High Performance Computing

GPU Programming with CUDA: Optimization Strategies (OpenMP and Cuda)

Στολτίδης Αλέξανδρος, Κουτσούκης Νικόλαος

January 25, 2022

Contents

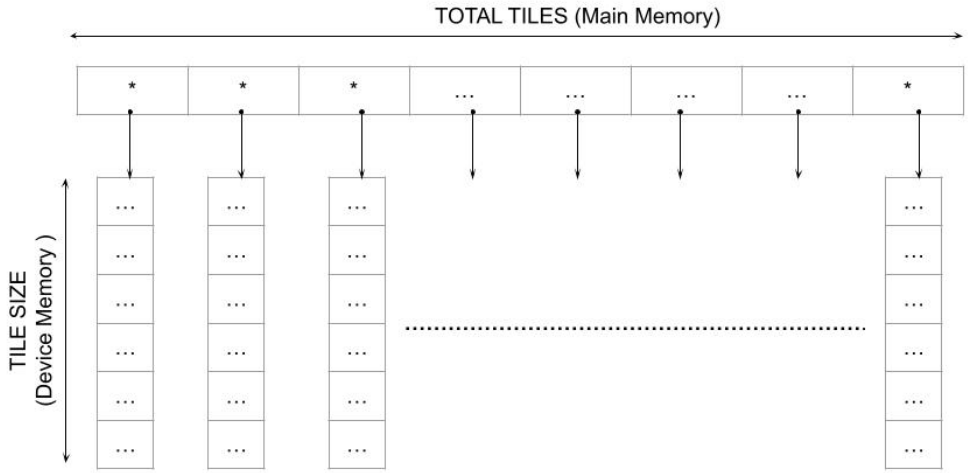
1 Introduction

Given a histogram equalization algorithm that runs sequentially we were tasked to optimize it as much as possible to reduce execution time. The equalization algorithm is actually consisted of three separate parts. The first part of the algorithm a histogram is calculated based on the input PGM image. After retrieving the histogram, a new, equalized, histogram is calculated from the old one. Finally, the image is equalized with the help of the new histogram. In order to optimize the sequential code we must use both our CPU's and GPU's capabilities and thus, a combination of OpenMP with Cuda is used.

