rgb[chosen_quadrant][2]  # B
```

While this implementation is straight-forward and follows strictly the math idea, it faces several key limitations. The most significant performance bottleneck comes from multiple global memory gather operations required for each pixel, where each thread must repeatedly access the global memory to gather values from its neighborhood window. The algorithm also suffers from potential thread divergence, particularly at image boundaries where threads must handle edge cases differently, causing warps to execute different code paths and reducing parallel efficiency. Finally, since the reduction operations (computing means and variances) are limited to thread scope rather than block scope, each thread must perform its own complete set of calculations, leading to non-efficient computations.

## 2.3 Filter with Shared Memory

Similar to with-out shared memory solution, but now we use shared memory tiles (include padding) for window neighborhood, we separate as shared arrays for RGB and V values to optimize the memory access. Each shared arrays has a fixed size allocation, ideally it should be larger than block_width + window_size * 2 (for padding calculation ), adding some extra to avoid register pressure.

```
shared_dim = block_width + 2 * window_size
check = 32 # Fixed size allocation (32x32) for shared memory constraints
shared_rgb = cuda.shared.array(shape=(check, check, 3), dtype=np.float32)
shared_v = cuda.shared.array(shape=(check, check), dtype=np.float32)
```

For each thread block. we copy the threads value into shared memory. This also come with some optimization technique like boundary checking and coalesced memory access.

```
for dy in range(0, shared_dim, block_height):
    for dx in range(0, shared_dim, block_width):
        shared_y = ty + dy
        shared_x = tx + dx
        global_y = tidy - window_size + dy
        global_x = tidx - window_size + dx

```

```
        if (global_y >= 0 and global_y < height and
            global_x >= 0 and global_x < width):
            shared_rgb[shared_y, shared_x, 0] = rgb_in[global_y, global_x, 0]
            # ... load G, B, V components
```

Now for each pixel in shared memory, we compute statistic value of each quadrant and storing the the means and standard deviation of quadrants into local arrays.

```
for quadrant in range(4):
    # ... quadrant boundary calculations
    for yi in range(y_start, y_end):
        for xi in range(x_start, x_end):
            count += 1
            r = shared_rgb[yi, xi, 0] # read from shared memory, write to local
    varibale
            g = shared_rgb[yi, xi, 1]
            b = shared_rgb[yi, xi, 2]
            v = shared_v[yi, xi]
```

This method with shared memory offers better performance (processing time). It reduces global memory bandwidth usage by loading data into shared memory once and reusing it across multiple threads in the same block (pixels are loaded once per block rather than per thread during the calculation). Shared memory has much lower latency for repeated access patterns (typically 100x faster than global memory). While adding complexity in managing shared memory tiles, it greatly contributes to faster execution time and better resource utilization compared to the non-shared memory version (complexity and performance trade-off).

# 3 Performance Analysis

## 3.1 Speedup Comparison

Our results demonstrate the significant advantages of GPU acceleration for image transformation over CPU processing. While CPU processing would need to handle each pixel's quadrant calculation sequentially, the GPU process them in parallel, guarantee for the speedup. On a 400x600x3 image (720,000 pixels), the shared memory implementation achieved a 1.79x speedup compared to the non-shared version, reducing execution time from 0.010 to 0.006 seconds. This improvement is primarily due to reduced global memory access latency and better data locality through shared memory caching. Additionally, the block size comparison revealed that a 16x16 configuration (0.007 seconds) significantly outperformed the 8x8 configuration (0.016 seconds), suggesting better GPU resource utilization with larger block sizes. These results clearly show that the Kuwahara filter, which requires intensive neighborhood calculations for each pixel, benefits substantially from GPU parallelization and memory optimization strategies.

## 3.2 Memory Access Optimization

We highlight some optimization technique:
**Memory Coalescing:** How threads within a warp access global memory. When memory accesses are coalesced, multiple memory operations from threads in the same warp can be combined into a single memory transaction.

```
# FOR EACH PIXEL: Load data into shared memory
for dy in range(0, shared_dim, block_height):
    for dx in range(0, shared_dim, block_width):
        shared_y = ty + dy
        shared_x = tx + dx
        # Coalesced loading pattern
        global_y = tidy - window_size + dy
        global_x = tidx - window_size + dx

        if (global_y >= 0 and global_y < height and
            global_x >= 0 and global_x < width):
            shared_rgb[shared_y, shared_x, 0] = rgb_in[global_y, global_x, 0]  # R
            shared_rgb[shared_y, shared_x, 1] = rgb_in[global_y, global_x, 1]  # G
            shared_rgb[shared_y, shared_x, 2] = rgb_in[global_y, global_x, 2]  # B
```

This code-writing-technique allows consecutive threads access consecutive memory locations. It takes advantage of memory transactions being done in blocks and reduce memory bandwidth.
**Bank Conflict**: When multiple threads in a warp try to access different addresses in the same bank simultaneously, they have to wait in line, causing serialization.

```
# 3D layout helps avoid bank conflicts
shared_rgb[shared_y, shared_x, 0] = rgb_in[global_y, global_x, 0]
```

We using 3D (or simplified 2D) layout ti reduces bank conflicts compared to 1D, in this way threads in a warp can access different banks more better, a worse writing style would be:

```
# Something like this, not optimized
shared_rgb_r[shared_y, shared_x]
shared_rgb_g[shared_y, shared_x]
```

### Reduce Register Pressure

```
# We write
shared_rgb = cuda.shared.array(shape=(check, check, 3), dtype=np.float32)
shared_v = cuda.shared.array(shape=(check, check), dtype=np.float32)
#Instead of something like
shared_r = cuda.shared.array(shape=(check, check), dtype=np.float32)
shared_g = cuda.shared.array(shape=(check, check), dtype=np.float32)
shared_b = cuda.shared.array(shape=(check, check), dtype=np.float32)
shared_v = cuda.shared.array(shape=(check, check), dtype=np.float32)
```

This writing technique require less register variables: Each shared memory array declaration requires a register to store (which is limited per thread, we already using register for: thread indices, loop counter, calculation, array indices). By using a single 3D array instead of three 2D arrays, we only need one register for the RGB data instead of three, leaving more register to be used for other complex calculations.