HPC-Driven Computer Vision: Accelerating Image Processing Stencils and Pointwise Operations via NVIDIA CUDA

University of Camerino, MSc in Computer Science, Parallel and Distributed Programming

1st Daniele Monaldi
School of Science and Technologies
University of Camerino
Camerino, Italy
daniele.monaldi@studenti.unicam.it

2nd Edoardo Papa
School of Science and Technologies
University of Camerino
Camerino, Italy
edoardo.papa@studenti.unicam.it

3rd Alessio Rubicini
School of Science and Technologies
University of Camerino
Camerino, Italy
alessio.rubicini@studenti.unicam.it

***Abstract*—This report examines the optimization and parallelization of three fundamental image processing algorithms: Sobel edge detection, Gaussian blur, and RGB-to-YUV color space conversion. We developed a progression of implementations, starting from single-threaded C++ baselines and evolving into CUDA solutions. The core objective was to bridge the gap between theoretical peak performance and actual throughput. For stencil-based operations, we implemented shared memory tiling and constant caching to mitigate the global memory bottleneck, reducing redundant DRAM fetches by nearly an order of magnitude. For pointwise transformations, we focused on maximizing memory bus saturation through a planar output layout and hiding PCIe communication overhead via asynchronous CUDA streams. Our results demonstrate that while algorithmic improvements like separability provide a solid foundation, the highest gains come from managing the memory hierarchy. We measured end-to-end performance using metrics such as throughput (MPixels/s) and effective bandwidth (GB/s). The final optimized kernels achieved significant improvements over optimized CPU versions, reaching the physical limits of the PCIe bus and driver launch latency. This analysis highlights the critical trade-off between compute density and data movement in modern high-performance computing environments.**

I. Introduction

A. Project Overview

Image processing is a core component of many modern applications, and the increasing demand for high-resolution data and real-time performance makes sequential implementations increasingly inadequate. GPU acceleration, and in particular NVIDIA CUDA, provides an effective platform to address these challenges by exploiting massive data parallelism and high memory bandwidth.

The objective of this project is to develop a high-performance image processing library using CUDA, focusing on both pointwise and stencil-based operations commonly found in computer vision pipelines. Rather than relying on naive GPU implementations, the project emphasizes the use of memory-aware optimization techniques to fully leverage the GPU architecture. Selected filters, including color space conversion, Gaussian blur, and Sobel edge detection, are used as case studies to analyze different computational complexities and memory access patterns. The project also explores the impact of algorithmic optimizations, such as the use of separable Gaussian blur and domain decomposition strategies, and evaluates performance through systematic benchmarking and profiling.

B. Motivation

Image processing algorithms are particularly well-suited for GPU acceleration due to the inherently data-parallel nature of image data. Most operations apply the same computation independently to each pixel or to small local neighborhoods, making them a natural fit for the Single Instruction Multiple Threads (SIMT) execution model used by modern GPUs. In pointwise operations, such as color space conversion, each output pixel depends only on the corresponding input pixel. This results in embarrassingly parallel workloads with minimal data dependencies. Similarly, stencil-based operations like Gaussian blur and Sobel edge detection apply the same convolution pattern across the image, allowing thousands of GPU threads to process different pixels concurrently.

Compared to CPUs, GPUs provide a much higher degree of parallelism and memory bandwidth, which can significantly reduce execution time when memory access patterns are carefully optimized. However, fully exploiting this potential requires an explicit awareness of the GPU memory hierarchy and execution model. This project is motivated by the need to understand how algorithm structure, data layout, and memory access patterns interact with the SIMT model to achieve high performance in practice.

II. Background and Methodology

A. CUDA Architecture

The experimental evaluation in this project is performed on an NVIDIA Tesla T4 GPU, commonly available in cloud environments such as Google Colab. The T4 is based on the

NVIDIA Turing architecture and is designed to provide a balance between computational throughput and energy efficiency for general-purpose GPU workloads. At the core of the CUDA programming model is the Streaming Multiprocessor (SM), which serves as the main execution unit of the GPU. Each SM is capable of executing thousands of lightweight threads concurrently by organizing them into groups called warps, each consisting of 32 threads executing the same instruction within the SIMT paradigm [1]. A distinctive feature of the Turing architecture [2] is its unified L1 Cache/Shared Memory unit, which can be dynamically partitioned to prioritize either hardware-managed caching or developer-controlled local storage. Efficient utilization of these resources is essential for achieving high performance, as they directly affect occupancy, memory latency, and overall throughput.

