;

Department of Electrical and Computer Engineering

University of Thessaly

ECE415: High Performance Computing (HPC)

Lab 4: Parallel Implementation and Optimization of Contrast Enhancement for grayscale images on GPU using CUDA

Ioanna-Maria Panagou, 2962

Nikolaos Chatzivangelis, 2881

Fall Semester 2022-2023

Contents

1 Summary

In this lab, we transformed a sequential CPU implementation of a contrast enhancement algorithm for grayscale images to a GPU implementation using CUDA and performed optimizations to increase GPU utilization and memory bandwidth.

2 Introduction

Histogram equalization is a technique used to improve the contrast in an image. It works by redistributing the intensity values of the pixels in an image such that the intensity histogram of the image is more evenly distributed. This can be particularly useful for images that are poorly lit or have low contrast and is extensively used in medical images, such as x-ray of bone structures.

Our algorithm first calculates the intensity histogram of the image. The histogram has 256 bins, since we are working with gray-scale images and each bin i is filled with the count of how many pixels in the image have an intensity value i. Once the histogram is calculated, the intensity values of the pixels in the image are mapped to new intensity values, which are calculated using the CDF(Cumulative Distribution Function) of the distribution of the histogram. More specifically, we want to normalize the intensity value to the interval [0, 255] and in a way that their distribution resembles a uniform distribution. We will use the cdf and the new value of the image at pixel i will be given by the following equation:

$$new\_value[i] = round(\frac{cdf[i] - cdf[min]}{cdf[max] - cdf[min]} \times 255)$$

, where min is the minimum value that contributes to the cdf (i.e. its histogram bin has a value of at least 1) and $cdf[max]$ is actually equal to the number of pixels in the image, since it is the sum of the counts of all histogram bins.

After we compute the new values and store them in a lookup table, we traverse the image and map each pixel intensity to the new value consulting the lookup table.

In order to make sure our implementation after each optimization is correct, we compare the output of the GPU with the output image of the CPU sequential algorithm and make sure that the PSNR is infinite.

The remainder of our report is structured as follows: section 3 illustrates the process we followed to evaluate our optimizations, sections 4 presents our proposed optimizations steps and section 5 concludes the report.

3 Evaluation

We ran experiments for all the provided images and included two additional images, dog.pgm ($4896 \times 3264 = 152$MBytes) and universe.pgm ($14575 \times 8441 = 117.3$MBytes). The decision that an optimization was effective was made based on the execution time of the largest available image, universe.pgm, however, we feature results for the other images as well. Every successful optimization is incorporated to the previous version of the code, i.e. each version contains all the previous successful optimizations unless noted otherwise.

4 Optimization Steps

4.1 CPU Baseline

The CPU sequential algorithm performed relatively fast, even for the largest image. Figure 1 presents the CPU execution time for all images. The code traverses the input array in a row-wise fashion once for the histogram computation and once for the histogram equalization, so the complexity of the algorithm is $O(N)$, where N is the size of the input image. Figure 2 confirms our assumption that the complexity and, therefore, the execution time forms a linear dependence with the input array.

