HPC - 2. Assignment - Parallel Histogram Equalization using CUDA

Kristjan Kostanjšek, Nejc Ločičnik

I. Introduction

Histogram equalization is a classic image enhancement technique that improves contrast by redistributing pixel intensities across the available range. The algorithm works by calculating and transforming the image's histogram, making it particularly suitable for parallel implementation on GPUs due to its pixel-independent operations. The transformation in image's histograms is demonstrated in Figure 1

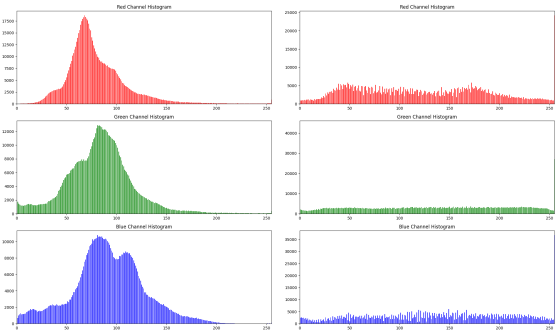

Figure 1. RGB histograms before and after histogram equalization.

We implemented and compared several CUDA optimization strategies for each stage of the pipeline, achieving significant speedups especially for larger images. Our best parallel implementation shows up to 8.89× improvement over the sequential version for 8K resolution images, with individual stages reaching speedups as high as 1792× for the most parallelizable operations.

II. Optimizations and Results

The following subsections detail the optimizations applied to the sequential code, along with the computational speed-up achieved for each optimization. Each optimization is looked at in isolation. This means that the reported optimization speed-ups are for that specific part of the algorithm and not for the full execution. The full execution speed-ups are reported at the end. The speed-up is calculated using the formula: $S = t_S/t_P$, where t_S represents the execution time of the sequential program, and t_P denotes the execution time of the parallel program. All reported times and speed-ups are averages of 100 runs.

A. RGB to YUV Conversion and Luminance Histogram Computation

This initial stage performs two critical operations simultaneously: converting RGB pixels to YUV color space while computing the luminance (Y channel) histogram. The conversion requires independent per-pixel calculations, making it ideal for parallel processing, while the histogram computation benefits from shared memory optimizations to efficiently combine results across threads. We evaluated five parallel implementations with increasing optimization complexity: (1) Global uses direct atomic operations on device memory; (2) Basic employs block-level shared memory histograms; (3) Large partitions histograms by warps to reduce contention; (4) Bitmap uses bit-wise operations for voting; (5) Bins distributes writes across multiple histogram buffers.

M\I Size	720x480	1024x768	1920x1200	3840x2160	7680x4320
Global	0.0722	0.1504	0.2914	1.1924	3.7753
Basic	0.0556	0.0987	0.3890	1.1829	3.5235
Large	0.0675	0.1221	0.2455	1.3456	4.2074
Bitmap	0.0654	0.1200	0.3737	1.4686	3.8025
Bins	0.0664	0.1658	0.3144	1.5327	5.1539

Table I
Performance comparison for RGB to YUV conversion and luminance histogram computation (speed-ups S).

The results in Table I show significant variance across implementations, with standard deviations of 180–240 ms due to inconsistent atomic function contention over 100 runs. Despite this variability, clear trends emerge: naive approaches like Global (atomic device memory) and Basic (block-level shared memory) work reasonably well at smaller resolutions (720x480, 1920x1200), but suffer from contention at higher resolutions. Warp-level optimizations (Large, Bitmap) reduce contention by partitioning work—either per-warp or via bit-wise voting—but at a steep cost in shared memory: 256×32 integers (one histogram per warp or one bit per thread) compared to just 256×4 integers for the Bins method. The large shared memory approach can actually be further optimized memory-wise by not using integers as we require 10 bits at most (integers provide 32 bits), but sadly atomic add is only implemented for integers.

Critically, Bins not only uses 8× less shared memory per block (improving occupancy) but also uniquely mitigates locality-based contention. By distributing consecutive pixels across separate bins, it breaks up clustered writes that would otherwise collide in warp- or block-level approaches. This makes it far more scalable than warp-level methods, particularly at high resolutions where spatial coherence in luminance values would otherwise exacerbate contention. The data confirms that Bins is the most robust choice for practical workloads, especially beyond 1920x1200.

B. Cumulative Histogram Computation

The cumulative histogram computation transforms individual bin counts into running sums, with our GPU implementation using a work-efficient parallel scan algorithm in shared memory (a single block with 256 threads) to minimize global memory accesses. The CPU employs a sequential approach compiled with optimizations (probably some SIMD vectorization), though its performance appears sensitive to the magnitude of values processed rather than just operation count.

I Size	720x480	1024x768	1920x1200	3840x2160	7680x4320
$t_S[ms]$	0.006	0.007	0.005	0.014	0.016
$t_P[ms]$	0.01	0.01	0.01	0.01	0.01
t_S/t_P	0.6	0.7	0.5	1.4	1.6

Table II
PERFORMANCE COMPARISON FOR CUMULATIVE HISTOGRAM COMPUTATION.

