High Performance Computing Systems
Project 4

Tsogkas Panagiotis Nikas Ioannis Iason Tsiantos Dimitrios

December 2025

Contents

1 The Algorithm 3

2 Initial Implementation 4

3 Implementation with two kernels 4

4 Eliminating Kernel Asymmetry 4

5 Avoiding Atomic Serialization using Privatization 5

6 Inclusive Parallel Scan Implementation 5

7 Memory Coalescence 6

8 Using Streams 7

9 Using Events 8

10 Second Kernel 10

11 Execution Times 10

1 The Algorithm

The CLAHE algorithm can be divided into two main steps. In the first step, we divide the image into tiles and create a histogram for each tile with greyscale values from $0 - 255$ on the horizontal axis and the pixel count on the vertical axis. Subsequently, we remove the count of values exceeding a predefined limit from the histogram and redistribute this amount equally among the remaining columns. Finally, we calculate and store the CDF of the histogram for each tile in an array, which constitutes the result of the first step.

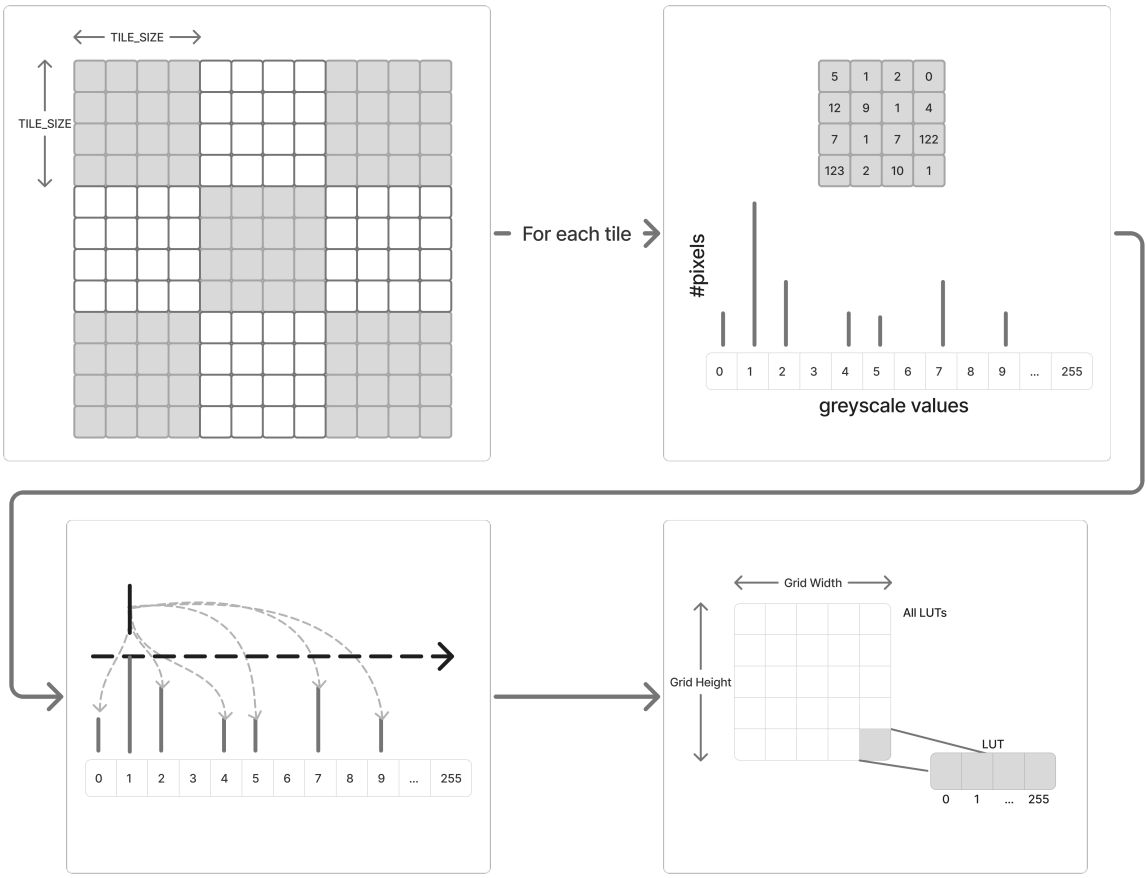


Figure 1: First step of the algorithm.