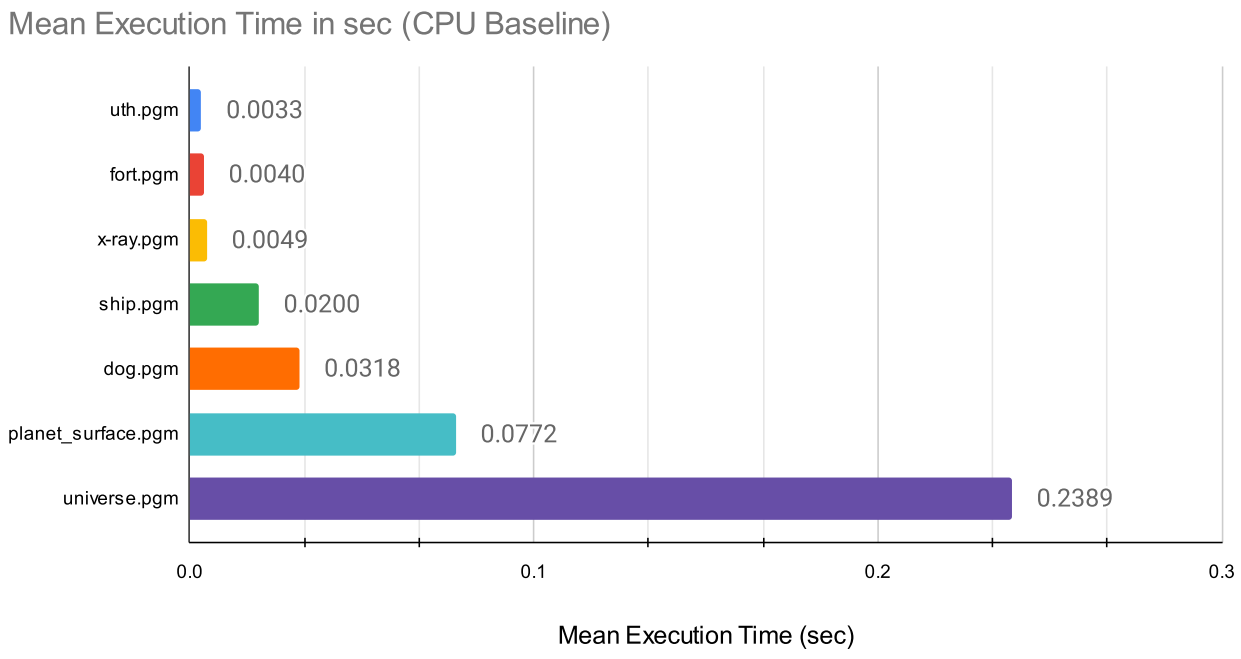

Figure 1: Mean Execution Time (sec) for sequential CPU implementation

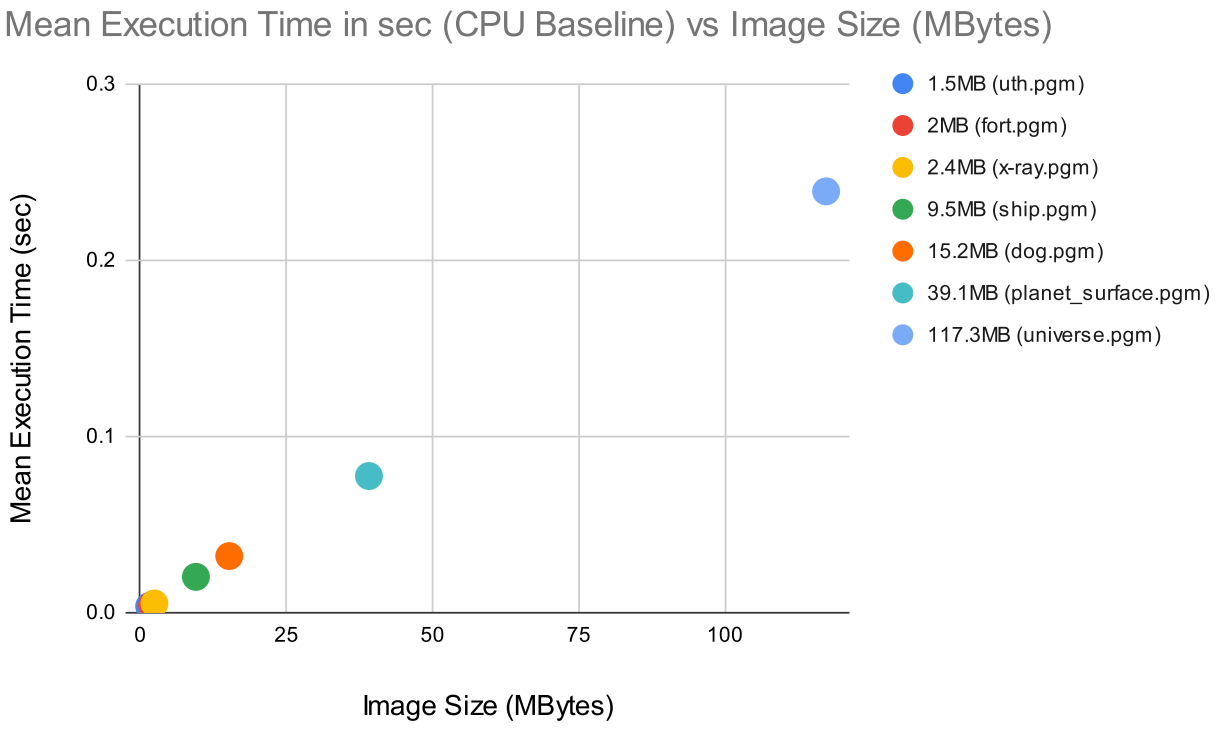

Figure 2: Mean Execution Time (sec) for sequential CPU implementation vs Image Size (MBytes).

