LAB 4

Christodoulos Zerdalis 03531 Charalampos Malakoudis 03516

December 12, 2025

1 Introduction

In this lab, building on the previous one, we focus on the CUDA environment. Specifically, we aim to parallelize and run a contrast enhancement function on real images using the GPUs of the CSL-Venus machines.

2 Our implementation

In the previous lab, we determined the optimal compile flags for the CSL-Venus machines. Therefore, in this lab, we are using those same flags: **-O4** and **-arch=sm_37** and the **icx** compliler for the cpu version with **-03** optimazation.

Our implementation is split into three stages. The **first** stage involves **computing the histogram**, **clipping**, **redistributing excess**, and calculating the **CDF** and **LUT**. The **second** stage covers **Bilinear Interpolation**, and the **third** focuses on **memory** transfer **optimizations**.

3 First Step

As for the architecture of the initial design, we decided to go for a **single-kernel** approach mainly not to increase the complexity of our design, but also to avoid frequent memory transfers between the kernels and the global memory. The global Memory is hundreds of times slower than the GPU's Shared Memory. By fusing these steps into one kernel, we keep the data residing entirely in high-speed Shared Memory for the duration of the calculations. We only write to Global Memory once at the very end, when the final LUT is complete.

3.1 Optimizations

Firstly, instead of saving histogram values directly to Global Memory, we allocated an array in the block's **Shared Memory**. This significantly reduces latency, as Shared Memory is physically closer to the ALUs and operates at speeds comparable to L2 cache (much faster than Global Memory). To further minimize atomic operations, each **warp** maintains its own **private histogram** in Shared Memory and once computation ends, these private copies are combined into the main block histogram. Although this optimization did not yield a massive performance jump in this specific case, we retained it because it is theoretically superior for scenarios with higher thread counts and heavy atomic contention.

Secondly, we used **reduction** instead of atomic additions to calculate the 'excess' variable with each thread having its own copy of the variable.

Thirdly, For the Cumulative Distribution Function (CDF), we employed a **parallel scan reduction** (prefix sum) on the histogram array. Finally, the writing of the calculated LUT to Global Memory was optimized for coalesced access. Unfortunately, the shared memory accesses during the parallel scan produce bank conflicts that are unavoidable (accessing 256 values from shared memory with 32 banks), but these conflicts are at least evenly distributed.

Initially, we implemented and launched the kernel with a geometry of **32x32** thread per block meaning that each thread processes on pixel since the size of the tiles is also 32x32. Even thought this resulted in the maximum occupancy of 2048 threads in k80 tesla with all of them residing in two blocks, we decided to lower the number of threads to **16x16** which also resulted in max occupancy but spreading the threads to 8 blocks (4 pixels per thread). This is better as the threads work on 8 shared memory modules instead of 2 **reducing the back conflicts** and also the **delays** from **__syncthreads()** are **lower** since there are fewer threads per block that need syncing.

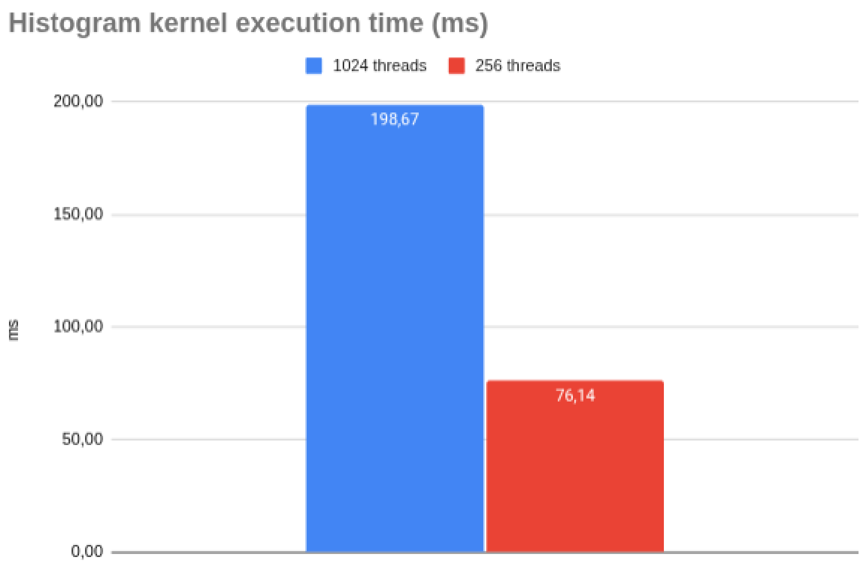

Figure 1: Hre we can see how the kernels delay reduces when we use fewer threads.

