PARALLEL GPU-BASED ALGORITHMS FOR IMAGE PROCESSING

A PREPRINT

Anonymous Author(s)

November 18, 2025

ABSTRACT

Image processing has become increasingly computationally demanding as image resolutions grow and real-time processing requirements intensify. Graphics Processing Units (GPUs), with their massively parallel architectures containing millions of threads, offer a perfect solution to these performance challenges. This project report presents GPU-based parallel implementations of fundamental image processing algorithms to address the computational demands and provide an introduction to high-resolution image processing. We implement and evaluate three core algorithms: Box blur, Gaussian blur, and Sobel edge detection, with and without optimizations. Our results demonstrate how parallel computing paradigms can effectively accelerate convolution-based filtering and explore manual, hard-wired optimizations for certain parallel algorithms, making real-time image processing feasible for high-resolution imagery.

1 Introduction

Image processing has become a cornerstone of modern computational applications, from medical diagnostics and autonomous navigation to multimedia systems and scientific visualization. As image resolutions continue to escalate up to 4K and 8K, the computational burden of processing millions of pixels in real-time has intensified dramatically Zamfir et al. [2023]. Traditional sequential algorithms executed on Central Processing Units (CPUs) face fundamental limitations in meeting these performance demands, necessitating a paradigm shift toward parallel computing architectures Sanders and Kandrot [2010].

Graphics Processing Units (GPUs) have emerged as accelerators for image processing tasks due to their massively parallel architecture. Modern GPUs contain thousands of processing cores capable of executing operations simultaneously, making them ideally suited to the data-parallel nature of image processing, where identical operations are applied independently across pixel arrays. The adoption of General-Purpose computing on GPUs (GPGPU) through frameworks such as CUDA and OpenCL has democratized access to this computational power, enabling speedups of 10x to 100x over CPU implementations for suitable workloads Podlozhnyuk [2007], Ryoo et al. [2008].

Convolution-based filtering operations mei W. Hwu et al. [2023], including blur filters and edge detection algorithms, represent fundamental primitives in image processing pipelines. Box blur and Gaussian blur serve as essential smoothing operations for noise. At the same time, on feature extraction Gonzalez and Woods [2018], edge detection algorithms like Sobel operators enable boundary identification, which is critical for computer vision tasks Canny [1986]. The computational intensity of these operations, particularly for large kernel sizes, makes them prime candidates for GPU acceleration. Previous work has demonstrated substantial performance gains through GPU implementations of convolution operations, with careful attention to memory access patterns and data locality proving crucial for optimization.

This paper presents parallel GPU implementations of three fundamental image processing algorithms: Box blur, Gaussian blur, and Sobel edge detection. We explore both naive and optimized implementations of Gaussian blur to illustrate the impact of GPU-specific optimization techniques, including memory coalescing, shared memory utilization, and reduction of global memory transactions Harris [2007], Micikevicius [2009]. Our work demonstrates practical strategies for exploiting GPU parallelism in image processing contexts and provides performance analysis highlighting the benefits and challenges of GPU acceleration for these canonical algorithms.

2 Project description

The project focuses on implementing parallel GPU-based algorithms for fundamental image processing operations. The primary objective is to implement these operations on the parallel architecture of modern Graphics Processing Units to accelerate computationally intensive image processing tasks that are necessary in computer vision, graphics, and multimedia applications. Image processing algorithms are data-parallel, as they typically apply identical operations to each pixel or local neighborhood independently, making them ideal candidates for GPU acceleration. By exploiting the thousands of processing cores available in contemporary GPUs, we aim to achieve substantial performance improvements over traditional CPU-based implementations, enabling the processing of high-resolution imagery.

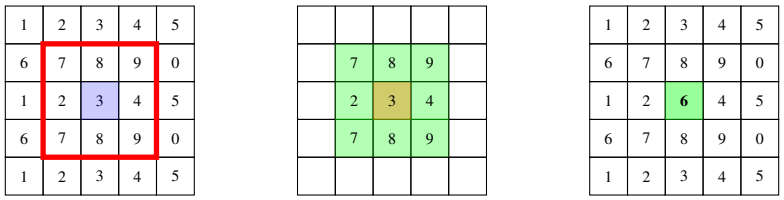
We developed and optimized GPU implementations of three core algorithms that represent different computational patterns in image processing. First, Box blur implements a simple averaging filter that provides efficient smoothing by computing the mean of pixels within a rectangular window. For example, Gaussian blur applies a weighted averaging operation using a Gaussian kernel, producing higher-quality smoothing that better preserves edges compared to box blur. For Gaussian blur, we implemented both naive and optimized versions to illustrate the impact of GPU-specific optimization strategies, including memory coalescing, shared memory utilization, and reduction of redundant global memory accesses. Third, Sobel edge detection combines convolution operations with gradient computation to identify boundaries and features within images. Through performance analysis and comparison of these implementations, this work demonstrates practical approaches to exploiting GPU parallelism and provides insights into the optimization techniques that are critical for achieving maximum performance in GPU-accelerated image processing applications.