4.2 Histogram Calculation Optimizations

4.2.1 GPU Histogram (1024 threads per block)

Our first attempt at optimizing the code was to transfer execution of the histogram to the GPU. We decided that we would use a 1D geometry for both the grid and the threads within each block, where each block would have a maximum of 1024 threads and we would have $\left\lceil \frac{width \cdot height}{1024} \right\rceil$ blocks of threads or 2147483647 blocks (max grid size in dimension x) blocks, in case the image is so large that exceeds this number. Listing 1 presents the GPU implementation of the histogram kernel. We have padded the histogram with zeros in the CPU, so that its total size is equal to 1024, in order to avoid warp divergence in the histogram initialization in line 11 and in line 23. Since each element of the image is only read once and is not reused throughout execution of the kernel, we did not find the need to transfer it into shared memory, because that would result in 2 reads, one from the global and one from the shared memory. On the other hand, the histogram does not have a regular access pattern and is quite small, so we figured that each block could work on its own private copy of the histogram that is kept in shared memory and after all the blocks have finished updating their copies, their partial sums are accumulated on the histogram that is kept in global memory. In this way, we reduce the worst number of threads that contend for an atomic operation on the same memory location in line 16 from img_size to $blockDim.x$.

```
__global__ void histogram_gpu(int *hist_out, unsigned char *img_in, int
    img_size, int nbr_bin) {

    int i = threadIdx.x + blockDim.x * blockIdx.x;
    extern __shared__ int shared[];
    int *private_hist = &shared[0];

    // stride is the total number of threads
    int stride = blockDim.x * gridDim.x;

    // all threads wait for private hist initialization
    private_hist[threadIdx.x] = 0;
    __syncthreads();

    while ( i < img_size) {
        int val = img_in[i];
        atomicAdd(&(private_hist[val]), 1);
        i += stride;
    }

    // Wait for all blocks to finish their private histogram computation
    __syncthreads();

    atomicAdd(&(hist_out[threadIdx.x]), private_hist[threadIdx.x]);

}
```

Listing 1: GPU Histogram Implementation (using 1024 threads per block)

4.2.2 GPU Histogram (256 threads per block)

Using the CUDA Occupancy Calculator, we found out that are configuration was suboptimal. Compiling with the option `"-ptxas-options="-v""`, we saw that we are using 7 registers per thread and we know that the amount of shared memory per block is $4 \cdot 1024 = 4096$ bytes for the padded histogram. This results in an occupancy of 67% of each multiprocessor as shown in figure 3. If we decide not to pad the histogram and use 256 threads instead, we can see that we achieve maximum multiprocessor utilization (figure 4).

2.) Enter your resource usage:	
Threads Per Block	1024
Registers Per Thread	7
User Shared Memory Per Block (bytes)	4048

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	1024
Active Warps per Multiprocessor	32
Active Thread Blocks per Multiprocessor	1
Occupancy of each Multiprocessor	67%

Figure 3: Occupancy of each Multiprocessor using 1024 threads per block

2.) Enter your resource usage:	
Threads Per Block	256
Registers Per Thread	7
User Shared Memory Per Block (bytes)	1024

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	1536
Active Warps per Multiprocessor	48
Active Thread Blocks per Multiprocessor	6
Occupancy of each Multiprocessor	100%

Figure 4: Occupancy of each Multiprocessor using 256 threads per block

Furthermore, the size of the global memory is 11997020160 Bytes, so we need a maximum of 11997020160/256=46863360 blocks, which is way less than the maximum number of blocks in dimension x of a grid (11997020160), so we are sure that there are enough threads for each thread to handle one element of the input image and therefore the while clause in line 14 of listing 1 is redundant and can be replaced with just an if clause that checks if the thread id is within the image borders. The modified code is presented in listing 2.

```
1  __global__ void histogram_gpu(int *hist_out, unsigned char *img_in, int
       img_size, int nbr_bin) {
2
3      int i = threadIdx.x + blockDim.x * blockIdx.x;
4      extern __shared__ int private_hist[];
5
6      private_hist[threadIdx.x] = 0;
7      __syncthreads();
8
9      if ( i < img_size) {
10         int val = img_in[i];
11         atomicAdd(&(private_hist[val]), 1);
12     }
13
14     __syncthreads();
15
16     atomicAdd(&(hist_out[threadIdx.x]), private_hist[threadIdx.x]);
17
18 }
```

Listing 2: GPU Histogram Implementation (using 256 threads per block)

4.2.3 Results

Figure 5 presents the execution time for the CPU, GPU with 1024 threads per block and GPU with 256 threads per block implementations the execution time for the histogram calculation. We can see that the GPU implementation with 1024 threads works better than the CPU baseline for the large images, but the GPU implementation with 256 threads achieves the a speedup of up to x3, which increases as the images get larger, compared to the CPU sequential

implementation.