In the second step, we apply bilinear interpolation for each pixel so that the final intensity value is derived from the four neighboring tiles. For each pixel, we identify the two closest tiles on the horizontal axis and the two on the vertical axis. From the array generated in the first step, we access the corresponding CDF values for the greyscale value of the specific pixel. This ensures that the transition between tiles occurs gradually.

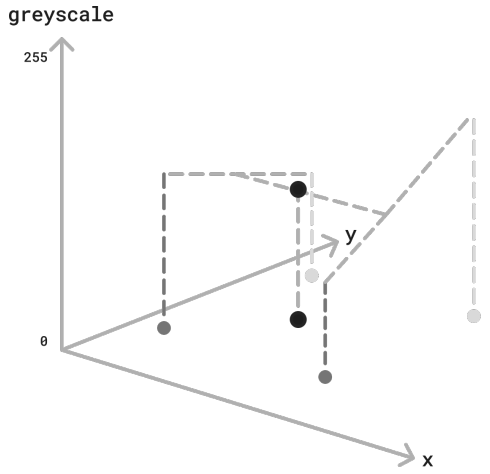


Figure 2: Second step of the algorithm. Example of bilinear interpolation.

2 Initial Implementation

In the simplest implementation of the algorithm, we created three different kernels. The first calculates the histogram and stores it in global memory so that it can be used by the second kernel, which calculates the LUTs for each tile. Then, the third kernel uses the LUTs to calculate the final pixel values by performing bilinear interpolation. This allows us to use the finest granularity for the threads we create, i.e., using one thread per pixel in the first and third kernels and one thread per histogram bin in the second kernel. However, this increases transfers between global and shared memory. Therefore, we proceeded to change the kernel structure to reduce transfers and to facilitate easier optimizations in subsequent versions.

3 Implementation with two kernels

The implementation that improves upon the previous one uses 2 kernels. The first calculates the histogram, using one thread per pixel and one block per tile, while the first 256 threads of each block also calculate the values for the LUT of that specific tile. Thus, the histogram no longer needs to be transferred to global memory at all, since it is not reused after the calculation of the LUTs. However, this reserves resources that are not used throughout the entire duration of the kernel execution. For this reason, we proceeded to an intermediate solution described below.

Run	Average Throughput (MPixels/s)	Std. Deviation	Speed-up vs Previous
3-kernels	703.491	21.390	-
2-kernels	470.069	5.089	0.668

Table 1: Speed comparison: 3-kernels → 2-kernels

4 Eliminating Kernel Asymmetry

With the merging of the code into two kernels, as seen in the previous step, an asymmetry arose in the way the kernels are called. Specifically, we ask the Streaming Multiprocessor to schedule blocks with 1024 threads, but before their execution is complete, we release only a part of them, as only the first 8 warps have remaining code to run while the remaining 24 finish.

This pattern can lead to a situation where threadsRequested – threadsUsed = 768 threads remain idle because there are not enough free threads to execute an additional block. As a result, execution continues serially, limiting the degree of parallelism and reducing performance.

In this specific implementation, we observe that we have 100% thread usage during histogram creation, but only 25% thread usage in the next stage. For this reason, we reduced the number of threads in each kernel block by a factor of four, to 256. In this way, each thread now calculates four elements of the histogram, while one thread calculates one index from the corresponding probability array (LUT) for each tile.

With this method, no thread remains unused, thus eliminating the potential parallelism bottleneck during scheduling by the Streaming Multiprocessor, at least regarding the number of available threads.

Run	Average Throughput (MPixels/s)	Std. Deviation	Speed-up vs Previous
2-kernels	470.069	5.089	-
symmetric	722.906	14.800	1.538

Table 2: Speed comparison: 2-kernels → symmetric

5 Avoiding Atomic Serialization using Privatization

A problem observed in previous implementations is the continuous use of `atomicAdd`, which can lead to atomic contention. In this state, multiple calls of the same function to the same memory address are forced to execute serially, creating significant delay. For this reason, in this specific implementation, we used privatization.

