Lecture 1

GPU Accelerated Bilateral Filter for Video Enhancement

Xavier Akers

Contents

1.1 Introduction

Image and video processing is currently facing drastic growth, led by many fields including computer vision, video game rendering, and medical imaging. Modern technologies are producing images and videos at higher resolutions with more pixels than ever before, resulting in increased computational demands. Developing efficient and optimized methods for image and video processing has become a critical challenge.

1.1.1 The Bilateral Filter

The bilateral filter, which is an extension of the Gaussian blur, is a popular image processing technique. It is a non-linear, edge-preserving, and noise-reducing smoothing filter for images. The filter is defined as

$$I^{\text{filtered}}(x) = \frac{1}{W_p} \sum_{x_i \in \Omega} I(x_i) f_r(\|I(x_i) - I(x)\|) g_s(\|x_i - x\|)$$

and the normalization term, W_p, is given by

$$W_p = \sum_{x_i \in \Omega} f_r(\|I(x_i) - I(x)\|) g_s(\|x_i - x\|)$$

where

- I^{filtered} is the filtered image;
- I is the original input image to be filtered;
- x are the coordinates of the current pixel to be filtered;
- Ω is the window centered in x, so $x_i \in \Omega$ is another pixel;
- f_r is the range kernel for smoothing differences in intensities (this function can be a Gaussian function);
- g_s is the spatial kernel for smoothing differences in coordinates (this function can be a Gaussian function).

Essentially, the bilateral filter reduces noise in images while preserving structural features. It works by determining the intensity and spatial information of pixels within a local window to determine the new pixel value. This method is valuable where maintaining edges is crucial.

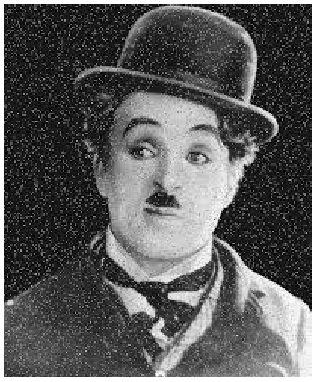
Figure 1.1: Original Image

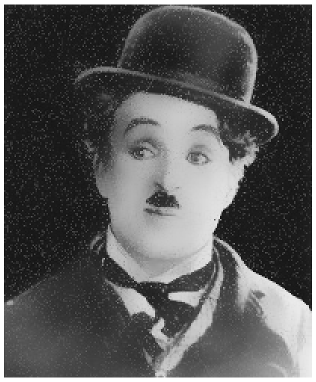
Figure 1.2: Filtered Image

1.1.2 Motivations

Processing images is computationally expensive, and this challenge is multiplied when dealing with high-resolution videos at high frame rates. The bilateral filter is straight forward to implement on a CPU, but the implementations are not scalable for large videos and real-time filters. CPUs have fewer degrees of parallelism, reduced bandwidth, and inefficient memory handling for this task.

However, GPUs are well-suited for this task with their highly parallel architecture. The bilateral filter's high computational complexity per frame but low computational complexity per pixel fits the strengths of GPUs, making it an ideal candidate for this problem.

1.2 Implementations

While the CPU implementation is straightforward—iterating over all the pixels and solving the equation—the GPU implementation is more complex. The naive approach might be to assign each thread to a pixel to compute the filtered value. However, this method fails to take advantage of the GPU's architecture. The bilateral filter on the GPU is a memory-bound task because each thread must load pixels from within the defined neighborhood. Additionally, copying each frame to and from the GPU is time-consuming. Reaching peak performance requires careful optimizations around the GPU's memory bandwidth and data transfers.

1.2.1 Required Packages

Our implementation requires the following packages.

Python

1. `numpy` for manipulating matrices.
2. `cv2` (OpenCV) for reading and writing images and videos.
3. `ctypes` for calling C functions and passing C pointers.
4. `pycuda` for handling data for the `DEVICE`.

CUDA

1. `CUDA toolkit` for the necessary libraries, compilers, and tools for developing CUDA code.