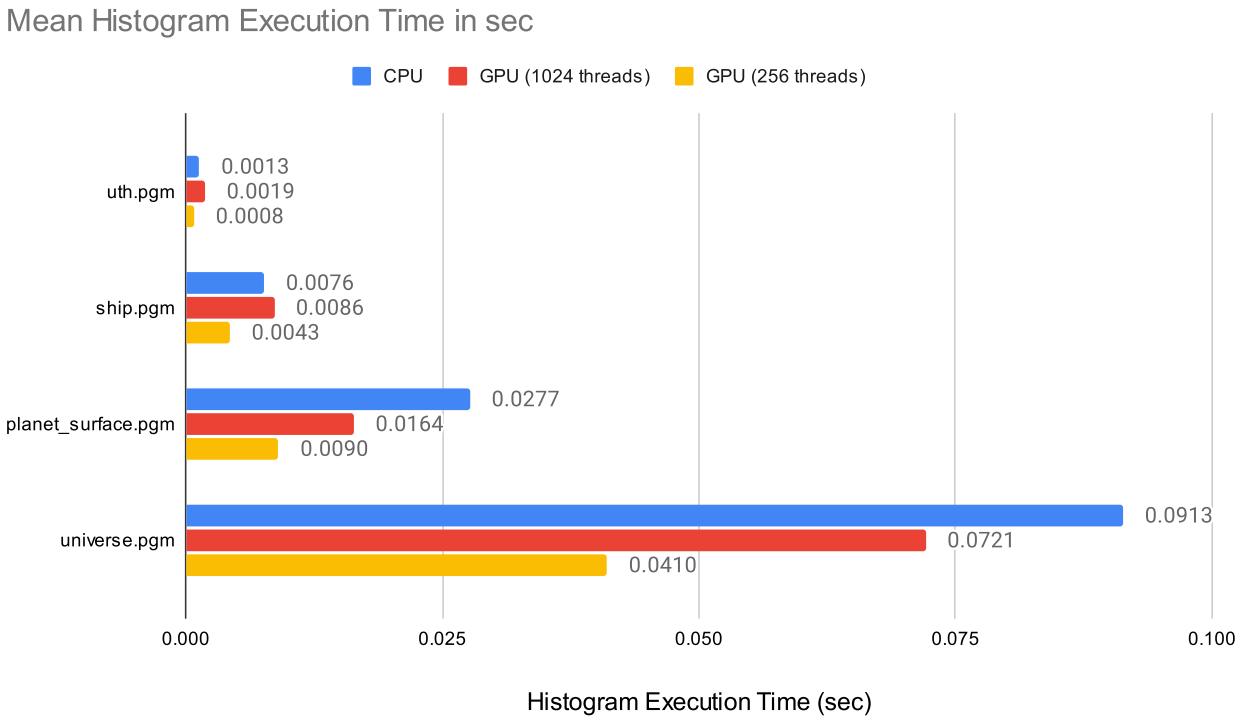

Figure 5: Mean Execution Time for histogram computation (sec)

Figure 6 presents the mean total execution time for the configurations described above. We can see that the version with the 256 threads outperforms the CPU baseline implementation, except for the smallest image, uth.pgm. One could ask why the overall execution time is higher, when the histogram execution time is lower. The answer is illustrated in fig.7. Computing the histogram in the GPU requires allocating memory on the device for the input image and histogram (GPU Memory Allocation), transferring the input image data from the CPU to the device (GPU Input Transfers) and copying the histogram from the device to the host CPU (GPU Output Transfers). So, even though the execution time for the histogram calculation has been reduced, the overhead added can overpower the speedup achieved in one kernel, especially when the performance in the particular kernel is small.