B. Memory Hierarchy

The performance of CUDA applications is strongly influenced by the GPU memory hierarchy, which provides different types of memory with varying latency, bandwidth, and scope.

- **Global memory** is the largest memory space available on the GPU and is accessible by all threads. However, it also has the highest latency (often exceeding 400-600 clock cycles) and limited cache effectiveness when access patterns are irregular. Naive implementations that rely heavily on global memory accesses often become memory-bound and fail to fully use the computational capabilities of the GPU.
- **Shared memory** is an on-chip memory region shared among threads within the same block. It offers significantly lower latency and higher bandwidth compared to global memory, making it well-suited for stencil-based operations such as convolutions. By cooperatively loading image tiles into shared memory, threads can reuse data efficiently and drastically reduce the number of global memory accesses.
- **Constant memory** is a small, read-only memory space cached on-chip and optimized for broadcast access. It is particularly effective when all threads read the same values, as is the case for convolution kernels and filter coefficients. Using constant memory for such parameters reduces memory traffic and improves cache utilization.

In addition to on-device memory, data transfers between host and device can represent a significant performance overhead. To mitigate this, pinned (page-locked) host memory must be allocated to enable Direct Memory Access (DMA) transfers, allowing data to be copied directly between host and device without intermediate buffering. When combined with asynchronous transfers and CUDA streams, pinned memory enables the overlap of data movement and computation, further improving overall application throughput.

C. Evaluation Metrics

To evaluate the effectiveness of the proposed implementations, performance is measured using a set of quantitative metrics that capture both computational efficiency and memory behavior. These metrics are applied consistently across CPU and GPU versions to enable fair comparisons.

The primary metric is the average execution time, computed over multiple iterations to reduce the impact of noise and transient effects. For GPU versions, we explicitly distinguish between the pure 'Kernel Time' and the 'Total Pipeline Time,' the latter including the overhead of memory migrations. From execution time, throughput is computed as the number of processed pixels per second (MPixels/s). This metric is particularly useful for image processing workloads, as it provides a resolution-independent measure of performance and highlights scalability with respect to image size.

To assess memory efficiency, the effective memory bandwidth is estimated using the formula:

$$\text{Bandwidth (GB/s)} = \frac{\text{Bytes read} + \text{Bytes written}}{\text{Execution Time} \cdot 10^9}$$

Although this value does not represent peak hardware bandwidth, it provides insight into whether an algorithm is memory-bound and how efficiently it uses the memory subsystem.

Finally, speedup is used to compare optimized implementations against baseline versions, such as standard CPU or naive algorithms. Speedup values help quantify the impact of algorithmic optimizations, including separable convolutions and improved memory access patterns.

For CPU-based benchmarks, data transfer overheads between host and device are naturally absent. These measurements serve as a reference point for evaluating the benefits and trade-offs of GPU acceleration, which are analyzed in later sections.

III. Software Architecture

A. Framework Design

The project is structured around a modular C++ framework designed to support a clear separation between algorithmic logic and low-level hardware management. We chose this approach to enable seamless comparisons between CPU and GPU implementations without duplicating boilerplate code. The system uses the single-header libraries `stb_image.h` and `stb_image_write.h` [4] for I/O operations. These tools were selected for their minimal footprint, allowing us to focus on computational kernels rather than complex image decoding pipelines. The software follows a CLI-driven model where parameters such as kernel size (K), blur intensity (σ), and the number of iterations can be passed at runtime. This flexibility was essential during the benchmarking phase, as it allowed us to move through different image resolutions and filter configurations to identify performance turning points.

B. RAII and Memory Management

Managing memory in CUDA often leads to fragmented code due to the manual nature of `cudaMalloc` and `cudaFree`. To address this, we implemented the Resource

Acquisition Is Initialization (RAII) pattern via a custom `CudaMemoryManager` template class. This manager handles both device-side allocations and host-side pinned memory. By wrapping raw pointers in this class, we ensured that resources are automatically deallocated when the object goes out of scope, eliminating memory leaks during the iterative testing of large images. We chose to prioritize pinned memory (`cudaHostAlloc`) for all host-side buffers. While pinned memory consumes more system resources and can lead to performance degradation if over-allocated, it is a requirement for high-speed DMA transfers. In our project, the increased complexity of managing page-locked memory was a necessary trade-off to enable the asynchronous execution models used in the RGB-to-YUV module. During development, we noticed that using standard `std::vector` for large transfers introduced a significant hidden cost due to the driver internally copying data to a staging buffer before the actual PCIe transfer. Switching to the `CudaMemoryManager` with pinned support removed this intermediate step, resulting in more predictable H2D and D2H latencies.