1.2.2 Pipeline

The pipeline for our GPU-accelerated bilateral filtering process has the following structure

1. **Read frames from the video**: We use OpenCV in Python to read video frames.
2. **Load frames into page-locked memory**: Using `PyCUDA`, we load the video frames into page-locked memory to optimize data transfers between the `HOST` and `DEVICE`. This step is essential to speeding up data transfers.
3. **Invoke CUDA filter**: We call the CUDA shared object from Python using `ctypes` for interactions with CUDA.

4. **Execute bilateral filter on the `DEVICE`**: The filter is applied to each frame on the `DEVICE`.
5. **Post-process and save filtered frames**: The filtered frames are processed and saved back to the disk into a viewable format using OpenCV in Python.

This workflow is visualized in Figure 1.3.

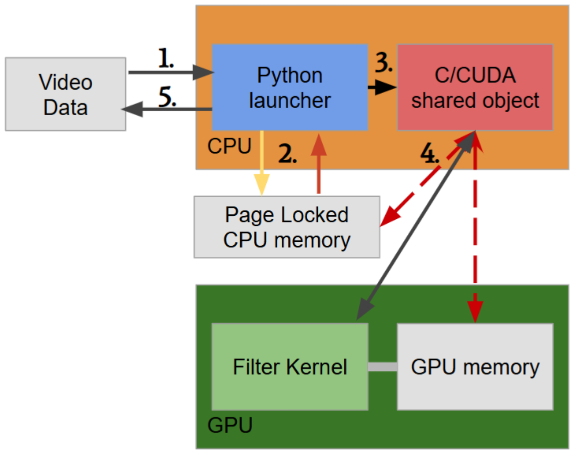

Figure 1.3: GPU-accelerated bilateral filter pipeline

1.2.3 Optimization Techniques

CUDA streams

One of the most expensive parts of video filtering is transferring each frame to and from the `DEVICE`. That is, the time spent copying the data can take longer than filtering the frame. We can mitigate this inefficiency by leveraging CUDA streams. Streams let us copy the next frame while the current frame is being processed.

Constant Cache

The spatial kernel applied to each frame is a constant lookup table. We can precompute the kernel and load them into constant cache. Constant cache has lower latency than both `DEVICE` memory and shared memory.

Shared Memory

Each thread can load its corresponding pixel into shared memory. This allows threads within a thread block to quickly access neighboring pixels.

Bit-Packing

Since each pixel consists of integer values ranging from 0 to 255, it can be represented as an 8-bit integer. However, the `DEVICE` performs efficient memory accesses of 32-bits chunks. By reading the frame in RGBA color channels and packing each channel into a 32-bit integer, we can take advantage of this. This also reduces the number of memory accesses since we only need to make one access per pixel instead of four.

1.2.4 Python

We can now take a look into the Python code. Our script takes the following arguments: `video_path.mp4`, `radius`, `intensity_sigma`, and `spatial_sigma`. We begin by loading in our necessary packages.

```python
import numpy as np
import cv2
import ctypes as ct
import pycuda.driver as cuda
```

We can utilize `numpy` to quickly compute the spatial kernel based on the input argument `radius` and `spatial_sigma`.

```python
def precompute_spatial_weight(radius, spatial_sigma):
    grid = np.arange(-radius, radius + 1)  # 1D grid for both x and y
    x, y = np.meshgrid(grid, grid)

    # Compute the squared Euclidean distance for the grid
    spatial_dist_sq = np.power(x, 2, dtype=np.float32) + np.power(y, 2, dtype=np.float32)

    # Compute Gaussian weights
    spatial_weights = np.exp(-spatial_dist_sq / (2 * np.power(spatial_sigma, 2, dtype=np.float32))
        )

    return spatial_weights
```

Next, we create our video reader object and collect properties about the frame.

```python
cap = cv2.VideoCapture(vid_path)
if not cap.isOpened():
    print(f"error : unable to open {vid_path}")
    exit(1)

# Some video properties
frame_width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
frame_height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
num_frames = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
frame_size = (frame_width, frame_height)
fps = cap.get(cv2.CAP_PROP_FPS)
colorspace = 4
```

Now that we know the number of frames and the size of each frame, we can allocate our page-locked memory using `pycuda`. This is essential for using CUDA streams later.

```python
# Create Page-locked memory
p_frames = cuda.pagelocked_empty(shape=(num_frames, frame_height+2*radius, frame_width+2*radius,
    colorspace), dtype=np.uint8)
p_filtered_frames = cuda.pagelocked_empty(shape=(num_frames, frame_height, frame_width, colorspace
    )dtype=np.uint8)
```

