CUDA-based AV1 Constrained Directional Enhancement Filter

Nuha Khadija Sahraoui and Won Sup Song

Abstract—The Constrained Directional Enhancement Filter (CDEF) is a key component of the AV1 video codec, designed to mitigate ringing artifacts while preserving natural edges. In this work, we present an optimized CUDA-based implementation of CDEF that exploits parallelism inherent in directional filtering. Our approach processes images in multiple stages, including RGB-to-YCbCr conversion, direction estimation, and non-linear filtering using primary and secondary taps. We leverage shared memory and thread synchronization to accelerate computations and ensure efficient memory access. Experimental results demonstrate significant improvements in filtering speed while maintaining high visual quality. We also discuss potential optimizations, including adaptive preset selection and lookup table-based constraint function evaluation, to further enhance the performance of the filter. Our findings highlight the advantages of GPU-accelerated filtering for real-time video processing applications.

Index Terms—Vectorization, CUDA, Video Compression, Directional Filtering, Parallel Computing, Image Processing, Edge Detection

1 Introduction

This paper describes the CUDA-based implementation of the constrained directional enhancement filter (CDEF) designed for the AV1 royalty-free video codec. [1] Its aim is to reduce ringing artifacts incurred by compression. The algorithms required to implement the stages of the filter are designed to exploit naturally-occurring data-level parallelism. Hence, CUDA was chosen for the single-instruction multiple data (SIMD) implementation of the AV1 CDEF.

The filter is a non-linear, low-pass filter with in-built pattern recognition. It targets high-frequency distortion by computing the "direction" of edges at each position in the image, and using the computed value to select a filter kernel that filters in the direction of the edge. This process ensures that natural edges are preserved in the filtered image.

Our project processes images in 4 primary stages, each implemented as a CUDA kernel with threads and blocks allocated strategically to leverage SIMD computing as often as possible.

2 Stage 1: Conversion from RGB to YCbCr Representation

Each frame in a video is converted to a 1D array for which memory is allocated to the GPU. The image, with dimensions $n \times m$, is represented by an $(n \times m)$-thread CUDA grid. The kernel operates on 16×16 filter blocks. This is a minor deviation from the original AV1 CDEF implementation that was made to account for limitations on the number of threads that can be assigned to a CUDA block on the test device. Each thread operates on pixel-level granularity, processing only the pixel that was assigned to it. Hence, the implementation of this kernel avoids race conditions.

• *N. Khadija and W. Song are with the Department of Center for Global & Online Education, Stanford University, Stanford, CA, 94305. E-mail: nsahraou@stanford.edu, wsong2@stanford.edu*

The purpose of converting from RGB to YCbCr is to take advantage of the visual independence of the luma and chroma attributes of an image. The filter normally targets luma and chroma with different parameters, referred to as **presets** in the original paper. The AV1 CDEF is adaptively optimized to select distortion-minimizing presets for luma and chroma separately. Our implementation, however, filters only across luma and assumes the same preset for chroma. The reason for this is to reduce complexity while prioritizing the optimal selection of filter taps for luma over chroma, of which the human eye can tolerate more distortion.

Conversion from YCbCr representation to RGB takes place in the same manner after the image is filtered. The process is identical to the above description, so it has been omitted from the discussion.

3 Stage 2: Direction Search

The direction search is implemented as a CUDA kernel that assigns each thread to a block sample containing 8×8 pixels. The kernel also assigns CUDA blocks with thread dimensions 8×8. Hence, each block spans 64×64 pixels, aligning with the AV1 standard. As in stage 1, the threads operate on independent regions of memory so as to avoid the need to synchronize, thereby improving performance. Furthermore, an 8×8 block of pixels is loaded into shared memory for local access by each thread within the filter block. 64×64 pixels of memory are allocated per block.

The direction search operates on each 8×8 block of pixels by computing the optimal direction d_{opt} that minimizes the distance between the current block and the nearest "perfectly directional block". The process is illustrated in Figure 1.