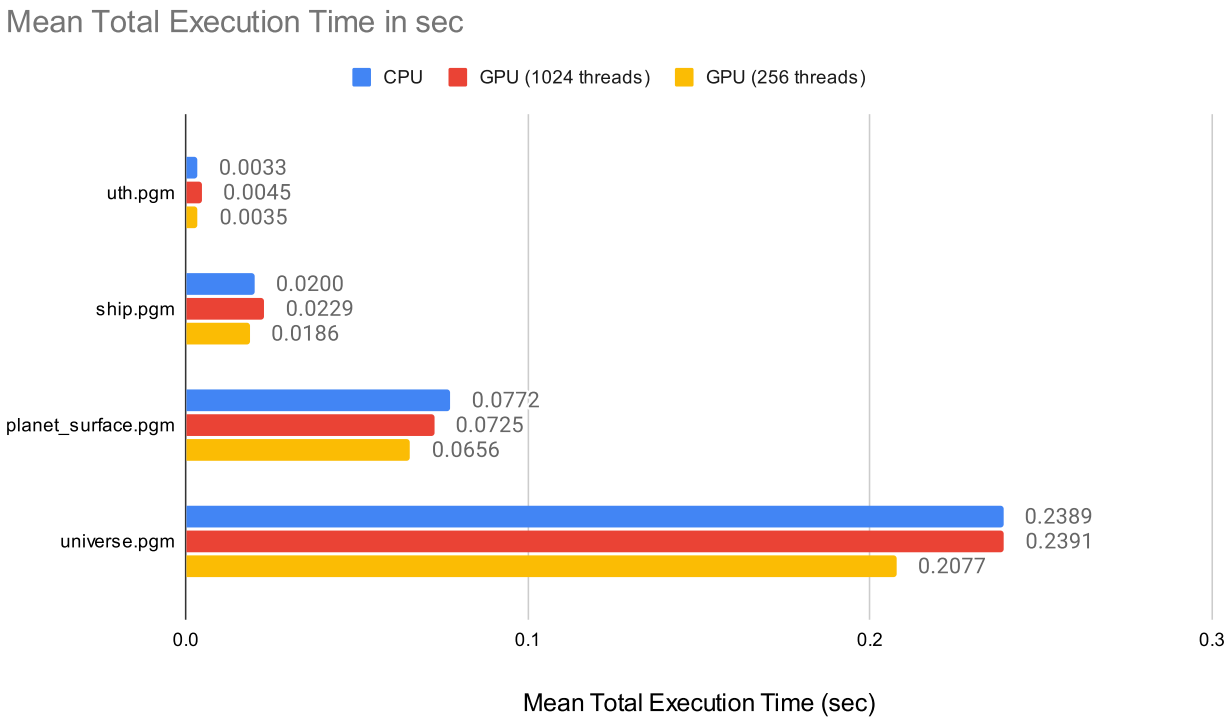

Figure 6: Mean Total Execution Time (sec)

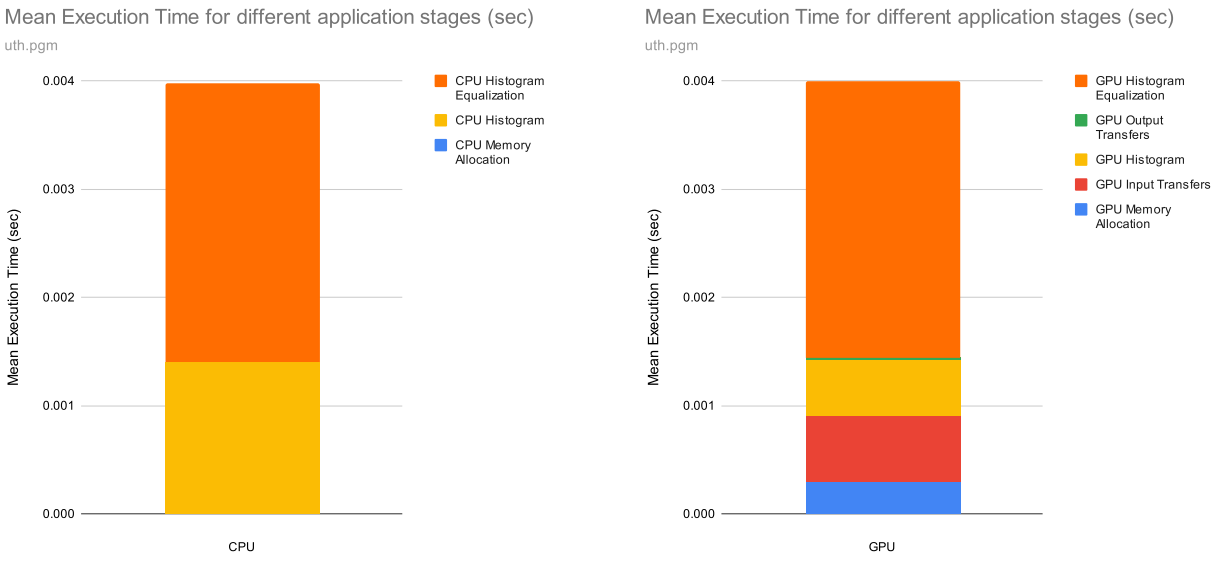

Figure 7: Mean Execution Time for different application stages for the uth.pgm image. We can see that compared with the baseline CPU approach on the left, the GPU implementation on the right has some additional steps that make up for a significant portion of the execution time. The GPU implementation total execution time is slightly higher, even though it is not apparent in the figure.

4.3 Histogram Equalization Optimizations

The histogram equalization is comprised of two steps:

1. The computation of the look up table using the cdf of the intensity histogram
2. Updating the result image using the look up table

We will split the histogram equalization kernel to two separate kernels that perform the above steps.