3 Fundamental Image Processing Algorithms

This section introduces three fundamental algorithms that are widely used in computer vision pipelines: box blur, Gaussian blur, and Sobel edge detection. These operators form the building blocks for more sophisticated image analysis techniques.

3.1 Box Blur

Box blur, also known as mean filtering, is one of the simplest image smoothing techniques. It operates by replacing each pixel with the average value of pixels in a rectangular neighborhood around it. This process effectively reduces high-frequency noise while preserving overall image structure. Figure 1 shows the concept of Box Blur algorithm, basically summing all the value in the box, then take the average by dividing the size of the box.



(a) Original (b) Calculation (c) Result

Figure 1: Box blur convolution process: (a) Original image with 3x3 window (red frame) centered at pixel with value 3 (blue shaded), (b) Computing the floor of the floor of mean of all 9 pixels within the window (7, 8, 9, 2, 3, 4, 7, 8, 9), yielding 6, (c) Writing the computed value to the center pixel position in the output image.

Mathematical Formulation For a discrete image $I(x, y)$, the box blur operation with kernel size $(2r + 1) \times (2r + 1)$ is defined as:

$$I_{box}(x, y) = \frac{1}{(2r + 1)^2} \sum_{i=-r}^{r} \sum_{j=-r}^{r} I(x + i, y + j) \tag{1}$$

where r determines the radius of the neighborhood. The box blur kernel K_{box} is a uniform matrix:

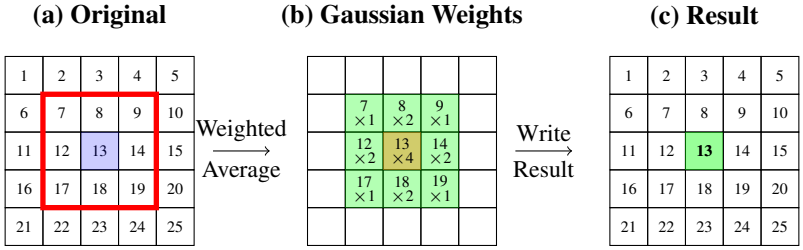
$$K_{box} = \frac{1}{(2r + 1)^2} \begin{bmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \cdots & 1 \end{bmatrix}_{(2r+1)\times(2r+1)} \tag{2}$$

Properties and Applications Box blur exhibits several notable characteristics: (1) computational efficiency: The uniform weighting allows for optimization through separable convolution, reducing complexity from $O(r^2)$ to $O(r)$ per pixel; (2) linear and shift-invariant: As a convolution operation, it satisfies the superposition principle; (3) averaging artifacts: The uniform weighting can produce blocky artifacts and does not approximate ideal low-pass filtering as well as weighted approaches.

Box blur is commonly used for quick noise reduction, motion blur simulation, and as a preprocessing step in feature detection algorithms.

3.2 Gaussian Blur

Gaussian blur applies weighted averaging where weights decrease with distance from the center pixel according to a Gaussian distribution. This produces smoother results than box blur and better approximates ideal low-pass filtering in the frequency domain. Figure 2 shows the concept of Gaussian blur algorithm.



$$G = \frac{1}{16}(1 \cdot 7 + 2 \cdot 8 + 1 \cdot 9 + 2 \cdot 12 + 4 \cdot 13 + 2 \cdot 14 + 1 \cdot 17 + 2 \cdot 18 + 1 \cdot 19) = \frac{208}{16} = 13$$

Figure 2: Gaussian blur convolution process: (a) Original image with 3×3 window (red frame) centered at pixel 13 (blue shaded), (b) Computing the weighted average using Gaussian kernel weights (center weight 4, edge weights 2, corner weights 1, normalized by 16), (c) Writing the computed weighted average (13) to the center pixel position in the output image.

Mathematical Formulation The Gaussian blur kernel is based on the 2D Gaussian function:

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right) \tag{3}$$

where σ is the standard deviation controlling the spread of the Gaussian. The blurred image is obtained through convolution:

$$I_{gauss}(x, y) = (I * G)(x, y) = \sum_{i=-r}^{r} \sum_{j=-r}^{r} I(x + i, y + j) \cdot G(i, j, \sigma) \tag{4}$$

The kernel is typically normalized such that $\sum_{i,j} G(i, j, \sigma) = 1$ to preserve image brightness. The kernel size is usually chosen as $(6\sigma + 1) \times (6\sigma + 1)$ to capture approximately 99.7% of the Gaussian distribution.