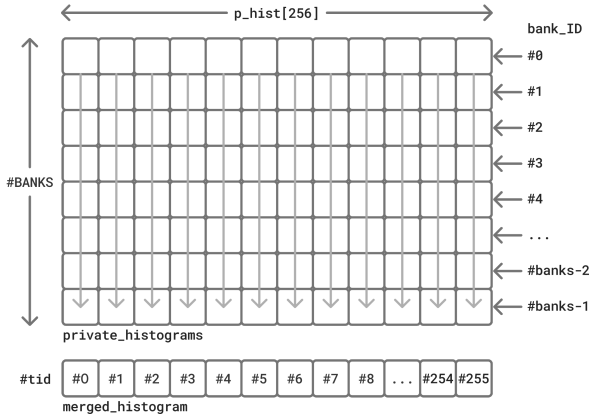


Figure 3: Structure of privatization implementation

The structure of the implementation includes a `__shared__` array, where each row corresponds to a private copy of the histogram. Each private copy is used by threads with consecutive thread IDs (tid), distributed cyclically per `NUM_BANKS`. Since the access pattern to the histogram positions is almost random, the number of BANKS effectively reduces the probability of serialized memory requests.

Run	Average Throughput (MPixels/s)	Std. Deviation	Speed-up vs Previous
symmetric	722.906	14.800	-
no-atomic-serial	854.920	18.813	1.183

Table 3: Speed comparison: symmetric → no-atomic-serial

6 Inclusive Parallel Scan Implementation

For the calculation of the CDF in each tile within the `compute_histogram` kernel, we observe that the use of the prefix sum algorithm is required. Until now, its implementation involved each thread calculating the sum for itself, which involves many more operations than ideal. Therefore, we implemented

the version that executes in parallel and within shared memory. Specifically, the algorithm operates iteratively with a logarithmic number of steps. In each iteration, the stride doubles, and each thread adds the value of the element located at distance *stride* to its own. This process requires careful synchronization (`__syncthreads()`) to avoid race conditions during reading and writing to shared memory. Regarding bank conflicts, the implementation avoids them thanks to linear memory access. Within a warp, consecutive threads access consecutive memory addresses (with a constant stride offset). The substantial gain is that we thus avoid the serialization that would arise if each thread calculated its own CDF serially.

Run	Average Throughput (MPixels/s)	Std. Deviation	Speed-up vs Previous
no-atomic-serial	854.920	18.813	-
scan-impl	897.553	32.925	1.050

Table 4: Speed comparison: no-atomic-serial → scan-impl

7 Memory Coalescence

Before proceeding, it is useful to examine the memory accesses we perform in global memory and to what extent they are acceptable.

In the first kernel, we have the following global memory accesses:

```
for (int cur_y = ty; cur_y < TILE_SIZE; cur_y += step_y) {
    for (int cur_x = tx; cur_x < TILE_SIZE; cur_x += step_x) {
        int gx = x_start + cur_x;
        int gy = y_start + cur_y;
        if (gx < w && gy < h) {
            unsigned char pix = data[gy * w + gx];
            ....
        }
    }
}
```

In the code segment above, we access the variable `data` in global memory in a way that allows threads to execute parallel and continuous requests to the corresponding memory locations, resulting in coalesced accesses. In Figure 4, threads are depicted with different colors, and the numbers indicate the serial accesses of each thread. As accesses occur in groups of 16 threads, coalescing is achieved.

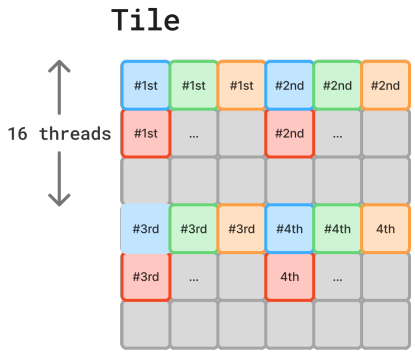


Figure 4: Access pattern within the tile.

By similar logic, in the first kernel during the execution of:

```
// Write to Global Memory
unsigned char *lut = &(all_luts[(blockIdx.y * gridDim.x + blockIdx.x) * 256]);
```

because 256 consecutive and simultaneous accesses are performed, the access pattern is also coalesced. Regarding the kernel `render_clahe`, the accesses are not necessarily coalesced as the access pattern depends on the input. However, the addresses are within a small range (0-256), which increases the probability that some accesses will be coalesced.

```
val = img_in[y * w + x];

// Fetch mapped values from the 4 nearest tile LUTs
tl = all_luts[(y1 * gridDim.x + x1) * 256 + val];
tr = all_luts[(y1 * gridDim.x + x2) * 256 + val];
bl = all_luts[(y2 * gridDim.x + x1) * 256 + val];
br = all_luts[(y2 * gridDim.x + x2) * 256 + val];
...

img_out[y * w + x] = (unsigned char)(final_val + 0.5f);
```