Results in Table II show that the GPU maintains constant 0.01ms execution across all sizes due to fixed parallel overheads, while CPU times increase with larger cumulative sums. The GPU only becomes advantageous for 4K+ resolutions (speedup >1), suggesting a hybrid CPU/GPU approach would be optimal - using CPUs for smaller images and GPUs for larger ones, with the crossover occurring around 3840x2160 resolution.

C. New Luminance Computation

The new luminance computation stage transforms histogram values into normalized luminance intensities using a simple arithmetic operation per intensity level. Our GPU implementation assigns one thread per luminance value (256 threads total) to parallelize these independent computations, while the CPU uses an optimized sequential loop with SIMD vectorization.

New luminance computation times and speed-ups are shown in Table III.

I Size	720x480	1024x768	1920x1200	3840x2160	7680x4320
$t_S[ms]$	0.00083	0.00082	0.00077	0.00082	0.00057
$t_P[ms]$	0.01	0.01	0.01	0.01	0.01
t_S/t_P	0.083	0.082	0.077	0.082	0.057

Table III
PERFORMANCE COMPARISON FOR NEW LUMINANCE COMPUTATION.

The CPU's 8-17× faster performance stems primarily from its caching advantage - the entire 256-value working set fits comfortably in CPU cache, while the GPU incurs overhead from shared memory access patterns and kernel launch latency. This cache benefit outweighs the GPU's theoretical SIMT advantage, demonstrating that memory access patterns often dominate performance for small, data-light computations. The results suggest keeping such operations on the CPU unless already processing data in GPU memory.

D. Assign New Luminance and YUV to RGB Conversion

The final stage performs two key operations in a single GPU kernel: (1) replacing original luminance values using our precomputed lookup table, and (2) converting the equalized YUV image back to RGB space. This fused kernel design eliminates intermediate memory transfers and maximizes memory locality by processing each pixel's complete transformation in a single thread.

I Size	720x480	1024x768	1920x1200	3840x2160	7680x4320
$t_S[ms]$	9.8	22.15	60.64	209.72	502.04
$t_P[ms]$	0.01	0.019	0.04	0.117	0.44
t_S/t_P	980	1166	1516	1792	1141

Table IV
PERFORMANCE COMPARISON FOR LUMINANCE ASSIGNMENT AND COLOR CONVERSION.

Results in Table IV demonstrate ideal GPU scaling, with speedups ranging from 980× to 1792× across image sizes. The GPU's sub-millisecond execution times (0.01-0.44ms) highlight the effectiveness of pixel-level parallelism for these memory-bound operations. While this stage shows near-perfect acceleration, its overall impact is constrained by earlier histogram computation bottlenecks. The massive speedups confirm that per-pixel operations represent the "sweet spot" for GPU acceleration in image processing pipelines.

E. Final results

We present the performance comparison between sequential execution (t_S) and the best-performing parallel implementations (t_P) for each image resolution. Our evaluation reveals that different histogram computation strategies excel at different scales: the global memory approach performed best for the smallest image (720x480), basic shared memory worked best for mid-range resolution (1920x1200), while the bin-based shared memory approach showed superior performance for all other tested resolutions. This demonstrates that optimal implementation choice depends heavily on the working set size.

I Size	720x480	1024x768	1920x1200	3840x2160	7680x4320
$t_S[ms]$	20.111	45.656	123.89	430.94	1188.9
$t_P[ms]$	142.49	139.49	162.62	144.46	133.74
t_S/t_P	0.1411	0.3273	0.7618	2.9831	8.8896

Table V
EXECUTION TIMES COMPARISON FOR EACH RESOLUTION, WITH CORRESPONDING SPEEDUP FACTORS (100 RUNS AVERAGE).

Results in Table V demonstrate several key trends. For small images (720x480), the overhead of parallelization outweighs the benefits. However, as image size increases, the parallel implementation shows increasingly better performance, reaching 8.89× speedup for 8K resolution. Notably, the parallel execution time remains remarkably consistent (133-163ms) across all resolutions, while sequential time grows linearly with pixel count. The high variance (180-240ms) observed across runs stems from contention in histogram computation, despite averaging over a 100 runs. The bin-based approach proves particularly effective for larger images, as its 4-bin distribution strategy effectively mitigates locality-induced contention while maintaining low memory overhead.

III. Conclusion

In this report, we implemented and optimized a histogram equalization algorithm using CUDA, with particular focus on the histogram computation stage which dominated 99% of execution time. We explored the critical balance between reducing atomic operation contention and maintaining sufficient block-level occupancy through shared memory optimizations. Our best parallel implementation achieved up to 8.89× speedup for 8K images, with pixel-level operations reaching 1792× improvements due to efficient GPU parallelism. The bin-based histogram approach proved most effective for larger resolutions by minimizing both contention and shared memory usage, while simpler methods worked better for smaller images.

Future work could explore hybrid CPU-GPU execution to leverage each architecture's strengths, along with further shared memory optimizations like adaptive bin counts based on image resolution. Additional improvements might include bit-packing histogram bins or dynamic switching between computation strategies to better handle varying image sizes. These enhancements would further optimize the algorithm's performance across different hardware configurations and use cases.