Properties and Applications Key properties of Gaussian blur include: (1) separability: The 2D Gaussian can be decomposed into two 1D Gaussians: $G(x, y, \sigma) = G(x, \sigma) \cdot G(y, \sigma)$, enabling efficient implementation; (2) rotational symmetry: the kernel weights are isotropic, providing uniform smoothing in all directions; (3) scale-space property: successive Gaussian blurs with variance σ_1^2 and σ_2^2 are equivalent to a single blur with variance $\sigma_1^2 + \sigma_2^2$; (4) optimal smoothness: among linear filters, Gaussian blur provides optimal localization in both spatial and frequency domains (uncertainty principle).

Gaussian blur is fundamental in scale-space analysis, serves as a preprocessing step in edge detection (e.g., Canny edge detector), and is widely used in feature extraction algorithms such as SIFT and image pyramids for multi-scale processing.

3.3 Sobel Edge Detection

The Sobel operator is a discrete differentiation operator that computes an approximation of the gradient of image intensity, highlighting regions of rapid intensity change that typically correspond to edges. Figure 3 shows the concept of Sobel Edge Detection.

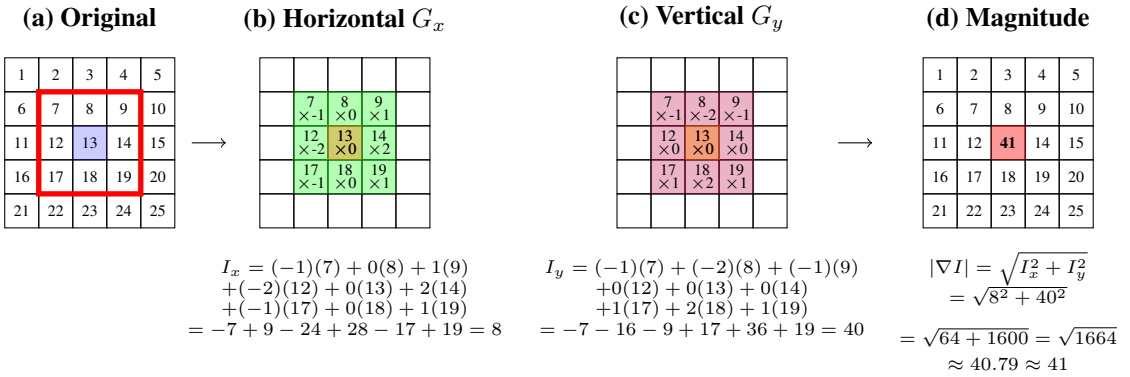


Figure 3: Sobel edge detection process: (a) Original image with 3×3 window (red frame) centered at pixel 13 (blue shaded), (b) Computing horizontal gradient I_x using G_x kernel, yielding 8, (c) Computing vertical gradient I_y using G_y kernel, yielding 40, (d) Computing gradient magnitude, indicating edge strength at the center pixel.

Mathematical Formulation The Sobel operator consists of two 3×3 kernels that estimate gradients in the horizontal and vertical directions:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \tag{5}$$

These kernels are convolved with the image to produce gradient approximations:

$$I_x(x, y) = G_x * I(x, y), \quad I_y(x, y) = G_y * I(x, y) \tag{6}$$

The gradient magnitude and direction at each pixel are computed as:

$$|\nabla I|(x, y) = \sqrt{I_x^2(x, y) + I_y^2(x, y)} \tag{7}$$

$$\theta(x, y) = \arctan\left(\frac{I_y(x, y)}{I_x(x, y)}\right) \tag{8}$$

For computational efficiency, the gradient magnitude is sometimes approximated as $|I_x| + |I_y|$.

Properties and Applications Main application of this algorithm is edge detection, which in futher analysis allows to perform structure analysis, which is included in tons of modern application, like computer vision, including autonomous driving for lane and obstacle detection, medical imaging to find tumors or organs, and robotics to help robots navigate and identify objects

3.4 Comparative Analysis

The three algorithms serve complementary roles in image processing pipelines. Box blur and Gaussian blur are both smoothing operations but differ in quality and computational cost. Gaussian blur produces superior visual results with better frequency characteristics at the expense of slightly higher computation. Sobel edge detection operates orthogonally to smoothing, enhancing high-frequency content to reveal structural boundaries. In practice, these algorithms are often combined: Gaussian smoothing reduces noise before edge detection, yielding more robust feature extraction as exemplified in the Canny edge detection algorithm.