C. *Error Handling Strategy*

Robustness in parallel programming is difficult to achieve because errors in asynchronous kernels often do not manifest until much later in the execution flow. Our strategy relies on the `gpuErrchk` macro, which wraps every CUDA API call and kernel launch. This macro performs an immediate check on the return status of the function and provides a detailed error string, the filename, and the line number upon failure.

For kernel launches, we followed each execution with `cudaPeekAtLastError` and a synchronization point during the debugging phase. We recognized that while constant synchronization allows for precise error localization, it introduces a significant performance penalty by stalling the pipeline. Consequently, for the final benchmarking builds, we limited explicit synchronization to the timing events, relying on the asynchronous error reporting for most of the processing. This balance ensured that the measured throughput reflects the real-world performance of the kernels rather than the overhead of the monitoring framework. A minor issue we encountered was the driver-side timeout for very long-running kernels (TDR). We mitigated this by ensuring our work-groups were sized correctly for the Tesla T4's scheduler.

IV. Image Processing Algorithms

A. *Sobel Edge Detection*

The Sobel Edge Detection algorithm is a widely used technique [3] in image processing for highlighting intensity variations and detecting edges. It is based on the computation of local image gradients and belongs to the class of stencil-based operations, where each output pixel depends on a small neighborhood of input pixels.

The sobel operator approximates the horizontal and vertical intensity gradients by convolving the input image with two fixed 3×3 kernels, typically denoted as G_x and G_y. The kernels emphasize intensity changes along the horizontal and vertical directions. For each pixel, the gradient magnitude is then computed by combining the two components, providing a measure of edge strength.

From a computational perspective, Sobel filtering exhibits regular memory access patterns but requires multiple reads per output pixel due to its stencil nature. Each pixel computation involves accessing neighboring pixels, which introduces data reuse opportunities but also makes the algorithm sensitive to memory latency if implemented naively. For GPU architectures, Sobel edge detection represents an ideal candidate to showcase the benefits of shared memory. By loading a tile of the image, including halo regions, into shared memory, multiple threads within a block can reuse the same data, significantly reducing redundant global memory accesses. This optimization is especially important for achieving high performance, as Sobel filtering is typically memory-bound rather than compute-bound.

B. *Gaussian Blur*

Gaussian blur is a fundamental image smoothing technique widely used in computer vision for noise reduction and scale-space analysis. It operates by convolving the input image with a Gaussian kernel, producing a weighted average of neighboring pixels that attenuates high-frequency components while preserving overall image structure.

In its standard formulation, Gaussian blur is implemented as a 2D convolution. For each output pixel, a weighted sum is computed over a $K \times K$ neighborhood, resulting in a computational complexity of $O(K^2)$ operations per pixels. While conceptually simple, this approach becomes increasingly expensive as the kernel size grows, both in terms of arithmetic operations and memory accesses.

A key property of the Gaussian kernel is its separability: a 2D Gaussian can be expressed as the product of two 1D Gaussians, one along the horizontal direction and one along the vertical direction. This allows the original 2D convolution to be decomposed into two successive 1D convolutions, first applied horizontally and then vertically. Exploiting this property reduces the computational complexity from $O(K^2)$ to $O(K)$ per pixel.

From a performance perspective, this reduction has significant implications. The separable formulation drastically lowers the number of arithmetic operations and memory accesses required for each output pixel. Moreover, the two-pass structure improves memory locality and reduces pressure on the memory subsystem, making the algorithm more efficient on both CPUs and GPUs.

In parallel architectures such as CUDA-enabled GPUs, Gaussian blur represents a representative stencil-based workload where performance is often limited by memory bandwidth rather than raw compute capability. For this reason, Gaussian blur, particularly in its separable form, is an ideal case study to analyze the interaction between algorithmic complexity, memory access patterns, and architectural optimizations.