8 Using Streams

Observing the results of the `nvprof` tool, we see that most time is spent on data transfers to and from the device. In our implementations so far, we move the entire image, using the blocking command `cudaMemcpy`, once at the beginning of the program from host to device and back again when processing is complete. However, a kernel block does not use the entire image. For this reason, as soon as the data corresponding to the processing tile of a kernel block arrive at the device, we can immediately start execution without waiting for the transfer of the entire image.

To implement this functionality, we used Streams. In an initial version, we mapped each tile to a separate stream (one-to-one mapping). This approach proved to be particularly slow, as it required the creation of a very large number of streams, significantly burdening the system.

In the current implementation, the number of streams is variable and determined by the parameter `NUM_STREAMS`. In this way, we achieve a better balance between parallelism and stream management overhead. After experimentation, we concluded that the optimal value for `NUM_STREAMS` is ~ 8.

However, this method of data management imposes multiple kernel calls to parallelize the input data; consequently, we must change the Grid on which each kernel works within a stream. We now partition our data in the way shown in Figure 5.

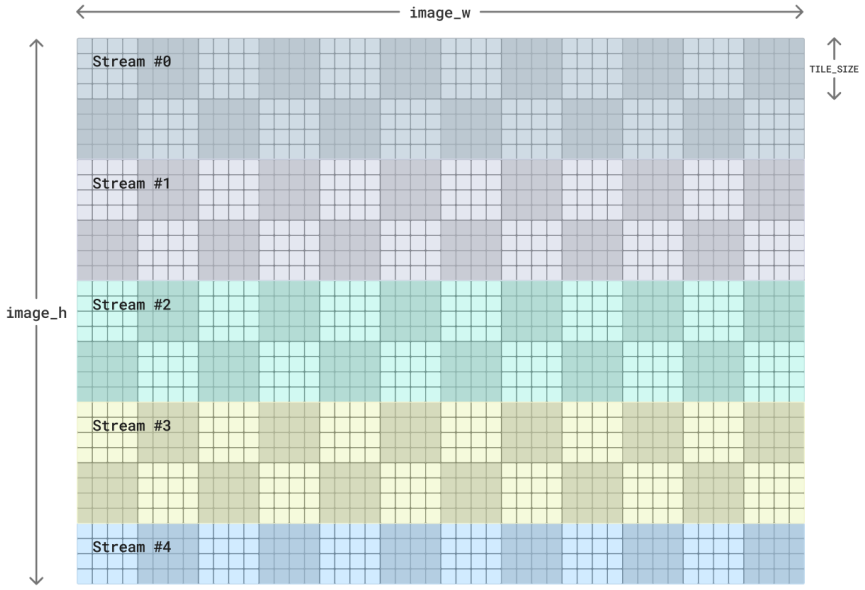


Figure 5: Structure of Streams over the image

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU	78.69%	195.07 ms	1	195.07 ms	195.07 ms	195.07 ms	[CUDA memcpy DtoH]
GPU	11.34%	28.100 ms	1	28.100 ms	28.100 ms	28.100 ms	[CUDA memcpy HtoD]
GPU	5.03%	12.473 ms	1	12.473 ms	12.473 ms	12.473 ms	compute_histogram
GPU	4.94%	12.243 ms	1	12.243 ms	12.243 ms	12.243 ms	render_clahe
API	57.23%	224.23 ms	2	112.11 ms	28.153 ms	196.07 ms	cudaMemcpy
API	22.48%	88.079 ms	2	44.039 ms	640 ns	88.078 ms	cudaEventCreate
API	10.35%	40.559 ms	1	40.559 ms	40.559 ms	40.559 ms	cudaDeviceReset
API	6.33%	24.811 ms	9	2.7567 ms	920 ns	12.475 ms	cudaDeviceSynchronize
API	3.37%	13.199 ms	3	4.3997 ms	369.28 µs	6.5288 ms	cudaFree
API	0.09%	358.53 µs	3	119.51 µs	90.306 µs	146.19 µs	cudaMalloc
API	0.08%	297.54 µs	2	148.77 µs	6.6900 µs	290.85 µs	cudaLaunchKernel

Table 5: `nvprof` results for execution without streams on an nvidia GTX 1070 graphics card.