2 Memory Structure

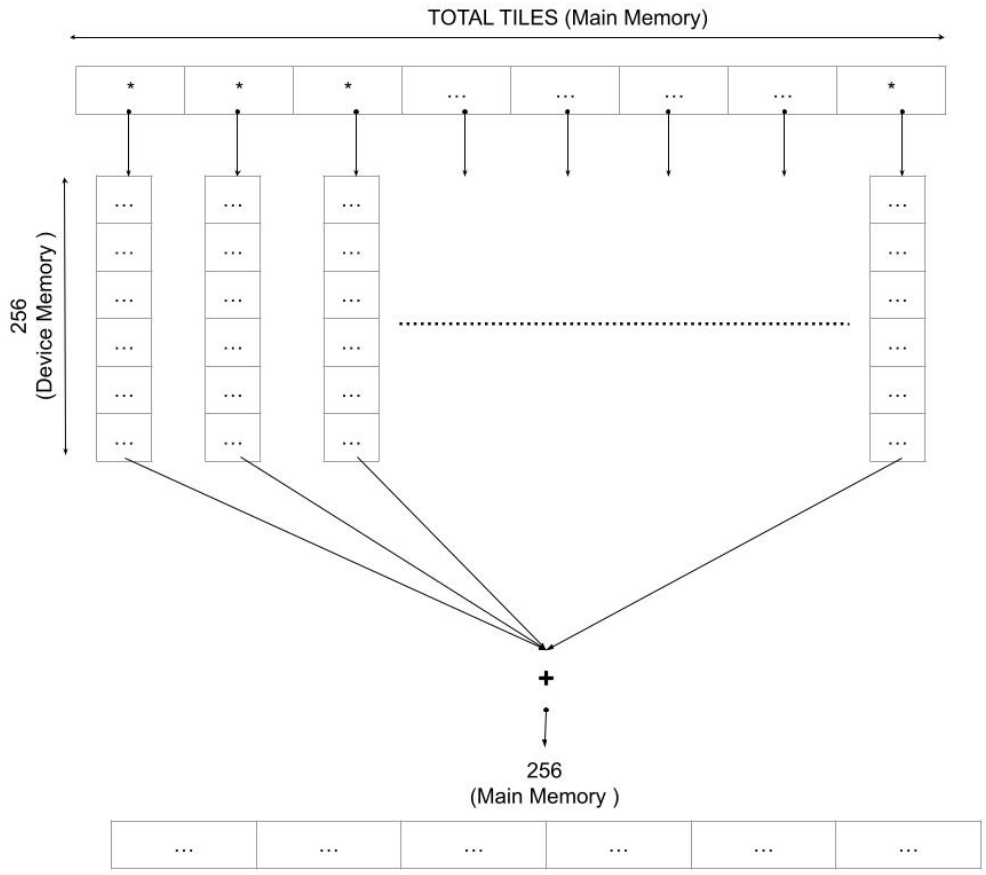
2.1 Tiles

To support infinitely large images we break the the image into multiple tiles. We treat each tile as a separate independent image to perform computations. After all computations have finished the tiles are merged back together and the image equalization is complete. Each tile is stored into the device's memory to reduce memory transfers over the PCIe bus. When the computation is finished, the tiles are pulled back into main memory to be reassembled. An array of pointers to each tile is also stored in main memory to handle memory operations from and to the device. An image of how the tiles are stored can be seen below.



2.2 Calculating the Histogram

Calculating a histogram in the GPU is an easy process but a poor implementation might lead to worse execution times. Many different threads access and update certain sections of memory and without any type of protection this might lead to race conditions and wrong results. When updating these particular regions of memory an atomic operation must be performed to insure that no race conditions might mess up the result. Atomic operations, however, do not allow for parallel execution which might lead to slower execution. Since atomic operations can not be avoided when calculating the histogram in GPU we must try to reduce them as much as possible. This reduction can be easily achieved by offering each tile its own histogram copy to perform computations. Each kernel can perform its own atomic operations on the tile that was tasked to process. This way all kernels can process their own tiles concurrently, with all other kernels, using a kernel private histogram instance. Finally, all private histograms are pulled into main memory and merged into one.



3 Optimizations

3.1 Streams

A certain number of available streams is defined into the header file. We explicitly defined that the maximum number of streams to be used in execution must not be greater than 16. Each tile is handled by an available stream. By using streams kernels can perform operations concurrently as it can be seen below.

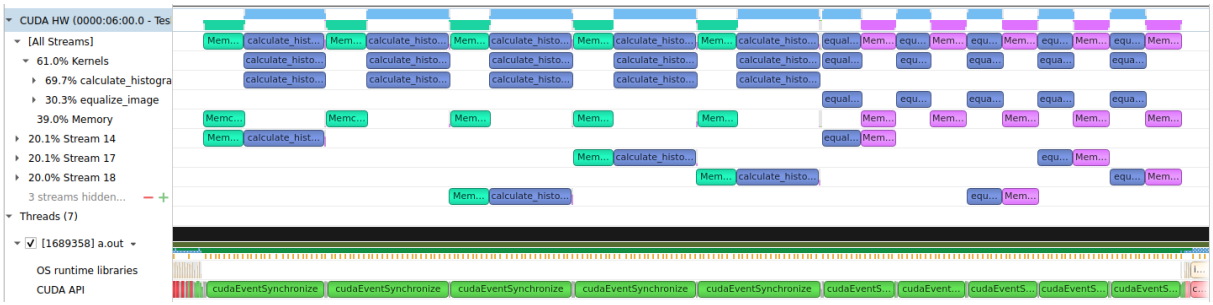


Figure 1: NVIDIA Nsight Systems

3.2 Calculating Histogram

3.2.1 Tiles in Shared Memory

When computing the histogram each tile is pulled into the shared memory. After the tile is pulled the histogram that is located at global memory can be calculated with atomic operations. Since the histogram is located in global memory the occupancy is reduced and the operations are still slow. Tiling with shared memory results in an immense performance boost. However, the overall execution time can reduced even more by increasing the GPU's occupancy.

3.2.2 Histogram in Shared Memory

By pulling the kernel private histograms in shared memory we increase GPU's occupancy while also accelerating atomic operations. After the histogram is calculated in shared memory we must transfer it back to global memory.

3.3 Image Equalization

3.3.1 Histogram in Constant Memory

Since the equalized histogram was extracted we can transfer it to constant memory to reduce memory access latency even more when equalizing the image. Tiles just like before are moved into shared memory to reduce memory access latency and accelerate operations. Tiles are transfered back to global memory when the equalization process is finished.

4 Failed Optimizations

4.1 Unified Memory

Instead of manually handling memory operations we tried to implement an on demand memory model. After implementation the benchmarks did not improve at all and we finally scraped the idea altogether.

4.2 Utilizing CPU with OpenMP

4.2.1 Calculating Histogram

When calculating the histogram the CPU can transfer tiles from main memory to the device, histogram instances from device to host and launch kernels using different streams concurrently. The performance boost making these routines run in parallel is not immense since the execution is bottle-necked by the memory operations latency and not by the actual CPU utilization.