C. Color Space Conversion (RGB-to-YUV)

Color space conversion from RGB to YUV is a classic pointwise image processing operation, where each output pixel depends solely on the corresponding input pixel. This characteristic makes the algorithm highly parallelizable and particularly well-suited for GPU execution under the SIMT model.

The conversion is defined by a linear transformation involving a fixed set of coefficients. While a straightforward implementation relies on floating-point arithmetic, this approach may introduce unnecessary overhead for a kernel that is primarily limited by memory throughput. To improve efficiency, the conversion can be implemented using fixed-point arithmetic, where coefficients are scaled to integers and normalization is performed through bit-shift operations. This reduces the number of floating-point instructions and register usage per thread, improving instruction throughput without compromising perceptual accuracy.

Memory layout proved to be the dominant factor for this kernel. A standard interleaved layout (RGBRGB...) prevents memory coalescing because threads in a warp access non-contiguous addresses. We transformed the data into a planar layout (YYYY... UUUU... VVVV...). This organization ensures that consecutive threads write to consecutive memory addresses. This change was the primary driver in saturating the memory bus, allowing the implementation to move closer to the theoretical peak bandwidth of the Tesla T4.

We recognize that while this specific layout maximizes kernel throughput, the overall benefit depends on the requirements of subsequent stages in an image processing pipeline. In our current implementation, this planar data is not further processed. However, this architectural choice provides a foundation for future vision tasks (such as neural network inference) where planar input is often a standard requirement.

V. Parallel Optimization Strategies

A. Shared Memory Tiling

The main performance limitation of the naive stencil implementations was the high ratio between global memory accesses and arithmetic operations. To address this issue, we implemented a tiling strategy that leverages the low-latency shared memory available on the Tesla T4. The image is partitioned into thread blocks, and threads within each block cooperatively load a data tile into a local `__shared__` array.

A critical aspect of this approach is the handling of the halo region. Since each output pixel in a convolution requires access to neighboring values, each block must load pixels beyond its nominal boundaries. We adopted a cooperative loading scheme in which threads are also responsible for loading the required halo pixels, applying coordinate clamping to manage boundary conditions. This design increases kernel complexity and introduces branch divergence during the loading phase. Nevertheless, it substantially reduces DRAM traffic. Each pixel is loaded from global memory once and reused by the stencil computations within the same block, reducing memory traffic and improving overall balance between memory access and computation. A `__syncthreads()` barrier is used to guarantee that all data are available in shared memory before the convolution phase begins, with the associated synchronization overhead accepted as necessary for correctness.

B. Constant Cache Exploitation

For both Sobel and Gaussian filters, the convolution masks and Gaussian weights are constant throughout kernel execution. Keeping these coefficients in global memory would result in repeated high-latency accesses by all threads. For this reason, the masks were moved to the `__constant__` memory space.

This choice takes advantage of the constant cache behavior on the Turing architecture. When all threads within a warp access the same constant memory location, the value is broadcast efficiently by the hardware. Although constant memory is limited to 64 KB, the filters used in this project require under 1 KB, making them suitable candidates. This change also reduced pressure on the L1 cache and slightly lowered register usage, which in turn improved warp residency on each SM.

C. Hiding Communication Latency

In the RGB-to-YUV conversion module, we observed that for simple pointwise kernels the PCIe transfer time can dominate overall execution time. To mitigate this communication overhead, we implemented a multi-stream pipeline based on asynchronous memory transfers. Synchronous `cudaMemcpy` calls were replaced with `cudaMemcpyAsync`, and the input image was divided into independent chunks.

Four CUDA streams were used to overlap Host-to-Device transfers, kernel execution, and Device-to-Host transfers on different chunks of the image. Pinned host memory was required to enable asynchronous DMA transfers. The benefit of this overlap depends on the chosen chunk size. When chunks are too small, the overhead introduced by stream scheduling becomes noticeable and limits performance. By selecting an appropriate chunk size, we reduced the overall execution time compared to a fully synchronous pipeline.

D. Domain Decomposition

To allow execution beyond a single GPU, the kernels were written using a simple domain decomposition scheme. Instead of assuming access to the full image, each kernel receives a `y_offset` and a `y_slice_height` parameter. On the host side, this makes it possible to split a high-resolution image (such as 4K or 8K) into horizontal slices that can be processed independently.