4 Design and Implementation

The main architectural and design problem was to implement the classes and codebase in such way, that it would be simple to add algorithms for both sides, host (CPU) and device (GPU). The image filtering application follows a layered architecture with clear separation of concerns, dividing functionality into distinct modules: `view`, `image`, `pixel`, and core filtering logic. This design emerged from careful consideration of maintainability, testability, and extensibility requirements. After monotonous iterations and code refinements, we implemented the code in such way, that we needed approximately 100 lines of code (LOCs) to add the Sobel Edge detection algorithm, 170 and 110 LOCs to add the Gaussian algorithm with and without optimization, and 130 LOCs to add box blurring. Comprehensive class diagram for whole project illustrated in the Figure 4.

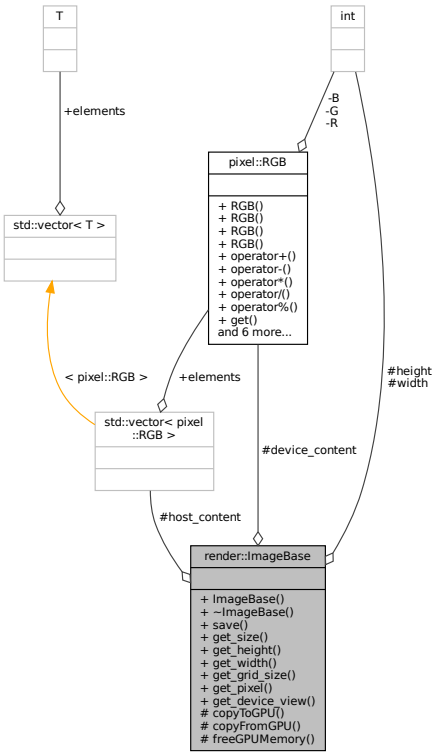


Figure 4: Comprehensive project structure with class diagram

The pixel module encapsulates low-level color representation and manipulation. By isolating pixel operations in their own module, we provide a single point of truth for pixel data structures and enable easy modification of color space representation (RGB, HSV, etc.). This separation simplifies unit testing of color operations and allows for future extensions such as alpha channel support or HDR capabilities without affecting higher-level code.

The image module manages 2D pixel arrays and image-level operations. This abstraction layer hides the underlying data structure implementation (whether using 2D arrays, 1D arrays with paddings, or other representations) from the rest of the application. The module provides boundary checking and safe pixel access, centralizes image I/O operations for loading and saving files, and crucially separates business logic from presentation concerns.

The view module handles all user interface and user interaction aspects. Following MVC (Model-View-Controller) pattern principles Necula [2024]. In this case, the *viewer* or *user* is GPU itself, and the code allows "viewing" the image object without making an extra copy. From the beginning until the end, the data is always stored inside GPU-oriented memory and retrieved at the moment of image writing to the actual file. Since this is the bottleneck of the project, we will not include the time of R/W operation,s as well as operations of memory initialization, and so on.

The filters themselves could be implemented as classes or just class functions. Since we don't want to overcomplicate the solution, which is sophisticated in itself, we decided to implement algorithms as class functions.

4.1 Alternative designs considered

A monolithic design was considered where all functionality would reside in a single module or file. While this approach offers simplicity for very small projects with minimal overhead, it suffers from being difficult to maintain and test as the project grows. The high coupling between components and the inability to reuse individual pieces made this approach unsuitable for anything beyond trivial implementations. That would break the Don't Repeat Yourself (DRY) pattern, whose main principle is to use existing code as much as possible.

A functional or procedural approach using standalone functions instead of classes was also evaluated. This method would be simpler for stateless operations and involve less boilerplate code. However, it becomes harder to maintain shared state, such as filter parameters and GPU buffer handles, offers limited extensibility, and risks namespace pollution. The object-oriented approach was ultimately chosen because it better supports complex filter configurations, GPU resource management, and inheritance patterns. Furthermore, it would perfectly align with the main design ideas and modern programming patterns.

The image and view classes were in the same class for a long time, since at first we decided to implement a prototype, and consider the design changes later. After observing repetitive code in some places, we decided to dedicate a separate class to the device side. This would eliminate the need for explicit conversions between CPU and GPU representations and reduce code duplication. The version with the shared class would make it impossible to optimize GPU memory layouts independently from CPU data structures. Different filters may require different GPU memory arrangements (row-major vs column-major, packed vs padded, different texture formats), and combining these concerns would force

compromises that hurt performance. Additionally, it would complicate testing since GPU operations would always be required even for simple CPU-side image manipulation.

Embedding pixel operations directly in both the image and view modules was another option. While this reduces the number of modules and provides tighter integration, it prevents reusing pixel operations independently and makes changing the color representation more difficult. It also leads to code duplication between CPU and GPU pixel handling, making maintenance harder when color space changes are needed.

5 Algorithms implementation

We decided to implement two parts for blurring algorithms: Optimized and Unoptimized, in order to show the significance of applying algorithm optimization to your application.