The optimal direction d_{opt} is found by maximizing

$$d_{opt} = \arg\max_{d} \sum_{k} \frac{1}{N_{d,k}} \left(\sum_{p \in P_{d,k}} x_p \right)^2 . \quad (1)$$

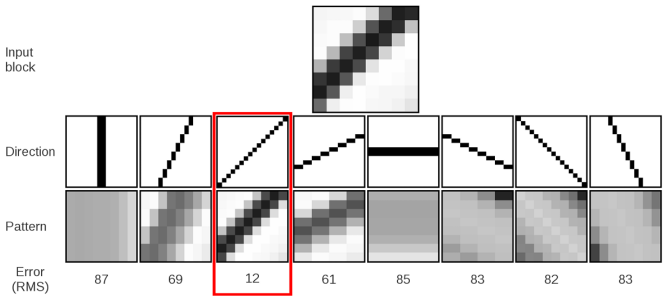

Fig. 1: Directional search in practice

where x_p is the value of pixel p, $P_{d,k}$ is the set of pixels in line k following direction d, and $N_{d,k}$ is the cardinality of $P_{d,k}$. This procedure is performed on a per-thread basis. Hence, each thread identifies the value of d_{opt} that maximizes the above expression for the 8×8 group of pixels assigned to it.

4 Stage 3: Non-linear Low-pass Filter

4.1 Directional Filter

To preserve directional edges and patterns, **primary taps** filter in the direction d_{opt}, while **secondary taps** filter 45° off the direction d_{opt}. The primary and secondary taps are given in Figures 2 and 3 respectively. For even **filter strengths**, the coefficients a and b equal 2 and 4 respectively. For odd filter strengths, $a = 3$ and $b = 3$. Rather than determine optimal strength presets, as is discussed in the original AV1 CDEF paper, this implementation chooses between even and odd presets depending on a metric described in section 6, thereby configuring a and b accordingly.

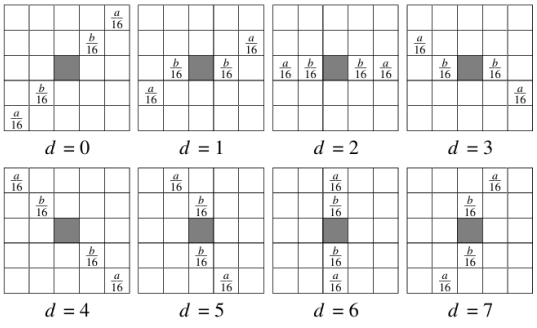

Fig. 2: Primary filter taps

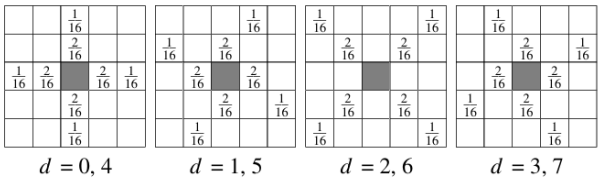

Fig. 3: Secondary filter taps

4.2 Convolution

The non-linear filter is implemented as a CUDA kernel. The grid is sectioned into 16×16 pixel filter blocks, and each thread is assigned to a single pixel. Hence, each thread at pixel (x, y) is responsible for evaluating the new value at (x, y) based on the expression

$$y(i,j) = x(i,j) + \sum_{m,n} w^{(p)}_{d,m,n} f(x(m,n) - x(i,j), S^{(p)}, D) + \sum_{m,n} w^{(s)}_{d,m,n} f(x(m,n) - x(i,j), S^{(s)}, D). \quad (2)$$

where $w^{(p)}_{d,m,n}$ is the **primary** weight for direction d of the pixel at position (m, n), $w^{(s)}_{d,m,n}$ is the **secondary** weight, and $f(x(m, n)$ is a constraint function given by the following:

$$f(d, S, D) = \min(d, \max(0, S - \lfloor d/2^{D - \log_2 S} \rfloor)). \quad (3)$$

The CUDA implementation aims to reduce the computational complexity of the convolution with the filter due to VRAM limitations of the test device. Hence, it notes that $f(d, S, D) = d$ for small values of d, which remains true for a majority of scenes in practice. Similarly, $f(d, S, D) = 0$ for large values of d. This compromise is appropriate for processing scenes containing objects of near-solid color and sharp edges, such that $f(d, S, D)$ swings between d and 0.