We read the frames directly into this page-locked array. The padding size is determined by `radius` to account for edge cases when filtering pixels at the boundaries of the image.

```python
# Read in all frames into page-locked memory
for idx in range(num_frames):
    ret, frame = cap.read()
    if not ret:
        break
    # Convert frame to RGBA and pad the edges
    frame_pad = np.pad(
        cv2.cvtColor(frame, cv2.COLOR_BGR2RGBA),
        pad_width=((radius, radius), (radius, radius), (0, 0)),
        mode="constant",
        constant_values=0,
    )
    p_frames[idx] = frame_pad
# Close video reader
cap.release()
```

We create the C pointers for our frames and spatial weights to be passed to the CUDA function.

```python
p_frames_cptr = np.ascontiguousarray(p_frames, dtype=np.uint8).ctypes.data_as(ct.POINTER(ct.
    c_uint8))
p_filtered_frames_cptr = np.ascontiguousarray(p_filtered_frames, dtype=np.uint8).ctypes.data_as(ct
    .POINTER(ct.c_uint8))
spatial_weights_cptr = np.ascontiguousarray(spatial_weights, dtype=np.float32).ctypes.data_as(ct.
    POINTER(ct.c_float))
```

We now invoke the CUDA object.

```python
# Load shared object library
lib = ct.cdll.LoadLibrary('./cudaBilatVid.so')
lib.filter(
    p_frames_cptr,
    p_filtered_frames_cptr,
    ct.c_int(num_frames),
    ct.c_int(frame_height),
    ct.c_int(frame_width),
    ct.c_int(colorspace),
    ct.c_int(radius),
    ct.c_float(intensity_factor),
    spatial_weights_cptr,
)
```

Once the filtering process is complete, we write our filtered frames to the disk and free our page-locked memory.

```python
# Set up video writer object
out = cv2.VideoWriter("output_raw.mp4", cv2.VideoWriter_fourcc(*"mp4v"), fps, frame_size)
for frame in p_filtered_frames:
    out.write(cv2.cvtColor(frame, cv2.COLOR_RGBA2BGR))
out.release()
p_frames.base.free()
p_filtered_frames.base.free()
```

The filtered video may need to be encoded for proper playback and can be done as follows.

```
subprocess.run(["ffmpeg","-hide_banner","-loglevel","quiet","-i","output_raw.mp4","-vcodec","
    libx264","-crf","23",outfile,"-y",],check=True,)
```

With the Python overhead handled, we can focus on the CUDA code.

1.2.5 CUDA

A shared object, also known as a dynamic library, is a compiled file that can be dynamically loaded and used by other programs at runtime. This approach allows us to optimize our bilateral filter by offloading the computations onto the GPU. The function header for the bilateral filter is defined as

```
void filter(
    uint8_t p_padded_frames[], uint8_t p_filtered_frames[], const int num_frames,
    const int frame_height, const int frame_width, const int colorspace,
    const int radius, const float intensity_factor, const float* spatial_weights);
```

As mentioned, to optimize memory access, we allocate constant memory for storing the precomputed spatial weights. This constant memory array is allocated at compiled time as follows.

```
__constant__ float const_spatial_weights[(2*MAXRADIUS+1)*(2*MAXRADIUS+1)];
```

where `MAXRADIUS` is a predefined maximum radius. We copy our spatial weights into constant cache as follows.

```
cudaMemcpyToSymbol(const_spatial_weights, spatial_weights, (2*radius+1)*(2*radius+1)*sizeof(float)
    , 0, cudaMemcpyDefault)
```

By precomputing these weights and storing them in constant memory, we reduce the overall memory latency.

Launch Configurations

With each thread loading its corresponding pixel into shared memory, we ensure each thread reads from `DEVICE` memory once. However, shared memory is only shared within a thread block, creating edge cases for pixels near the borders of the thread blocks that depend on neighboring pixels.

To handle this, we use a padded thread block technique. That is, the thread block will include a “halo” region with width `radius`. Threads within the halo region load additional data in shared memory, while only the central threads compute and write data back to global memory. This guarantees that all the required data is available in shared memory. Figure 1.4 illustrates this design.