4.2.2 Merging Histograms

When kernel private histograms are retrieved by the CPU they must be merged together into one final histogram. This operation is actually a vector addition between two, at a time, arrays and can be optimized using the full potential of our CPU. However, an addition of 256 sized arrays is not that much of a heavy work even for a single thread, especially when these arrays are few. The actual trade-off here is that the total number of kernel private histograms is equal to the total tiles. When the image is split into more tiles, more kernels are launched but each kernel does not fully utilize the GPU's streaming multiprocessors. In this case the process of histogram merging is favoured since more operations are performed in CPU. On the other side, when fewer tiles are used the GPU is utilized, contrary to the CPU which actually spends more time spawning threads than actually performing calculations. There is a "sweet spot" where the amount of tiles, for huge images, both favours the GPU and the CPU but this is not the general.

4.2.3 Equalizing Image

Just like when we calculated the histogram, kernel launches and tile transfers from device to main memory to be stored into the actual result image can run concurrently. For general cases, the performance boost here is also not immense.

5 Benchmarks

5.1 Small Images

We started testing with smaller images (10MiB) with 8KiB sized tiles. As it can be seen below the the code that uses shared memory outperforms all other implementations.

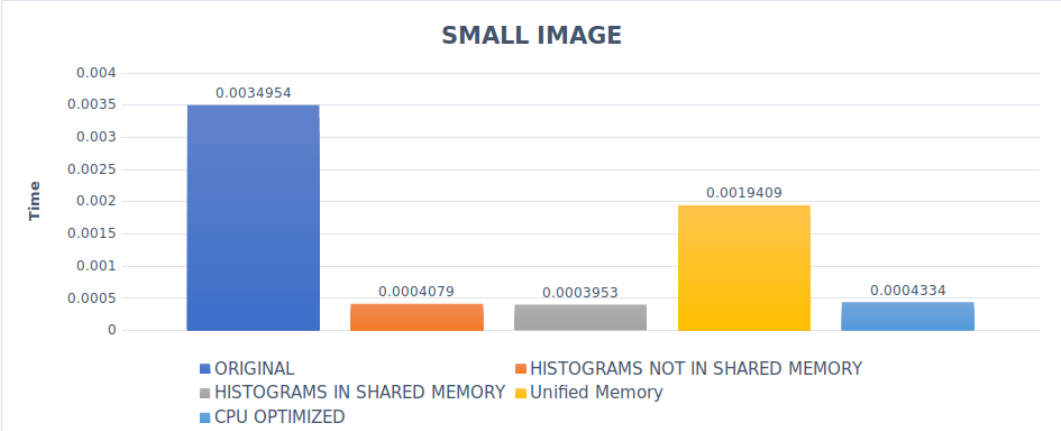


Figure 2: 10MiB Image with 8KiB Tiles

When using unified memory the pipe-lined and asynchronous memory operations and computations are not utilized the execution might block and wait for memory to be retrieved. Thus the increase in time of execution.

5.2 Large Images

When experimenting with larger images (1GiB) and the same tile size (8KiB) we observe that for a 14900% increase in sequential execution time we only have a 1250% increase when the GPU is utilized.

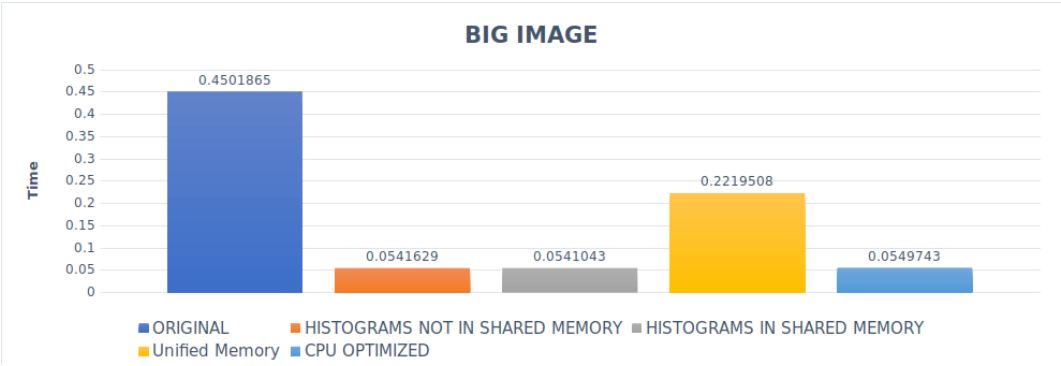


Figure 3: 1GiB Image with 8KiB Tiles

5.3 Notes

If the image was larger and more larger tiles were used the CPU optimized version would surely perform better and most probably outperform all other versions.