The GPU uses d_{opt} to index into primary and secondary filter tap LUTs and retrieve $w^{(p)}_{d,m,n}$ and $w^{(s)}_{d,m,n}$. The convolutional kernels are of size $k_{dim} = 5$. Also, $(16 + k_{dim} - 1) \times (16 + k_{dim} - 1)$ pixels of shared memory are allocated to each block. Similarly, $(16 + k_{dim} - 1) \times (16 + k_{dim} - 1)$ threads are declared for each block. The purpose of this is to retrieve pixels beyond the boundary of the filter block that are needed to perform convolution on boundary-resident pixels.

This portion of the implementation is prone to race conditions, as threads are dependent on regions of shared memory that have been written to by other threads. Hence, threads are synchronized to ensure correct behavior.

5 Encoder Search

The encoder determines optimal filter presets by minimizing a sum of squared errors (SSE) distortion metric

$$D_{frame} = \|s - d\|^2. \quad (4)$$

where s denotes a vector containing the source pixels and d denotes a vector containing the filtered pixels. The value of d will differ depending on whether an even preset or odd preset is used in convolution, as described in section 5.1. Accordingly, the final image is set to the result that minimizes SSE. Furthermore, this evaluation is made only across luma entries. Chroma values are forced to adopt the luma presets for the reason described in section 3. This is a simplification of the original AV1 CDEF implementation, which determines the optimal strength preset for each filter block out of 128 possibilities for luma and chroma separately.

This computation is performed by the host rather than the GPU. Section 8 explores parallel implementations that were not pursued due to time constraints.

6 Experimental Results

This implementation of the AV1 CDEF was evaluated qualitatively with various test images containing ringing artifacts. As shown in Figure 4, the filtered result contains less ringing. The difference is significant to the human eye, and can be improved with optimal filter preset selection that aligns with the original AV1 CDEF implementation.

The program itself also executes quickly. Although there was no serialized model available for performance comparison, the rapid execution of a computationally-heavy filtering procedure highlights the advantages of SIMD-based designs over serial designs.

Additionally, the direction search algorithm is effective in detecting natural edges. Figure 5 is a visual aid that loosely depicts the detected direction as a function of the pixel position. Directions are computed such that filtering is not perpendicular to the direction of a natural edge.

Fig. 4: Original (top) vs. Filtered Result (bottom)

Fig. 5: Camera Photo: Original (top) vs. Detected Direction as a Function of the Pixel Position (bottom)

7 Discussion

The visual performance of the filter can be improved by implementing the AV1 CDEF architecture in its entirety. The full computation of the constraint function f, for example, can be included without incurring performance losses by passing the values of f for each d, S, and D to the GPU in the form of a 3D LUT.

Likewise, filter presets can be adaptively improved with each frame, assuming a degree of similarity between frames. The first frame may evaluate a subset of presets that result in the lowest distortion based on the SSE metric. These presets may be passed onto the subsequent frame to reduce the amount of computation required to select optimal filter presets. With each frame, the presets will gradually converge to an optimal subset of presets across all possibilities. The complexity of this design is beyond the scope of this project, but may be explored in the future.

Lastly, distortion values can be processed for all filter blocks in parallel provided that threads are carefully synchronized with mutex locks to circumvent race conditions resulting from per-pixel distortion accumulation.

8 Conclusion

The AV1 CDEF is a filter that can be implemented easily in a parallel computing framework, such as CUDA, for enhanced performance. This is crucial for the real-time display of consecutive frames with channel compression. The simplified implementation described in this paper was successful in eliminating ringing artifacts in a short amount of time. The modifications described in section 8 can be included in future work to enhance image quality.

Acknowledgments

The authors would like to thank the teaching staff of CS248A for their dedication throughout the course. Their support and expertise greatly contributed to our learning experience and the completion of this work.

References

[1] S. Midtskogen and N. Egge, "The constrained directional enhancement filter," in *Proceedings of the IEEE International Conference on Image Processing (ICIP)*. IEEE, 2017, pp. 3615–3619.