To determine the number of thread blocks needed, we exclude the halo region from the thread block dimensions and calculate the grid size as follows.

```
const dim3 block(32, 32);
const dim3 grid(
    ((frame_width + (block.x - 2 * radius) - 1) / (block.x - 2 * radius)),
    ((frame_height + (block.y - 2 * radius) - 1) / (block.y - 2 * radius)));
```

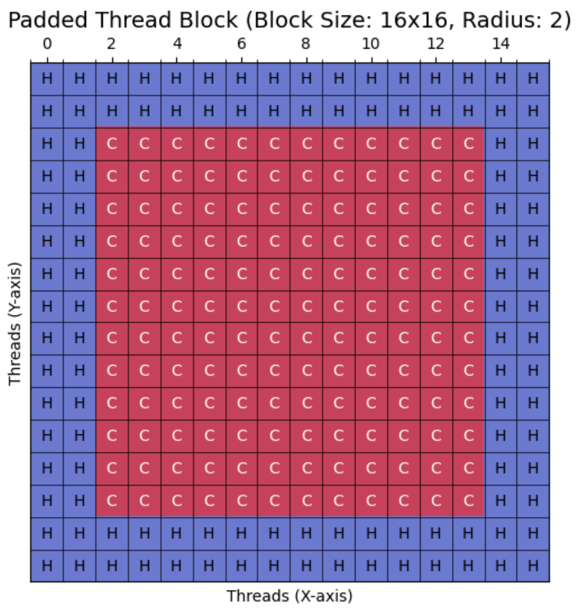

Figure 1.4: A 16x16 thread block padded with a halo region of `radius = 2`. Threads within the halo (denoted as H) load additional data into shared memory, while the central threads (denoted as C) perform computations and write back to `DEVICE` memory.

As mentioned, we incorporate CUDA streams into our algorithm to load in the next frame while processing the current one. This reduces time spent waiting for data to transfer over the `DEVICE` bus. Figure 1.5 illustrates the ideal loading process, influenced by the University of New England’s lecture notes (University of New England, Lecture 22). This approach is not completely scalable due to risk of flooding the PCIe bus.

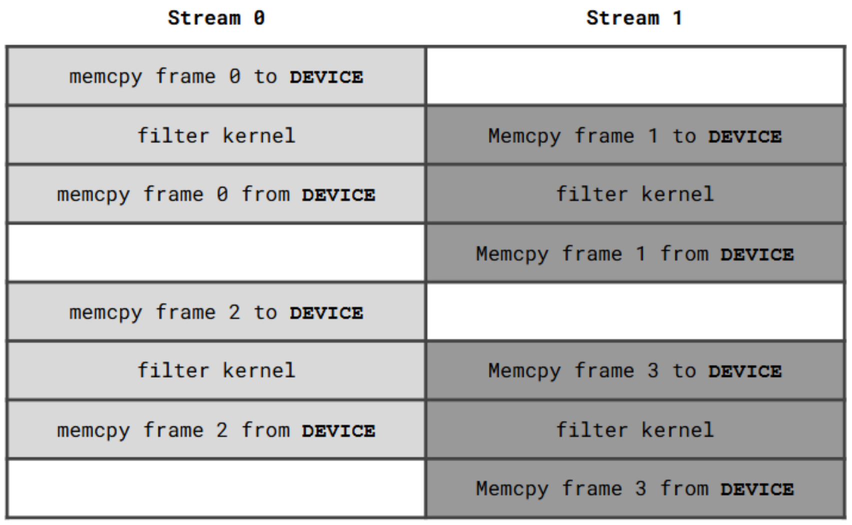

Figure 1.5: Illustration of the frame loading process using CUDA streams, enabling concurrent data loading and processing.