5.1 Box blurring

Box blur is the simplest convolution-based blur filter, applying a uniform averaging kernel to each pixel. For a given blur size, each output pixel is computed as the average of all pixels within a square window centered on that pixel. The algorithm replaces each pixel value with the mean of its neighbors within the specified radius.

Naive Implementation The unoptimized box blur implementation uses a straightforward two-dimensional approach where each thread computes one output pixel by iterating through the entire blur window. For each pixel, the thread reads all pixels within the square region defined by the blur radius, accumulates their RGB values, and divides by the total number of samples to produce the averaged result. This direct implementation is conceptually simple but suffers from significant performance limitations.

The work complexity of such an algorithm is $\mathcal{W}(w * h)$, where w is the image width and h is the image height. Time complexity is $\mathcal{O}(r^2)$, where r is the size blur radius. Each pixel (thread) requires processing a window of size proportional to r^2, leading to quadratic growth in computation time as blur radius increases.

Optimized Implementation The optimized box blur implementation uses an integral image technique Ehsan et al. [2015], also known as a summed area table, which enables constant-time computation of rectangular region sums regardless of box size. This is a dramatic improvement over the naive approach, especially for large blur radii where the quadratic cost becomes prohibitive.

An integral image is a data structure where each pixel contains the sum of all pixels above and to the left of it in the original image. Once constructed, any rectangular sum can be computed using exactly four array lookups and three arithmetic operations, regardless of the rectangle's size. For a rectangle with corners at (x_1, y_1) and (x_2, y_2), the sum is: $integral[x_2, y_2]$ - $integral[x_1 - 1, y_2]$ - $integral[x_2, y_1 - 1]$ + $integral[x_1 - 1, y_1 - 1]$.

The implementation proceeds in three distinct phases:

1. (Row prefix sum): h threads, one per row. Each thread reads w pixels and writes w accumulated values. Total operations: $h * w$ reads and writes.
2. (Column prefix sum): w threads, one per column. Each thread reads H values from the row-summed data and writes h accumulated values. Total operations: $w * h$ reads and writes.
3. (Blur using integral): $w * h$ threads, one per output pixel. Each thread reads exactly 4 values from the integral image regardless of blur radius. Total operations: $w * h * 4$ reads.

First two passes require h and w threads, respectively. The last pass requires $w * h$ threads, so total work complexity is $\mathcal{W}(w * h)$. Meanwhile, time complexities are following: $\mathcal{O}(log(w))$ for first pass (compute prefix sum for each row in parallel), $\mathcal{O}(log(h))$ for second pass (compute prefix sum for each column in parallel) and $\mathcal{O}(1)$ for last pass, so the total complexity is: $\mathcal{O}(\mathbf{log(w) + log(h)})$, which is significant upgrade from $\mathcal{O}(r^2)$. This way, we require constant time for any window size, and our time complexity fully depends on the image size itself.

5.2 Gaussian blurring

Gaussian blur produces a higher quality blur than box blur by using a weighted average based on the Gaussian distribution. Pixels closer to the center of the kernel contribute more to the output than pixels at the edges, creating a more natural-looking blur effect. The Gaussian function ensures smooth falloff and better preservation of edge characteristics compared to uniform averaging.

Naive Implementation The unoptimized Gaussian blur implements a straightforward two-dimensional convolution. Each thread computes one output pixel by iterating through the entire blur kernel in both horizontal and vertical dimensions simultaneously. For each position in the kernel, the thread reads the corresponding input pixel, multiplies it by the precomputed Gaussian weight, and accumulates the result.

The performance characteristics mirror those of unoptimized box blur, with massive redundant global memory access. Multiple threads read the same input pixels repeatedly because neighboring output pixels have overlapping blur windows. Additionally, the two-dimensional traversal pattern exhibits poor memory access locality, as consecutive threads may access memory locations that are far apart, leading to cache misses and memory bandwidth saturation.

Launching $w * h$ threads for an image of dimensions $w * h$, where each thread processes one output pixel, so the work complexity is $\mathcal{W}(w * h)$. Each thread reads r^2 input pixels and performs r^2 multiply-accumulate operations with Gaussian weights, so the time complexity is $\mathcal{O}(r^2)$

Optimized implementation The optimized Gaussian blur leverages the separability property of the Gaussian kernel, which states that a two-dimensional Gaussian convolution can be decomposed into two sequential one-dimensional convolutions (horizontal followed by vertical). This mathematical property reduces computational complexity from quadratic to linear in kernel size.

The implementation uses a two-pass approach: (1) applying of horizontal Gaussian blur, storing results in a temporary buffer; (2) reading from this temporary buffer and applying a vertical Gaussian blur to produce the final output. Each pass is significantly faster than the unoptimized version because threads only traverse a single dimension, improving memory access patterns and cache utilization.