4.3.1 CDF Naive Approach

For the cdf computation kernel, we calculate a reduction tree as shown in figure 8. The algorithm is an in-place scan algorithm that operates on the input vector, which also contains the final result. Let x be the name of the input-output vector of size N (recall that the algorithm is an in-place scan). In the first iteration, each position except $cdf[0]$ receives the sum of its current content and that of its left neighbor. So, position i is equal to $x[i-1] + x[i]$. In the second iteration, all position except $x[0]$ and $x[1]$ receive the sum of their current content and the element that is two positions away. At the end of this step, position i is equal to $x[i] + [i-2] = x[i] + x[i-1] + x[i-2] + x[i-3]$. Continuing in this fashion, one can easily see that at the end each iteration n, $n \in \{1, 2, \ldots log_2(N)\}$, the elements up to indices 2^{n-1} have the final values and the other positions contain the sum $\sum_{k=i-2^n+1}^{i} x_k$. We will need in total $log_2 N$ iterations. Listing shows the CUDA kernel for the cdf computation. We will be using one block of 256 threads. In line 5, each thread copies one element of the histogram to the vector for the cdf, which is stored in shared memory. In lines 8-10, we search for the minimum element that contributes to the cdf. Lines 14-17 perform the actual computation and line 19 calculates and stores the result to the look up table, which resides in global memory.

4.3.2 CDF Better Work Efficiency Approach

A sequential implementation would scan the vector from left to right and perform an addition in every step, thereby having a computational complexity of $O(N)$. Our parallel implementation performs $log_2(N)$ iterations and in each iteration n, $N - 2^n$ additions are performed, thus resulting in a complexity of $\sum_{n=1}^{log_2(N)} (N - 2^n) = Nlog_2(N) - \frac{1(1-2^{log_2(N)})}{1-2} = Nlog_2(N) - \frac{1-N}{-1} = Nlog_2(N) - N + 1 = O(Nlog(N))$. For a vector with $N = 256$, the naive parallel algorithm does approximately 8 times more work than the sequential, albeit in parallel. We can further reduce the computational steps by employing a reduction tree method as shown in figure 9. The computation happens in two stages:

1. The first step consists of $log_2(N)$ steps. At each iteration n, the element in positions that are multiples of $i = k \cdot 2^n - 1, k \in \{1, 2, 3, \ldots\}, i < N$ have the sum $\sum_{k=i-2^n+1}^{i} x[k]$. This

step performs $\sum_{i=1}^{log_2(N)} \frac{N}{2^i} = N \sum_{i=1}^{log_2(N)} \left(\frac{1}{2}\right)^i = \frac{N}{2} \sum_{i=0}^{log_2(N)} \left(\frac{1}{2}\right)^i = \frac{N}{2} \frac{1-2^{log_2(\frac{1}{n})}}{1-\frac{1}{2}} = N(1-\frac{1}{N}) = N-1$ additions.

2. The second step distributes the partial sums to the positions that can use these values as quickly as possible. At the end of the first step, position $2^n - 1 = N, N/2 - 1, N/4 - 1, N/8 - 1, \ldots$ have the final value ($i = 2^n - 1$, so the summation described in the first step starts from zero). We obtain the final sums gradually. The element in position $3N/4 - 1$ can get the partial sum that needs from the position $N/4$ away ($N/4 - 1$). Then, the elements in $3N/8 - 1, 5N/8 - 1$ and $7N/8 - 1$ can obtain the partial sums they need from positions $N/8$ away ($N/8 - 1, N/4 - 1, 3N/4 - 1$) and so on. This step results in less than $N - 1$ operations, which was the number of additions in the previous step, and, thus, the entire algorithm performs less than 2 times the operations of the sequential algorithm, a cost easily amortized by the number of computing resources in the GPU.

The code for the work-efficient prefix sum is featured in listing

```
__global__ void compute_cdf_gpu(int *lut, int *hist_in, int img_size, int
    nbr_bin) {
    int min = 0, min_index = 0, d;
    __shared__ int cdf[BINS];

    cdf[threadIdx.x] = hist_in[threadIdx.x];
    __syncthreads();

    while (min == 0) {
        min = cdf[min_index++];
    }

    d = img_size - min;

    for (unsigned int stride = 1; stride <= threadIdx.x; stride*=2) {
        __syncthreads();
        cdf[threadIdx.x] += cdf[threadIdx.x - stride];
    }

    lut[threadIdx.x] = max((int)(((float)cdf[threadIdx.x] - min)*255/d +
    0.5), 0);
}
```

Listing 3: `Naive GPU CDF Computation`

```
__global__ void compute_cdf_gpu(int *lut, int *hist_in, int img_size, int
    nbr_bin) {
    int min = 0, min_index = 0, d;
    __shared__ int cdf[BINS];