To implement this technique, we define the number of streams, create the streams, and allocate `DEVICE` buffers for each stream.

```
// Number of Streams
int num_streams = 2;

// Streams
cudaStream_t streams[num_streams];

// DEVICE buffers
uint32_t* d_frames[num_streams];
uint32_t* d_filtered_frames[num_streams];

for (int stream_idx = 0; stream_idx < num_streams; stream_idx++) {
    cudaStreamCreate(&streams[stream_idx]);
    cudaMalloc(&d_frames[stream_idx], pad_img_size);
    cudaMalloc(&d_filtered_frames[stream_idx], img_size);
}
```

Next, we process using `async()` functions. These functions return before completion, allowing the CPU to proceed with its next task. Note that the `uint8_t HOST` array is copied into a `uint32_t DEVICE` array, which effectively packs the bits into a 32-bit integer.

```
for (int idx = 0; idx < num_frames; idx++) {
    uint8_t* p_frame = &p_padded_frames[idx*(padded_frame_height*padded_frame_width*COLORSPACE})];
    uint8_t* p_filtered_frame = &p_filtered_frames[idx*(frame_height*frame_width*COLORSPACE)];

    // Get stream number
    int stream_idx = idx % num_streams;

    // Async copy to device from page-locked memory
    cudaMemcpyAsync(d_frames[stream_idx], p_frame, pad_img_size, cudaMemcpyHostToDevice, streams[
        stream_idx]);
    // Run the bilateral filter kernel
    bilateralFilterKernel<<<grid, block, 0, streams[stream_idx]>>>(
        d_frames[stream_idx], d_filtered_frames[stream_idx],
        frame_height, frame_width, colorspace,
        radius, intensity_factor);

    // Async copy filtered frame back to page-locked memory
    cudaMemcpyAsync(p_filtered_frame, d_filtered_frames[stream_idx], img_size,
         cudaMemcpyDeviceToHost, streams[stream_idx]);

    // Synchronize previous stream
    if (idx >= num_streams - 1) {
        cudaStreamSynchronize(streams[(stream_idx-1+num_streams) % num_streams]); // Ensure proper
             indexing
    }
}
```

With our kernel configurations complete, we can efficiently develop our CUDA kernel.

Kernel

We define our kernel as follows.

```
__global__ void bilateralFilterKernel(
    uint32_t* input_pad, uint32_t* output,
    const int rows, const int cols, const int colorspace,
    const int radius, const float intensity_factor);
```

Next, we allocate shared memory with one index for each thread in a thread block.

```
__shared__ uint32_t s_input_pad[32][32];
```

We then calculate the global padded frame index for accessing the global array and the local frame index for accessing shared memory.

```
// Calculate global and local thread indices
const int global_c_pad = threadIdx.x + blockIdx.x * (blockDim.x - 2 * radius);
const int global_r_pad = threadIdx.y + blockIdx.y * (blockDim.y - 2 * radius);

const int local_c = threadIdx.x;
const int local_r = threadIdx.y;

const int cols_pad = cols + 2 * radius;
const int rows_pad = rows + 2 * radius;
```

Once each thread is assigned to a pixel, we load the corresponding pixel into a local variable and write it to shared memory.

```
uint32_t centered_pixel;
// Each thread loads one element from global memeory to shared memory
if (global_r_pad < rows_pad && global_c_pad < cols_pad) {
    centered_pixel = input_pad[global_r_pad * cols_pad + global_c_pad];
    s_input_pad[local_r][local_c] = centered_pixel;
}
__syncthreads();
```

Since each RGBA color channel is packed into a 32-bit integer, we use bitwise operations to extract each channel.

```
// Extract color channels
uint8_t red = (centered_pixel >> 24) & 0xFF;
uint8_t green = (centered_pixel >> 16) & 0xFF;
uint8_t blue = (centered_pixel >> 8) & 0xFF;
uint8_t alpha = centered_pixel & 0xFF;
```