Since we still require launch threads for each pixel, our work complexity is still $\mathcal{W}(w * h)$. However, due to two-pass approach, our algorithm significantly reduces time complexity and removes quadratic time complexity. First pass: $\mathcal{O}(r)$, second pass: $\mathcal{O}(r)$, so the total time complexity is $\mathcal{O}(r + r) = \mathcal{O}(r)$

5.3 Sobel edge detection

Sobel edge detection identifies edges in images by computing gradient approximations using specialized convolution kernels. The algorithm applies two 3x3 kernels (one for horizontal gradients, one for vertical gradients) to detect changes in intensity. The horizontal Sobel kernel emphasizes vertical edges, while the vertical kernel emphasizes horizontal edges. The final edge magnitude is computed as the Euclidean distance of the gradient vector.

Since the Sobel edge detection algorithm always requires constant 3x3 kernels, we should not "optimize" them in any way, since the algorithm requires at most 9 accesses for each pixel, which is theoretically constant time.

The work complexity for this algorithm is the same as the work complexity for many image processing algorithms, which is $\mathcal{W}(w * h)$. Since each thread requires accessing 9 elements, the time complexity is $\mathcal{O}(9) = \mathcal{O}(1)$.

6 Evaluation

The main research question of this project is to assess how differs naive solution of basic image-processing algorithms differs from the hand-optimized solution.

6.1 Workload and experiment setup

We decided to evaluate three images of different sizes. The evaluation was done using the well-known "Lenna Forsen" picture of two different sizes and a custom 4k picture in order to stress-test algorithms as well. Sample images are illustrated in Figure 5.

As it was mentioned before, we will not include any R/W instructions or initialization in the evaluation section. All recordings are done by using the `cudaEventCreate` function with CUDA events, widely used for measuring GPU computation time Al-Turany and Uhlig [2011].

Experiment setup All experiments are conducted on NVIDIA GeForce RTX 3050 Ti Mobile 4 GiB, Ubuntu 22.04, AMD Ryzen 5 5600H @ 4.280G, 16 GB RAM.

6.2 Evaluation results

Unoptimized and optimized algorithm, as expected, differs by runtime. For the Box filter, the integral image approach shows dramatic improvements for large kernel sizes, up to 465x time optimization for a 500x500 kernel on a 4284x4284 image. For small kernels (10x10), performance is comparable since the overhead of computing the integral image dominates. The optimized version maintains nearly constant execution time regardless of kernel size, confirming the $\mathcal{O}(log(w) + log(h)$ per-pixel lookup characteristic.

Separable Gaussian blur demonstrates significant speedup, though less dramatic than the Box filter. At a 500x500 kernel on a 4284x4284 image, the optimized version (279 *ms*) is 169x faster than the unoptimized (47193 *ms*). The speedup ratio increases with kernel size, validating the $\mathcal{O}(r)$. vs $\mathcal{O}(r^2)$ time complexity advantage. Even at moderate kernel sizes (100x100), the optimized version shows 16x improvement.

All algorithms scale approximately linearly with image area, as expected from O(W × H) work complexity. Sobel edge detection shows clean linear scaling: 2 *ms* (512x512) $\rightarrow$ 17 *ms* (1960x1960) $\rightarrow$ 81 *ms* (4284×4284), matching the 64x increase in pixel count from smallest to largest image.

After manually evaluating images obtained using optimized algorithms and without them, we saw that the content was different, but this difference was not visible to the naked eye, and with careful inspection as well. The pixel value differs from 1 to 3 from the unoptimized solutions, which is considered reasonable (< 0.5%). The pixel difference is caused by different calculation methods, especially with the Box blurring algorithm.

Images of all algorithms illustrated in the appendix, Figures: 6, 7, 8, 9, 10, 11,, 12.

7 Conclusion

This work demonstrates the significant performance benefits achievable through careful optimization of GPU-based image filtering algorithms. The implementation of box blur, Gaussian blur, and Sobel edge detection using CUDA reveals that algorithmic choices profoundly impact execution time, particularly for operations with large kernel sizes. The integral image approach for box filtering achieves up to 465x speedup over naive convolution for large kernels, maintaining constant-time complexity regardless of kernel size. Similarly, exploiting the separability property of Gaussian kernels reduces time complexity from $\mathcal{O}(r^2)$ to $\mathcal{O}(r)$, yielding 169x speedup for 500x500 kernels on high-resolution images.

The modular architecture separating pixel, image, and view concerns provides clear boundaries between CPU and GPU memory representations while maintaining code maintainability and extensibility. This separation facilitates independent optimization of GPU memory layouts and enables straightforward addition of new filters without modifying existing components. The distinction between work complexity and time complexity in parallel algorithms proves crucial for understanding GPU performance characteristics, where memory access patterns and data reuse opportunities often dominate raw computational costs.