This design requires careful handling of global coordinates, especially for boundary conditions, so that clamping logic refers to the actual image limits rather than the local slice. All experiments in this work were performed on a single Tesla T4. However, the same structure can be extended to multi-GPU or MPI-based executions if needed. The only overhead introduced by this approach is a small increase in register usage to store the offset parameters.

VI. Results and Performance Analysis

A. Test Environment and Methodology

All benchmarks were run on a Google Colab instance equipped with an NVIDIA Tesla T4 GPU and an Intel Xeon CPU. To obtain reliable measurements, we first included a warm up run for each implementation, in order to avoid counting the one time overhead from CUDA context initialization and initial memory allocations. Average execution times were calculated over N iterations, specifically $N = 200$ for CPU and $N = 100$ for GPU kernels, in order to mitigate the impact of transient system noise. We isolated kernel performance using CUDA events for algorithmic analysis while measuring the full pipeline time to assess PCIe bus influence.

B. Pointwise and Stencil Scaling

The relationship between image resolution and execution time for pointwise and stencil operations remains strictly linear across our tests.

As shown in the plots for RGB-to-YUV scaling (see figure 1) and Sobel scaling (see figure 2) the execution time grows proportionally with the total number of pixels. This confirms that our GPU kernels effectively saturate the device's processing units as the workload increases.

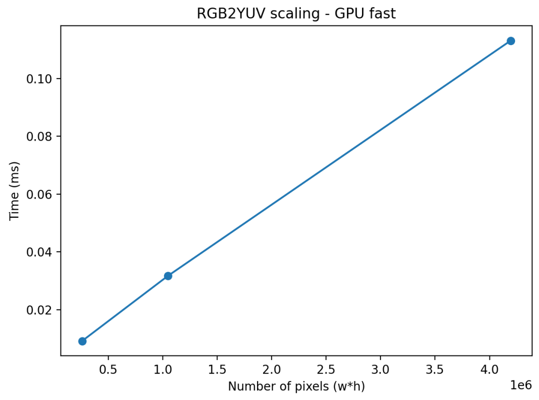

Fig. 1. RGB-to-YUV scaling

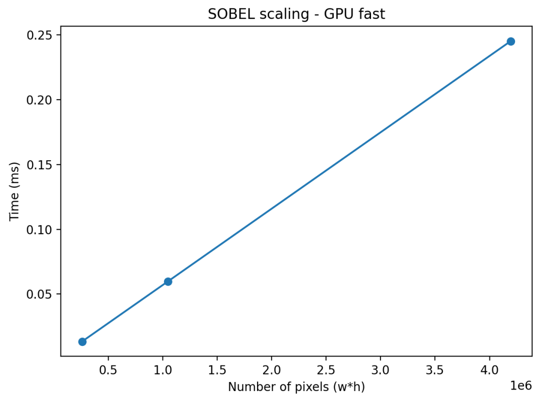

Fig. 2. Sobel scaling (GPU Fast)

Data for RGB-to-YUV conversion indicates that we reached a bandwidth limit early in the scaling process. For the 2048×2048 resolution, the GPU baseline kernel ran in 0.1181 ms, while the optimized planar version achieved 0.1131 ms. The proximity of these values proves that the conversion is already bandwidth-limited and close to the hardware throughput ceiling. While the planar layout simplifies independent channel processing in later pipeline stages, it brings almost no performance gain for the conversion step itself.

The plot in figure 3 shows the divergence between CPU and GPU performance of Sobel implementation. The sequential CPU baseline scaled from 40.79 ms to 645.48 ms as resolution increased. Meanwhile, GPU implementations maintained sub-millisecond execution times. We encountered a performance regression in the optimized shared-memory Sobel kernel. At 2048×2048, the optimized version required 0.2452 ms, whereas the baseline GPU kernel finished in 0.1763 ms.