4 Second step

In this step, we developed a kernel to perform Bilinear Interpolation between the tiles. We maintained a 2D geometry with a block size of 16x16 threads. This kernel was significantly simpler to implement than the previous stage, as it required no synchronization barriers (__syncthreads()), reductions, or atomic operations.

The primary optimization applied here was storing the LUT (Lookup Table) in **Texture Memory**. Since the LUT accesses are frequent and exhibit random patterns, Texture Memory is superior for this task. It uses a dedicated texture cache that is specifically optimized for spatial locality (2D/3D access patterns), offering better performance than standard Global Memory reads for this specific workload.

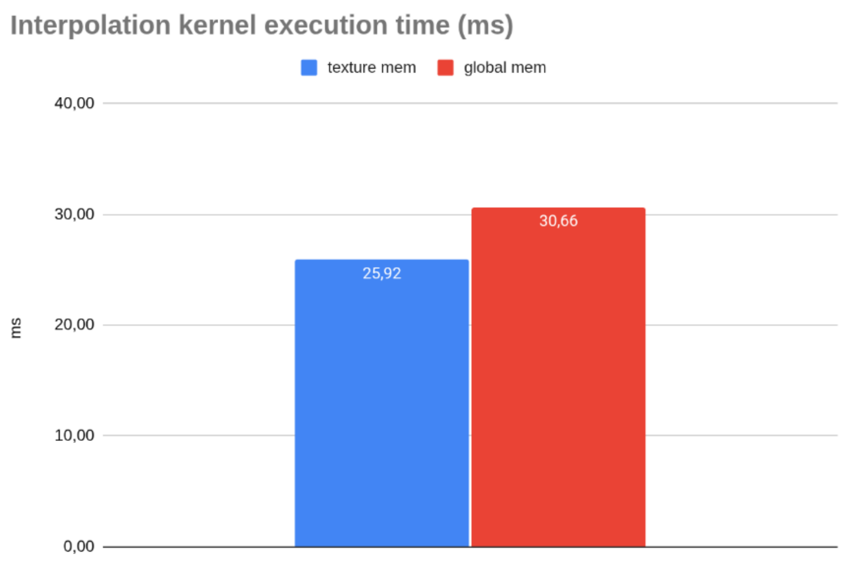

Figure 2: Here we can see how the texture memory inpacts the delay of the interpolation kernel.

5 Third step

A big part of the processing delay of our kernel is the memory transfers from host to device and back.

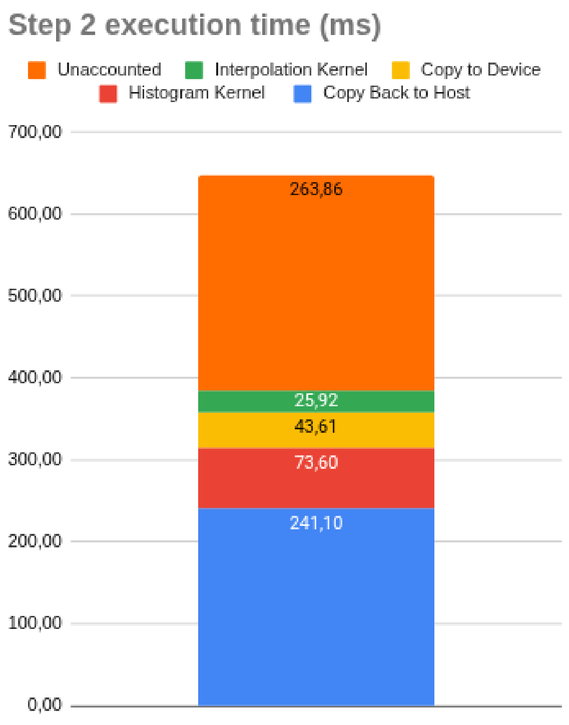

Figure 3: Here we can see from what exactly the delay of the apply_clahe function is caused.

