Kuwahara Filter Implementations and Performance Check on CPU and GPU

Professor: Tran Giang Son
Student: Nguyen Tu Tung

November 7, 2024

1 Overview of the Kuwahara Filter

The Kuwahara filter is a non-linear, edge-preserving smoothing filter commonly used in image processing. It used is by dividing the neighborhood around each pixel into four overlapping sections, computing the mean and variance of each region. The mean has the lowest variance of the section will be used to replace the value of its pixel. This method will produce a distinctive "painterly" or smoothed appearance that is helpful in both visual applications and noise reduction jobs where edges must be preserved and smoothing consistent regions.

The total area of the neighborhood throughout which smoothing takes place is determined by the main parameter of the Kuwahara filter, ‘omega‘. Stronger smoothing can be accomplished through a larger ‘omega‘, but computing complexity also rises. This feature makes the Kuwahara filter computationally difficult, especially when dealing with huge pictures or high ’omega‘ values. The performance optimization of the Kuwahara filter on CPU and GPU platforms is addressed in this work, along with how different approaches manage the computational needs of the filter, especially for large ‘omega‘ values.

2 Introduction

With a special focus on GPU improvements using CUDA, this work evaluates the Kuwahara filter’s launch and improvement across CPU and GPU platforms. Our goal is for improved execution speed and scalability through deploy the filter in GPU and improving memory access, especially through shared memory. The effectiveness of several approaches to implementation is evaluated over different neighborhood sizes (controlled by the filter parameter, ‘omega‘). These approaches span from a sequential CPU-based approach to different GPU setups. In order to improve the filter’s capabilities for high-performance image processing, the research also investigates sophisticated shared memory setups that allow bigger ‘omega‘ values.(paralling)

3 Implementation Details

3.1 CPU Implementation

The Kuwahara filter's first implementation was executed through in a simple sequential way on the CPU. This method finds the mean and variance for every pixel in an image within a particular neighborhood that is divided into four sections. In order to smooth consistent areas and retain borders, each pixel is assigned the mean of the section with the least variation. Despite being successful, this approach is extremely costly, particularly for bigger photos at values of omega‘. It is a sequential method that lacks the speed benefits of parallel processing and is mainly used as a performance standard for GPU-based solutions.

3.2 GPU Implementation Without Shared Memory

Instead of utilizing shared memory, the original GPU method allocates each pixel to a separate thread. Because to uncoalesced access patterns, each thread directly accesses global memory for the vicinity of its pixel, resulting in increased memory latency. Without exchanging data, each thread computes the Kuwahara filter independently. The lack of shared memory causes repeated global memory accesses, which raises latency even though this approach uses GPU to outperform the CPU counterpart.

3.3 GPU Implementation With Shared Memory

The following GPU implementation uses shared memory to minimize the waiting time related to global memory access. Threads inside each block can access neighborhood data with low latency by storing it in on-chip shared memory. Because each block loads a tile of the picture into shared memory, with padding to account for the neighborhood size defined by ‘omega‘, this method minimizes unnecessary global memory access. However, because of the GPU's restricted per-block shared memory capacity, shared memory utilization imposes limitations, especially on the size of ‘omega‘. The maximum possible ‘omega‘ value in this arrangement is 8, after which memory constraints preclude proper operation. Despite this limitation, by reducing global memory latency, shared memory significantly boosts performance.

3.4 Handling High Omega Values with Optimized Shared Memory Management

The filtered picture sometimes displayed black dots and artifacts when omega‘ was increased to the highest values it can get (e.g., ‘omega = 8‘), which did not exist at lower ‘omega‘ values. Incorrect data processing results from the constraints in shared memory allocation while managing bigger neighborhoods, since the amount of memory needed per block surpasses the GPU's available shared memory capacity.

Solution: Optimizing the Use of Shared Memory via Parallelization: In order to solve this problem, we created a parallel Kuwahara filter implementation that makes effective use of shared memory, paying significant attention to memory coalescing and preventing bank conflicts. With 'omega = 8' or even 9, we were still able to obtain precise and artifact-free results thanks to this method, which also made sure the filter ran inside GPU memory constraints. This enhanced solution enhances memory consumption and performance by carefully controlling shared memory allocation and access patterns, enabling the filter to efficiently handle bigger neighborhoods.

4 Performance Evaluation

4.1 Experimental Setup

The implementations were tested on an image of specified dimensions, and each configuration was evaluated across multiple values of 'omega' (1, 3, 5, 7, 9). Execution times were recorded for each setup to provide a comprehensive comparison of performance across configurations.

5 Execution Time and Speedup Comparisons

We present both the execution times (in seconds) and speedup factors for each implementation. Speedup is calculated as follows:

$$\text{Speedup}_{\text{X vs. Y}} = \frac{\text{Execution Time of X}}{\text{Execution Time of Y}}$$

where X and Y represent two different implementations being compared.

Omega	CPU Time (s)	GPU Time w/o Shared Mem (s)	Speedup (CPU vs. GPU w/o Shared)
1	338.2432	0.0767	4409x
3	334.3252	0.0675	4951x
5	340.1106	0.0812	4188x
7	335.8716	0.0648	5181x
9	339.8808	0.0659	5155x

Table 1: Execution Times and Speedup: CPU vs. GPU without Shared Memory

Omega	GPU Time w/o Shared Mem (s)	GPU Time with Shared Mem (s)	Speedup (Non-Shared vs. Shared)
1	0.0767	0.0630	1.22x
3	0.0675	0.0600	1.13x
5	0.0812	0.0632	1.28x
7	0.0648	0.0517	1.25x
9	0.0659	N/A	N/A

Table 2: Execution Times and Speedup: GPU without Shared Memory vs. GPU with Shared Memory

Omega	GPU Time with Shared Mem (s)	Parallel GPU with Shared Mem (s)	Speedup (Shared vs. Parallel Shared)
1	0.0630	0.0510	1.24x
3	0.0600	0.0484	1.24x
5	0.0632	0.0441	1.43x
7	0.0517	0.0497	1.04x
9	N/A	0.0586	N/A

Table 3: Execution Times and Speedup: GPU with Shared Memory vs. Parallel GPU with Shared Memory

The results demonstrate that the parallel shared memory implementation achieves the best performance across all values of ‘omega‘, with significant speedup over the CPU baseline.

6 Conclusion

The results of this investigation shows how well GPU-based optimization methods work for the Kuwahara filter, especially when it comes to memory access control and shared memory use. Compared to a basic CPU implementation, the filter’s execution time is significantly increased by utilizing shared memory, memory coalescing, and eliminating bank conflicts. Performance is further improved by the sophisticated shared memory architecture, which makes it possible to handle bigger ’omega‘ values effectively. These results highlight how crucial memory optimization is for high-performance computing, particularly for activities like image processing that often access neighborhood data.