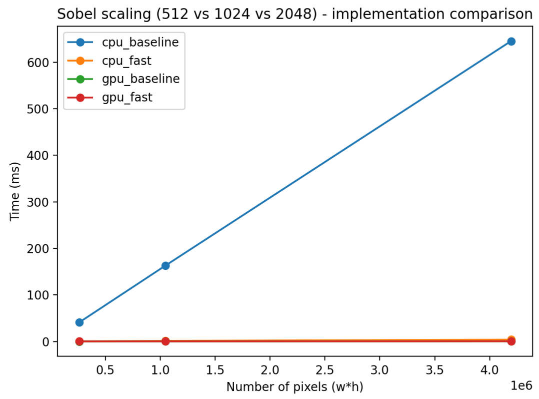

Fig. 3. Sobel scaling (all resolutions)

This result comes from a clear trade off between handling caching manually and relying on the GPU's built in cache. The 3×3 Sobel stencil does not involve much computation compared to the amount of memory it accesses. In the baseline version, the GPU's hardware cache already reuses data efficiently. When we switched to manual shared memory tiling, we added synchronization overhead and more complicated indexing to load the halo regions. For such a small 3×3 window, that extra management cost was higher than the savings from reducing global memory accesses, which explains the slowdown at high resolutions.

C. Gaussian Complexity Paradox

The Gaussian blur implementation provides the most significant deviation between theoretical algorithmic complexity and actual hardware performance. We evaluated the standard 2D convolution against the separable 1D passes across multiple kernel sizes K. The results highlight how memory behavior can dominate performance and reduce the practical benefits suggested by the lower theoretical complexity.

1) CPU and GPU Baseline Behavior: On the CPU, the separable implementation remains the only viable strategy as

K increases. In the plot in figure 4, the 2D convolution follows a quadratic $O(K^2)$ growth, with execution time reaching 985.1239 ms for $K = 11$. Conversely, the separable version scales linearly, maintaining a much lower profile at 168.4606 ms for the same kernel size. The gap comes from performing far fewer arithmetic operations and making better use of the cache. In the 2D baseline, each pixel requires K^2 memory accesses, which quickly exceeds the capacity of the L1 and L2 caches as the kernel becomes larger.

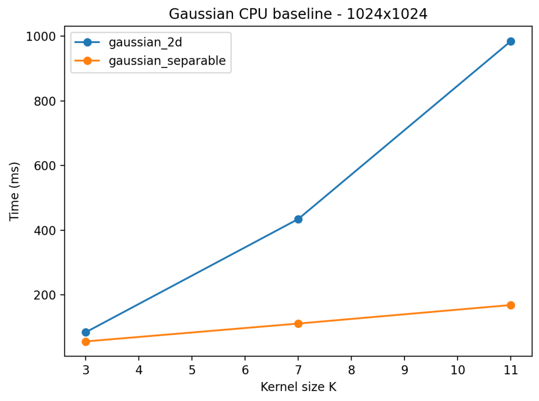

Fig. 4. Gaussian CPU Baseline (1024x1024)

The same behavior is initially visible in the GPU baseline (see figure 5). Without specific memory optimizations, the separable approach performs better because global memory bandwidth becomes the limiting factor. In the 2D GPU version, each output pixel still requires K^2 reads from global memory, generating a large amount of DRAM traffic that ultimately determines the execution time.

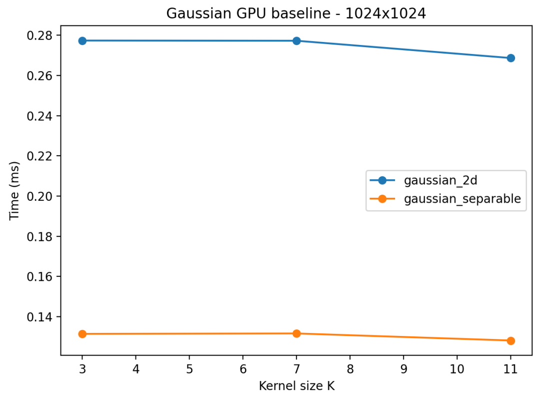

Fig. 5. Gaussian GPU Baseline (1024x1024)

2) Shared Memory Inversion: The results from the optimized GPU implementation (see figure 6) contradict standard complexity theory. In this configuration, the optimized 2D kernel consistently performs better than the separable version. This turnaround happened because the system shifted from being compute-bound to being memory-bound.