5.1 Device to Host optimazation

Firstly, we tried to optimize the transfer from device to host as it was the biggest bottleneck of our design. We tried approaches like pinned memory and parallel streams, but the method with the best results by far was **Unified Memory**. Unified Memory is allocated in the host's DRAM but can be accessed directly from the device. Although the latency of a single transfer is much higher (since it travels through the PCIe bus) in comparison with the device's local DRAM, the overall results are faster because the transfer delay overlaps with the kernel's execution time. This means that if one thread is finished, it can transfer its result to the output image while the rest are still working, without needing to wait for a large cudaMemcpy to the host.

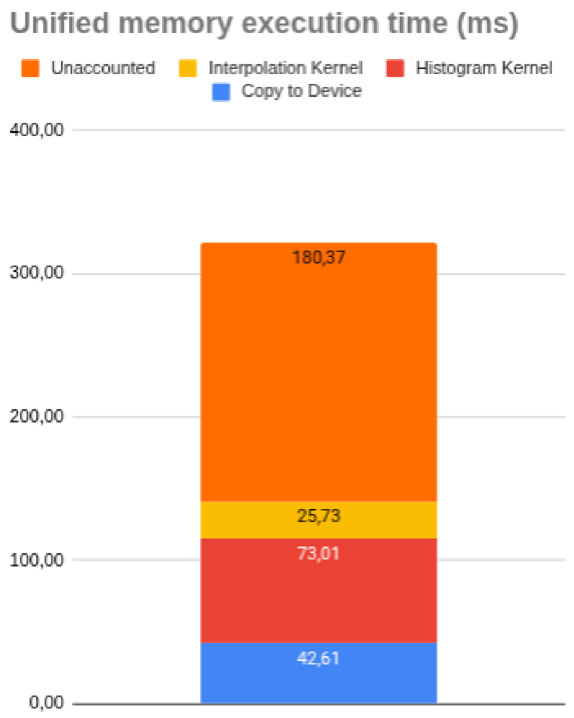

Figure 4: Apply clahe delay after unified memory.

5.2 Host to Device optimazation

For the transfers from host to device, Unified Memory made the design a little slower. **Pinned Memory** actually worked the best. This happens because Pinned Memory has a guaranteed position in physical memory and is not pageable. This allows the GPU to use its DMA engine, which is much faster than a normal memcpy

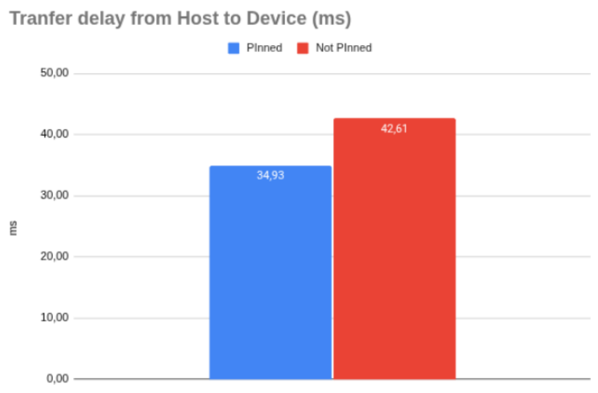

Figure 5: Different from normal to pinned Host memory from input transfers.