Performance evaluation across multiple image resolutions and kernel sizes confirms theoretical predictions, showing that optimized algorithms scale favorably with both image dimensions and kernel size. The results demonstrate that careful consideration of GPU memory hierarchy, particularly the strategic use of shared memory and reduction of global

(a) 512×512 (b) 1960×1960 (c) 4284×4284

Figure 5: Sample images used for performance testing

Algorithm	**Kernel**	**Execution Time** *(ms)*		
		512×512	1960×1960	4284×4284
Box Filter (UO)	10×10	3	21	108
	100×100	29	380	1780
	500×500	606	8914	44179
Box Filter (O)	10×10	3	22	95
	100×100	4	22	95
	500×500	4	23	95
Gaussian Filter (UO)	10×10	2	22	106
	100×100	33	427	2004
	500×500	629	9767	47193
Gaussian Filter (O)	10×10	2	19	89
	100×100	3	27	122
	500×500	5	61	279
Sobel Edge Detection	3	2	17	81

Table 1: Performance comparison of GPU-based image processing algorithms. (UO) stands for unoptimized, O - optimized. Execution time shows the CUDA runtime in milliseconds for different image sizes.

memory traffic, transforms memory-bound operations into compute-bound ones, allowing modern GPU architectures to achieve their full potential. These optimization techniques generalize beyond the specific filters implemented here, providing valuable insights for GPU-accelerated image processing applications.

References

Eduard Zamfir, Marcos Conde, and Radu Timofte. Towards real-time 4k image super-resolution. 06 2023. doi:10.1109/CVPRW59228.2023.00155.

Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 2010. URL `https://developer.nvidia.com/cuda-example`.

Victor Podlozhnyuk. Image convolution with cuda. Technical report, NVIDIA Corporation, 2007. URL `https://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/convolutionSeparable/doc/convolutionSeparable.pdf`.

Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 73–82, 2008. doi:10.1145/1345206.1345220. URL `https://doi.org/10.1145/1345206.1345220`.

Wen mei W. Hwu, David B. Kirk, and Izzat El Hajj. Chapter 7 - convolution: An introduction to constant memory and caching. In Wen mei W. Hwu, David B. Kirk, and Izzat El Hajj, editors, *Programming Massively Parallel Processors (Fourth Edition)*, pages 151–171. Morgan Kaufmann, fourth edition edition, 2023. ISBN 978-0-323-91231-0. doi:https://doi.org/10.1016/B978-0-323-91231-0.00008-2. URL `https://www.sciencedirect.com/science/article/pii/B9780323912310000082`.

Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. Pearson, 4th edition, 2018.

John Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(6):679–698, 1986. doi:10.1109/TPAMI.1986.4767851. URL `https://doi.org/10.1109/TPAMI.1986.4767851`.

Mark Harris. Optimizing parallel reduction in cuda. Technical report, NVIDIA Developer Technology, 2007. URL `https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf`.

Paulius Micikevicius. 3d finite difference computation on gpus using cuda. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 79–84, 2009. doi:10.1145/1513895.1513905. URL `https://doi.org/10.1145/1513895.1513905`.

Sabina Necula. Exploring the model-view-controller (mvc) architecture: A broad analysis of market and technological applications, 04 2024.

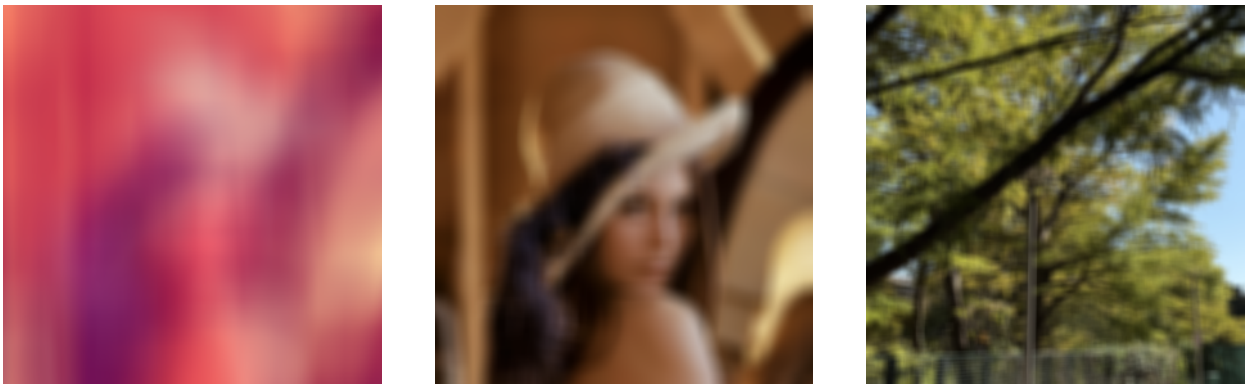
Shoaib Ehsan, Adrian F Clark, Naveed ur Rehman, and Klaus D McDonald-Maier. Integral images: Efficient algorithms for their computation and storage in resource-constrained embedded vision systems. *Sensors*, 15(7):16804–16830, 2015.