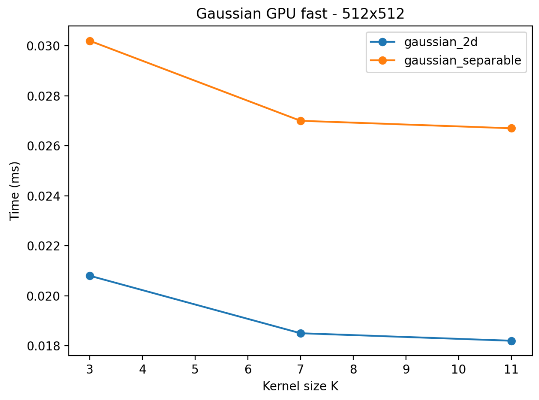

Fig. 6. Gaussian GPU Fast (512x512)

The optimized 2D kernel performs a single tiled convolution using shared memory, which limits global memory access to one read and one write. In contrast, the separable implementation requires two separate kernel launches and an intermediate buffer in global memory. We used the `float` format for this buffer to maintain precision, which increased global memory traffic by four times compared to the `uint8` format.

The main limitation moved from computational intensity to memory bandwidth. While the separable version requires fewer multiplications, $O(2K)$ versus $O(K^2)$, the cost of writing and then reading a large intermediate float buffer from global memory is greater than the cost of the additional arithmetic operations in the 2D tiled pass. During development, we observed that for kernel sizes up to 11×11, the overhead of the second kernel launch and the extra DRAM traffic makes the 2D version more efficient. This indicates that the separable version becomes meanginful only for much larger kernel sizes, where the $O(K^2)$ arithmetic cost eventually becomes larger than the memory overhead.

D. Throughput and Bandwidth Analysis

The test results clearly indicate the relative efficiency of each option (see table I). We calculated the effective bandwidth by measuring the total amount of data transferred through global memory and dividing it by the kernel execution time. This metric is the main indicator of how effectively our implementations use the Tesla T4 memory controller.

The throughput results highlight a huge gap between the sequential CPU baseline and the optimized GPU kernels. RGB2YUV achieved a peak throughput of more than $33,000$ MPixels/s. This level of performance is possible because the kernel is highly parallel and uses a planar layout that allows perfect memory coalescing. The effective bandwidth of 198.47 GB/s corresponds to about 62% of the Tesla T4 theoretical peak, which is an excellent outcome for a pointwise operation.

Stencil operations show lower throughput due to increased complexity. Sobel filtering achieved $17,534$ MPixels/s, while Gaussian blur 2D reached $8,893$ MPixels/s. The difference is primarily due to the larger memory footprint of the 7×7 Gaussian kernel compared to the 3×3 Sobel stencil. We

TABLE I
COMPARATIVE PERFORMANCE METRICS FOR 1024×1024 RESOLUTION.

Implementation	**Avg. Time (ms)**	**Throughput (MPix/s)**	**Eff. Bandwidth (GB/s)**
Sobel CPU Naive	163.1003	6.43	0.01
Sobel CPU Fast	1.2636	829.83	1.66
Sobel GPU Naive	0.0637	16461.16	32.92
Sobel GPU Opt.	0.0598	17534.72	35.07
Gauss 2D CPU Naive	434.7314	2.41	0.00
Gauss 2D CPU Fast	198.0247	5.30	0.01
Gauss 2D GPU Naive	0.2773	3781.38	7.56
Gauss 2D GPU Opt.	0.1179	8893.77	17.79
Gauss Sep. CPU Naive	111.0698	9.44	0.09
Gauss Sep. CPU Fast	76.0997	13.78	0.14
Gauss Sep. GPU Naive	0.1316	7967.90	79.68
Gauss Sep. GPU Opt.	0.1448	7241.55	72.42
RGB2YUV CPU Naive	86.6562	12.10	0.07
RGB2YUV CPU Fast	29.1893	35.92	0.22
RGB2YUV GPU Naive	0.0336	31207.62	187.25
RGB2YUV GPU Opt.	0.0317	33078.11	198.47

observed that the separable Gaussian version reported a higher effective bandwidth (72.42 GB/s) than the 2D version (17.79 GB/s). This is not an indication of higher efficiency. Instead, it reflects the massive volume of data moved by the intermediate buffer. The 2D version is objectively faster in terms of wall-clock time because it avoids this redundant traffic entirely.

E. PCIe Overhead and Latency Analysis

The most important finding from our study is the clear impact of the so-called ”Communication Wall.” In our most optimized kernels, such as the GPU-accelerated RGB2YUV, the computation itself takes less than 0.1 ms. However, transferring the 1024×1024 RGB image over the PCIe bus requires several milliseconds. As a result, the GPU spends most of its time waiting for data instead of actually processing it.