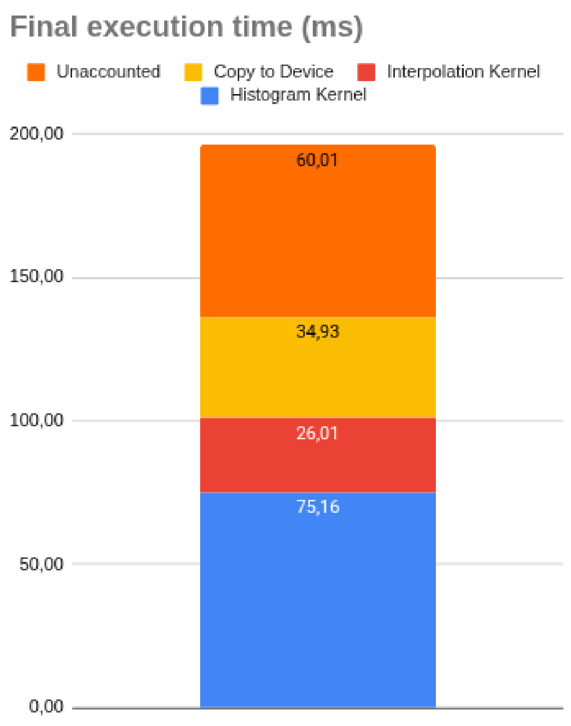

Figure 6: Final version Apply clahe latancy.

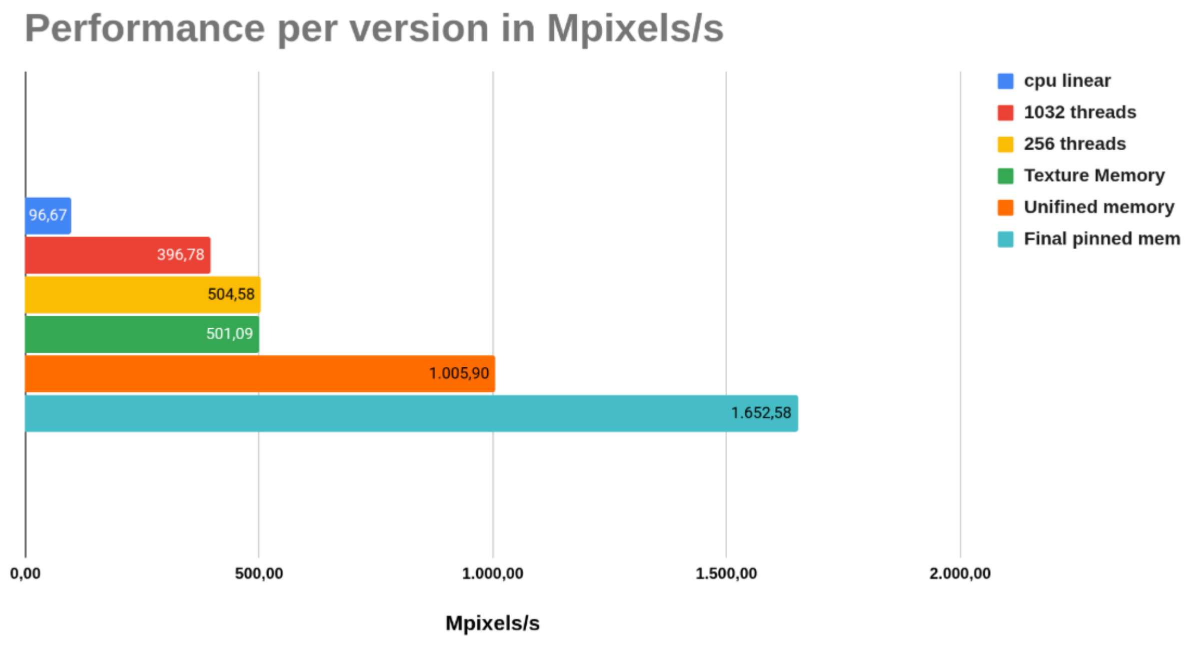

Figure 7: Performance in Mpixels/s for all versions.

6 Extra optimazations

There where two extra optimazations that we did not manage to implement. The first is splitting the input array transfer and histogram computation into two streams in order to hide some of the initial HostToDevice memory transfer latancy. The second is the texture's memory ability to do interpolation in hardware. Unfortunately, we could not find many code examples of this techique to work from.