Mohammad Al-Turany and Florian Uhlig. Applying cuda computing model to event reconstruction software. page 014, 02 2011. doi:10.22323/1.093.0014.

8 Appendix

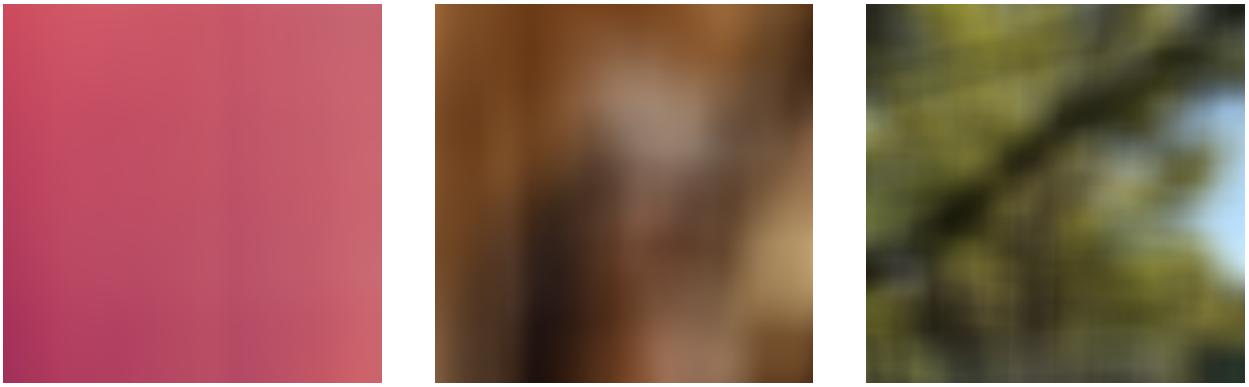
(a) 512×512 (b) 1960×1960 (c) 4K

Figure 6: Box blur results with 10×10 kernel applied to three test images



(a) 512×512 (b) 1960×1960 (c) 4K

Figure 7: Box blur results with 100×100 kernel applied to three test images



(a) 512×512 (b) 1960×1960 (c) 4K

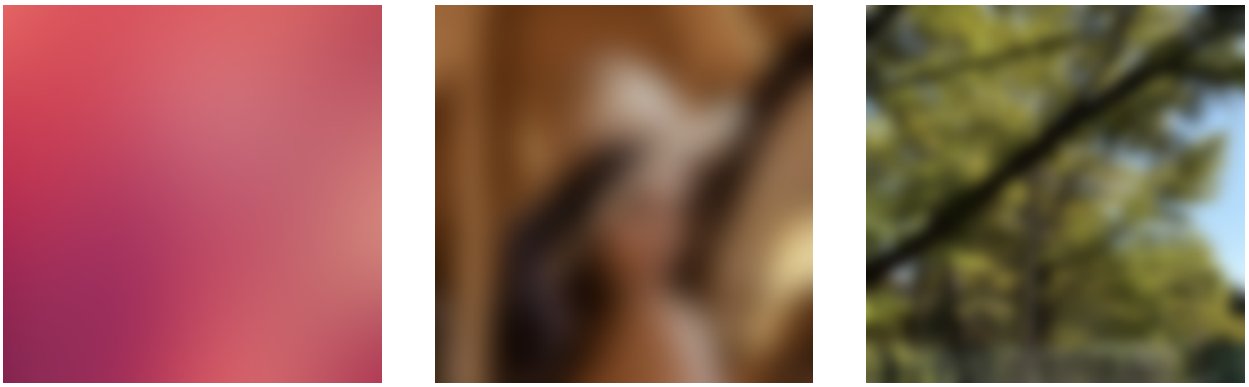
Figure 8: Box blur results with 500×500 kernel applied to three test images

(a) 512×512 (b) 1960×1960 (c) 4K

Figure 9: Gaussian blur results with 10×10 kernel applied to three test images

(a) 512×512 (b) 1960×1960 (c) 4K

Figure 10: Gaussian blur results with 100×100 kernel applied to three test images



(a) 512×512 (b) 1960×1960 (c) 4K

Figure 11: Gaussian blur results with 500×500 kernel applied to three test images

(a) 512×512 (b) 1960×1960 (c) 4K

Figure 12: Sobel edge detection results applied to three test images