Next, the kernel performs the bilateral filter on its assigned pixel. Only the inner threads process their corresponding pixels.

```
// Only inner threads process its pixels
if ((local_r >= radius) && (local_r < blockDim.y - radius)
    && (local_c >= radius) && (local_c < blockDim.x - radius)
    && (global_r_pad >= radius) && (global_r_pad < rows + radius)
    && (global_c_pad >= radius) && (global_c_pad < cols + radius)) {

    // Total weight
    float Wp = 0.0f;
    float filtered_pixels[COLORSPACE] = { 0.0f };

    // Iterate over neighborhood
    for (int wi = -radius; wi <= radius; wi++) {
        for (int wj = -radius; wj <= radius; wj++) {
            int nbor_i = local_r + wi;
            int nbor_j = local_c + wj;

            uint32_t nbor_values = s_input_pad[nbor_i][nbor_j];

            // Extra nbor color channels
            uint8_t nbor_red = (nbor_values >> 24) & 0xFF;
            uint8_t nbor_green = (nbor_values >> 16) & 0xFF;
            uint8_t nbor_blue = (nbor_values >> 8) & 0xFF;
            uint8_t nbor_alpha = nbor_values & 0xFF;

            float intensity_dist_sq = (float)(nbor_red - red) * (nbor_red - red)
                + (float)(nbor_green - green) * (nbor_green - green)
                + (float)(nbor_blue - blue) * (nbor_blue - blue)
                + (float)(nbor_alpha - alpha) * (nbor_alpha - alpha);

            float weight = const_spatial_weights[(wi + radius) * (2 * radius + 1) + (wj + radius)]
                * lorentzian(intensity_dist_sq, intensity_factor);

            filtered_pixels[0] += weight * nbor_red;
            filtered_pixels[1] += weight * nbor_green;
            filtered_pixels[2] += weight * nbor_blue;
            filtered_pixels[3] += weight * nbor_alpha;

            Wp += weight;
        }
    }
    float Wp_inv = 1.0f / Wp;
    uint32_t packed_filtered_pixel = ((uint8_t)(filtered_pixels[0] / Wp_inv) << 24)
        | ((uint8_t)(filtered_pixels[1] * Wp_inv) << 16)
        | ((uint8_t)(filtered_pixels[2] * Wp_inv) << 8)
        | (uint8_t)(filtered_pixels[3] * Wp_inv);

    output[(global_r_pad - radius) * cols + (global_c_pad - radius)] = packed_filtered_pixel;
}
```

We also optimize the Gaussian function by approximating it with the Lorentzian function, defined as

$$\frac{1}{1+(x^2/2\sigma^2))}.$$

The Lorentzian function is often used when accuracy is not crucial. To futher improve computational performance, we precompute the inverse σ factor on the `HOST`.

```
intensity_factor = 1 / (2 * intensity_sigma * intensity_sigma)
```

We now have the optimized `DEVICE` function

```
__device__ inline float lorentzian(float x_sq, float factor) {
    // Approximate the Gaussian function with the Lorentizian function
    return 1.0f / (1.0f + x_sq * factor);
}
```

The CUDA kernel for the bilateral filter has been efficiently implemented using shared memory, reducing the time spent from `DEVICE` memory. By leveraging shared memory, we minimize expensive memory accesses during computation.

1.3 Results

The results were obtained from three different implementations: C, OpenMP, and CUDA. The tests were conducted on a system featuring an Nvidia GeForce GTX 1080 Ti GPU, an Intel Xeon E5-2620 v4 CPU with 32 cores, and 125 GB of memory. Two input videos were used for testing overall performance: a 21 second 3480x2160 (4k) resolution video at 25 fps, totaling at 528 frames, and a 10 second, 7680x4320 (8k) resolution video at 30 FPS, totaling at 300 frames. The filter was applied with a radius of 1.

1.3.1 Timing

The timing results for the bilateral filter for the 4k and 8k videos are summarized in Table 1.1 and Table 1.2, respectively. These timings do not include reading and writing the data to the disk.

4k Video Performance

Version	Time (s)	Time per Frame (ms)	Speed-Up (compared to C)
C	267.829	507.252	1.00
OpenMP	21.593	40.900	12.404
CUDA	2.764	5.235	96.899

Table 1.1: Timing results for the bilateral filter on a 4k video across different implementations.

As shown in Table 1.1, the CUDA implementation provides the best performance for the 4k video, with substantial speed-ups over the C and OpenMP implementation.

8k Video Performance

Version	Time (s)	Time per Frame (ms)	Speed-Up (compared to C)
C	607.943	2026.477	1.00
OpenMP	47.147	157.157	12.897
CUDA	5.972	19.907	101.799

Table 1.2: Timing results for the bilateral filter on an 8k video across different implementations.