Run	Average Throughput (MPixels/s)	Std. Deviation	Speed-up vs Previous
streams	1234.382	53.577	-
stream-both	1223.278	70.379	0.991
stream-events	1333.821	55.206	1.091

Table 6: Speed comparison: streams → stream-both → stream-events

Using the `nsight systems` tool, we observe the parallelization between transfers from the GPU to the main CPU memory and the kernel invocations.

9 Using Events

Between the first and second steps of the algorithm, instead of using the function `cudaDeviceSynchronize` and waiting for all kernel invocations to complete, we can use events to inform the next kernel that its dependencies are ready. That is, transitioning from the first kernel to the second, it is necessary for the previous, current, and next streams to have completed. This version of the algorithm (version 9) is also the final one with the highest throughput.

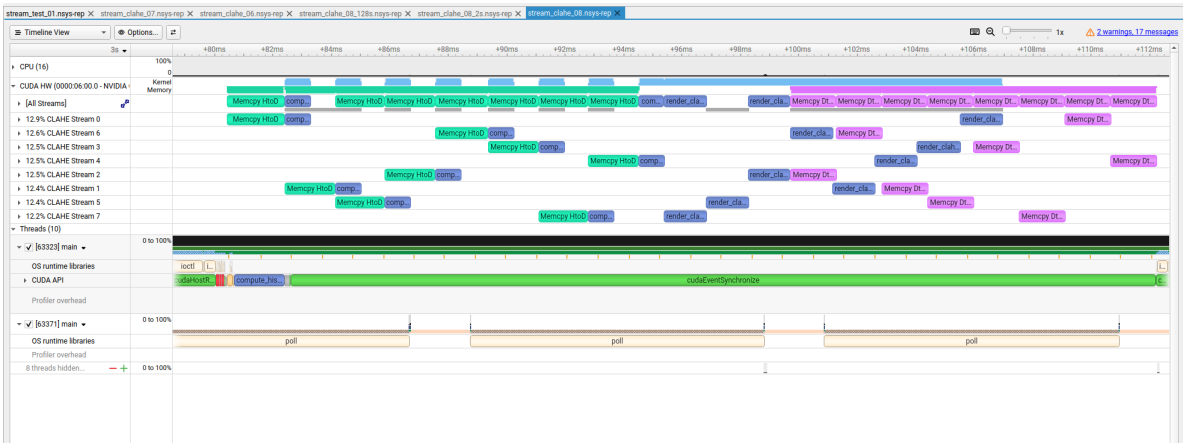


Figure 6: H2D transfers (green), kernel executions (blue), and D2H transfers (pink).

10 Second Kernel

In the implementations so far, we have not mentioned the second kernel. For this, we can apply an optimization based on transferring the probability data (LUT) of neighboring tiles to shared memory. In this way, when a pixel needs to calculate its value, it can look it up directly in shared memory, avoiding accesses to global memory.

With a simple calculation, we see that storing the LUTs of 9 neighboring tiles requires:

$$256 \cdot 4 \cdot 9 = 9216 \text{ bytes.}$$

In practice, however, after the changes required to calculate the new indices in the array `shared_lut`$[256 \cdot 4 \cdot 9]$, no substantial improvement in speed was observed. Therefore, in the final code, we do not have changes in the logic of kernel 2.

11 Execution Times

To compare the throughput per code version, we used a device from an nvidia K80 graphics card, executing each version 25 times and removing the two extreme values. Each code version builds upon the previous one.

Number	Name
01	3-kernels
02	2-kernels
03	256-threads per tile
04	privatization
05	inclusive parallel scan
06	streams H2D
07	streams D2H
08	streams both
09	events

Table 7: Code versions.