To address this imbalance, we used asynchronous CUDA streams. By splitting the image into smaller chunks and using pinned memory, we were able to overlap the Host-to-Device transfer of chunk $N + 1$ with the processing of chunk N. This approach masked a large portion of the data transfer latency. However, this optimization has its limits. When the chunks become too small, the overhead from launching many kernels and handling stream synchronization begins to undermine the performance gains.

Our analysis shows that GPU acceleration is not a universal solution. For small images or simple filters, the overhead of memory transfers and kernel launches can exceed the execution time of an optimized CPU implementation. The real benefit of the GPU becomes clear at higher resolutions and with more computationally intensive workloads. Indeed, at 2048×2048, the ratio between computation and data transfer becomes favorable, allowing the high throughput of the SMs to justify the cost of PCIe transfers.

VII. CONCLUSIONS

A. Final Remarks

This project examined the gap between theoretical algorithmic complexity and actual hardware throughput on the NVIDIA Turing architecture. By evolving our implementations from naive C++ baselines to optimized CUDA kernels, we identified that performance in modern computer vision workloads is rarely defined solely by arithmetic intensity, but rather by the efficiency of the memory hierarchy and the PCIe bus.

Our results on the RGB-to-YUV conversion demonstrate that pointwise operations are strictly bandwidth-bound. By adopting a planar output layout, we achieved an effective bandwidth of 198.47 GB/s, effectively saturating more than the half of the Tesla T4’s memory controller. This confirms that for low-arithmetic intensity kernels, memory coalescing and layout optimization yield higher returns than instruction-level tuning.

Conversely, the Gaussian Blur analysis highlighted a ”Complexity Paradox.” While the separable formulation is theoretically superior ($O(K)$ vs $O(K^2)$), our benchmarks showed that the Optimized 2D Tiled implementation outperformed the separable version for standard kernel sizes (3×3 to 11×11). The overhead of the intermediate global memory buffer required by the separable filter proved more costly than the redundant arithmetic of the 2D kernel. This finding underscores that on GPUs, minimizing global memory traffic is often more critical than reducing the total number of floating-point operations.

Finally, the Sobel experiments revealed the trade-off of manual cache management. While shared memory tiling significantly reduced DRAM fetches, the synchronization overhead became a regression factor at very high resolutions compared to the hardware-managed L2 cache.

B. Potential Improvements

While our current implementations achieve significant speedups over CPU baselines, further architectural optimizations could push performance closer to the theoretical hardware limits:

- **Vectorized Loads (uchar4)**: currently, our kernels process image data using standard `unsigned char` pointers. A significant optimization would be to implement vectorized memory access using intrinsic data types like `uchar4` or `float4`. By loading a packed pixel (RGB or RGBA) in a single 32-bit transaction rather than three separate 8-bit reads, we could reduce the number of issued load instructions and improve instruction-level parallelism. This is particularly relevant for the RGB-to-YUV kernel, where increasing the memory transaction size could further improve bus utilization.
- **Advanced Asynchronous Pipelines (CUDA Graphs)**: we successfully employed CUDA Streams to hide PCIe latency by overlapping data transfers with kernel execution. However, as noted in our results, small chunk sizes introduce CPU-side launch overhead that can negate the benefits of concurrency. Future iterations could leverage CUDA Graphs. By capturing the sequence of memory transfers and kernel launches into a static graph, we could eliminate the runtime overhead of the driver's work submission. This would allow for much finer-grained concurrency, making the streaming approach viable even for lower resolutions or smaller chunk sizes.
- **Multi-GPU Scalability**: the domain decomposition logic currently implemented prepares the software for distributed execution. Extending this project to support Multi-GPU environments via NCCL (NVIDIA Collective Communications Library) would allow for real-time processing of 8K video streams, overcoming the memory capacity limits of a single device.

C. Code Availability

The source code is publicly available at: https://github.com/DarkShrill/GPUImageLab.git

References

[1] NVIDIA Corporation, "CUDA C++ Programming Guide," *NVIDIA Documentation*, ver. 12.0, 2023. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[2] NVIDIA Corporation, "NVIDIA Turing GPU Architecture: Whitepaper," *NVIDIA Corporation*, 2018. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

[3] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 4th ed., Pearson, 2018.

[4] S. Barrett, "stb_image.h - v2.28 - public domain image loader," *GitHub Repository*, 2023. [Online]. Available: https://github.com/nothings/stb