For the 8k video, the CUDA implementation remains the fastest, as stated in Table 1.2. We can visualize the performance comparisons of the implementations in Figure 1.6 and view the proportion of total time spent for each implementation in Figure 1.7. While CUDA outperformed the other implementations, the performance scaled linearly by a factor of four from the resolution increase (4k to 8k), suggesting the GPU is not being fully utilized.

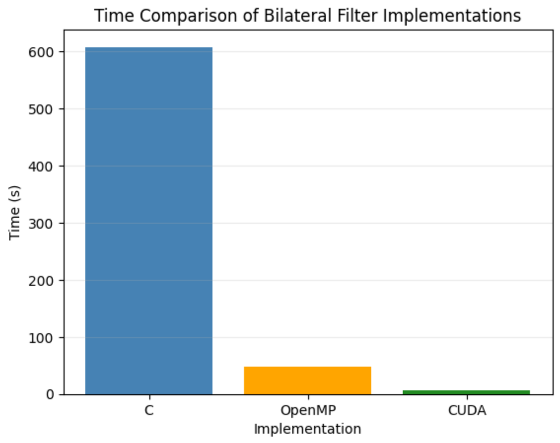

Figure 1.6: Filter times for bilateral filter implementations on the 8k video.

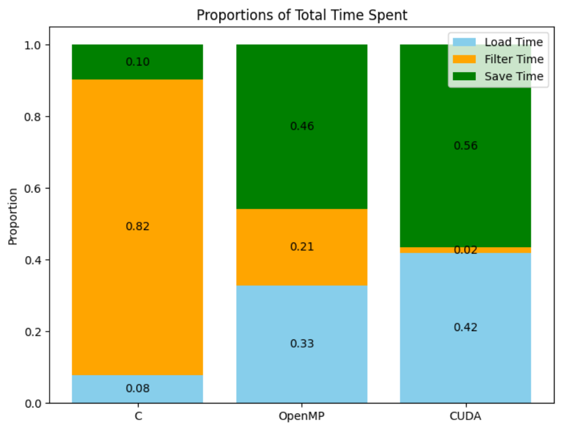

Figure 1.7: Proportion of overall time for bilateral filter implementations on the 8k video.

1.3.2 Profiling and Performance Analysis

We use `nvprof` to profile the CUDA implementation, measuring global memory reads, global memory writes, shared memory reads, shared memory writes, and achieved occupancy. Profiling was ran on the same 8k video with results detailed in Table 1.3.

Metric	Value
Global Memory Reads (GB/s)	50.885
Global Memory Writes (GB/s)	50.868
Shared Memory Reads (GB/s)	487.95
Shared Memory Writes (GB/s)	57.831
Achieved Occupancy	0.844787

Table 1.3: CUDA Profiling Results

We confirm that the CUDA implementation underutilizes the GPU. The global memory throughput of approximately 50 GB/s represents significant improvement over the naive implementation's throughput of less than 1 GB/s. However, we fail to reach the GeForce GTX 1080 Ti's theoretical peak of 484 GB/s. The most performance gains come from the high shared memory read throughput, as a result of the padded thread block design. Similarly, we fail to reach the peak shared memory bandwidth of 2.4 TB/s. The high achieved occupancy ($\tilde{8}4\%$) indicate efficient workload distribution. Despite all the memory optimizations, we conclude the kernel implementation is still bottlenecked by memory accesses, supporting the linear scaling of performance with image size.

1.4 Conclusion

1.4.1 Summary

The analysis of the three bilateral filter implementations proved the GPU to be a valuable candidate for this intensive task. Utilizing CUDA streams to concurrently load data to and from the GPU substantially increased overall performance. Benchmarks showed considerable speed-ups with CUDA compared to C and OpenMP. We minimized memory access times by leveraging constant cache, shared memory, and bit packing. However, even with the optimizations, the performance remained bottlenecked by memory.

1.4.2 Future Work

Future efforts will focus on optimizing the CUDA implementation further, particularly in memory management. Improving our shared memory utilization and leveraging `__shfl_sync()` instructions for faster loading of neighboring pixels in line with the columns. Loading entire datasets into memory is not always feasible for large videos and methods to chunk the data may be needed. Additionally, experimenting with arithmetic intensity balancing may help throughput without hindering performance. In the end, the goal is to implement a real-time webcam filter.

Public Git Repository
Google Colab Demo
Google Slides