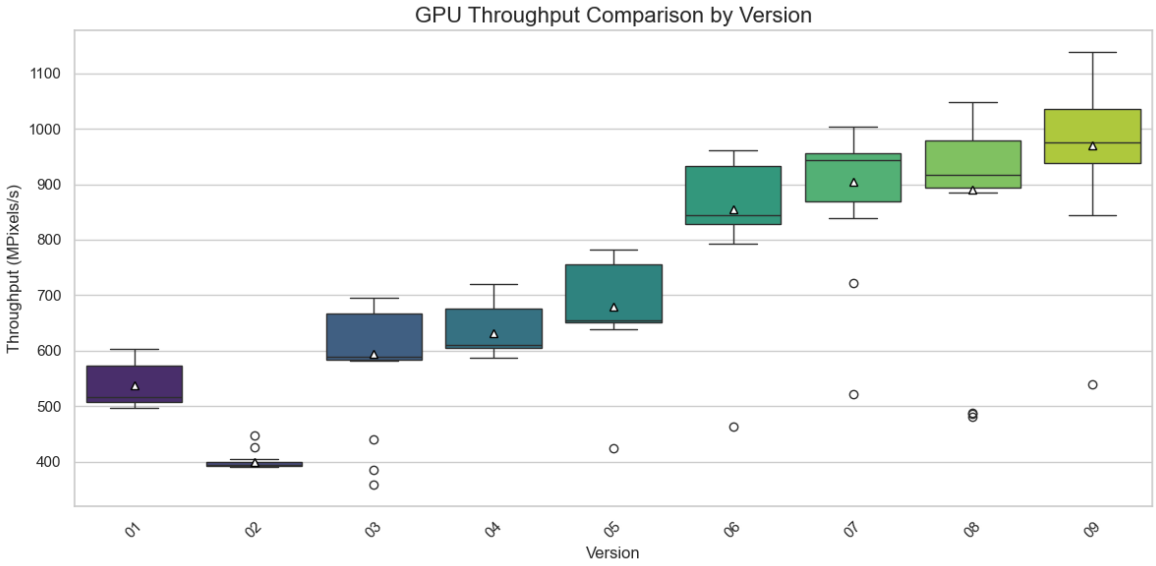


Figure 7: Throughput of image `planet_surface.pgm`.

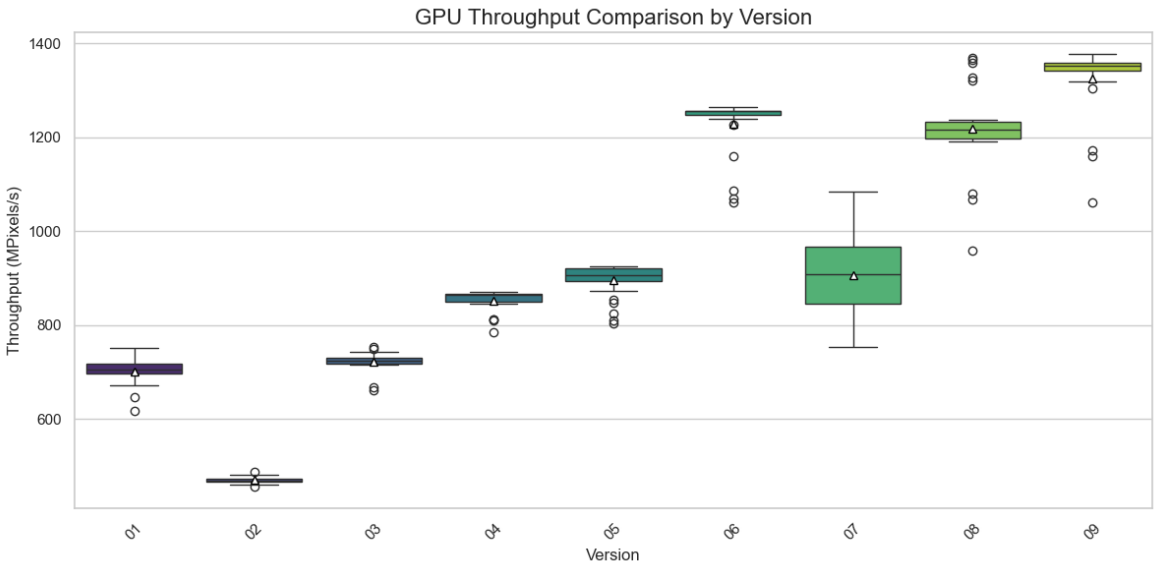


Figure 8: Throughput of image `senator.pgm`.

Run	Average Throughput (MPixels/s)	Std. Deviation	Speed-up vs 3-kernels
3-kernels	703.491	21.390	1.000
stream-events	1333.821	55.206	1.897
CPU	55.778	3.656	0.079

Table 8: Comparison of initial implementation and fastest GPU implementation with the CPU version for image `senator.pgm`.