```

```
    cdf[threadIdx.x] = hist_in[threadIdx.x];
    __syncthreads();

    while (min == 0) {
        min = cdf[min_index++];
    }

    d = img_size - min;

    // First step
    for (unsigned int stride = 1; stride < blockDim.x; stride*=2) {
        __syncthreads();
        int index = (threadIdx.x + 1) * 2 * stride - 1;
        if (index < blockDim.x) {
            cdf[index] += cdf[index - stride];
        }
    }

    // Second step
    for (int stride = BINS/4; stride > 0; stride /=2) {
        __syncthreads();
        int index = (threadIdx.x + 1) * stride*2 - 1;
        if (index + stride < BINS) {
            cdf[index + stride] += cdf[index];
        }
    }
    __syncthreads();

    lut[threadIdx.x] = max((int)(((float)cdf[threadIdx.x] - min)*255/d +
   0.5), 0);

}
```

Listing 4: Work-effiecient GPU CDF Computation

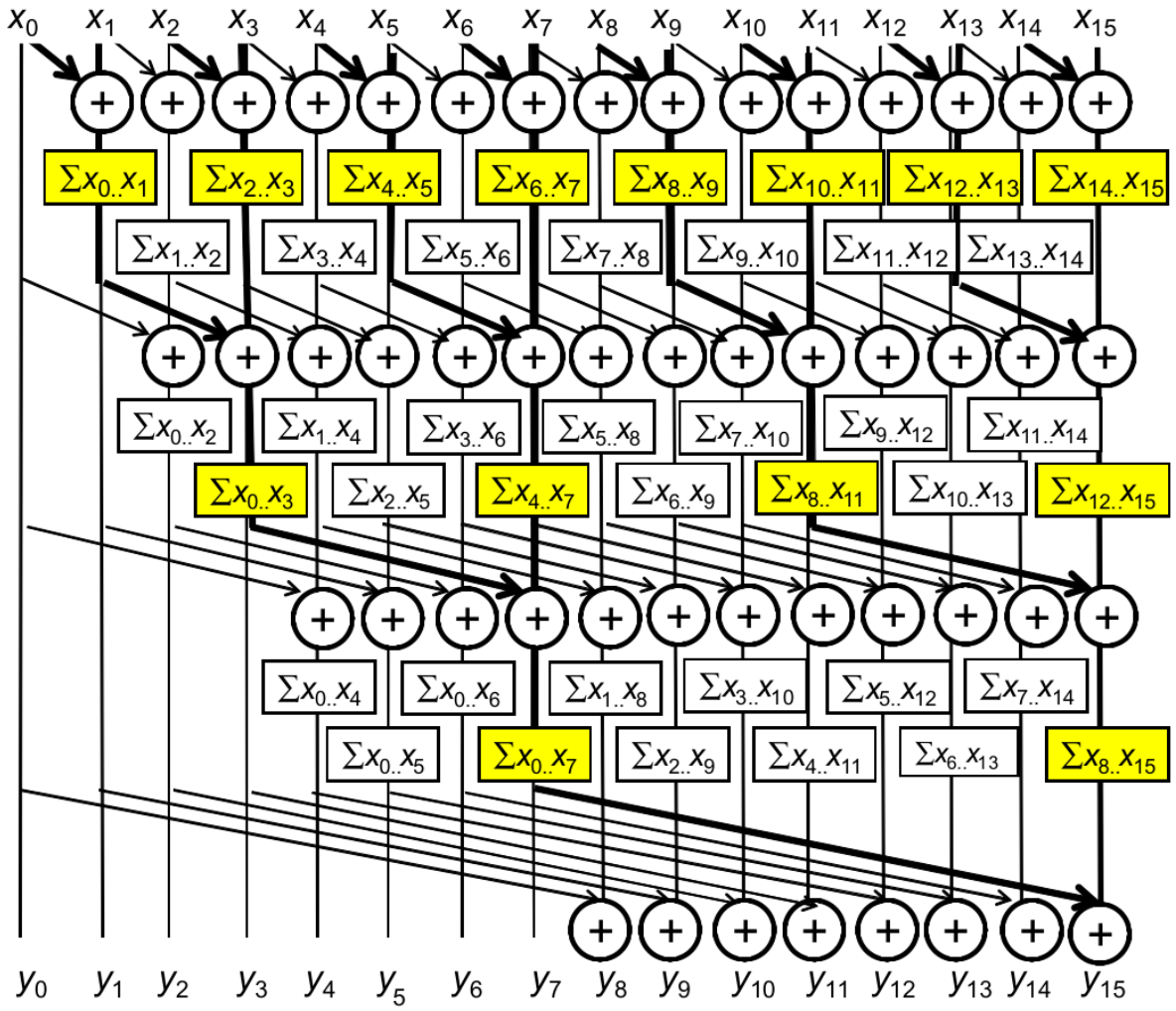

Figure 8: Naive reduction tree for cdf computation

4.3.3 CDF Results

Figure 10 presents the execution time for the histogram equalization using the naive CDF and work-efficient CDF. Despite the work-efficient implementation being more efficient in theory, the synchronization barriers and multiple for loops have overpowered the fewer calculations and resulted in an execution time that is only slightly less than the naive implementation. We decided to keep the work-efficient implementation, however, due to the less variation in the results (standard deviation of 0.55ms for the naive vs 0.26ms for the work-efficient for the universe.pgm).

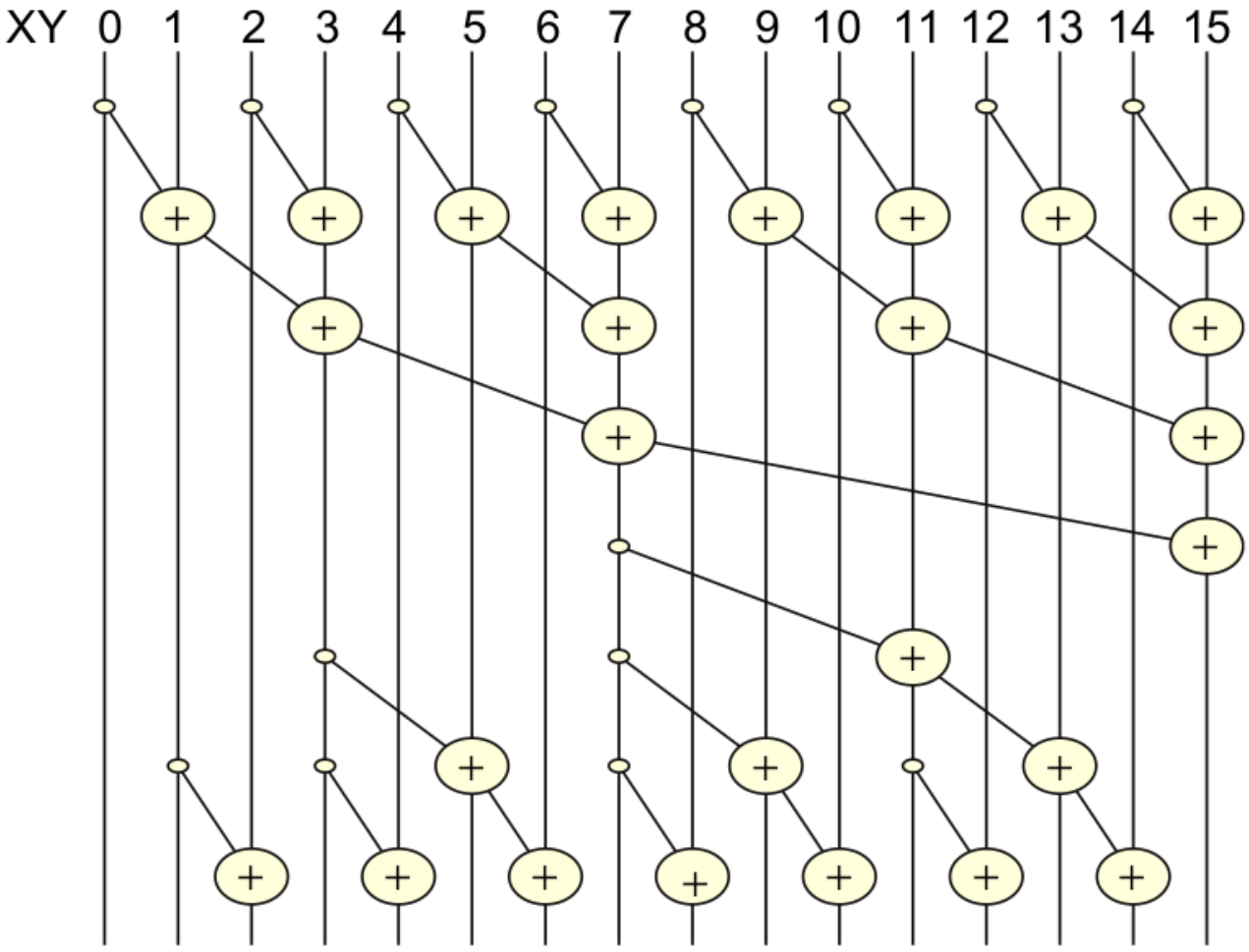

Figure 9: Work-efficient reduction tree for cdf computation

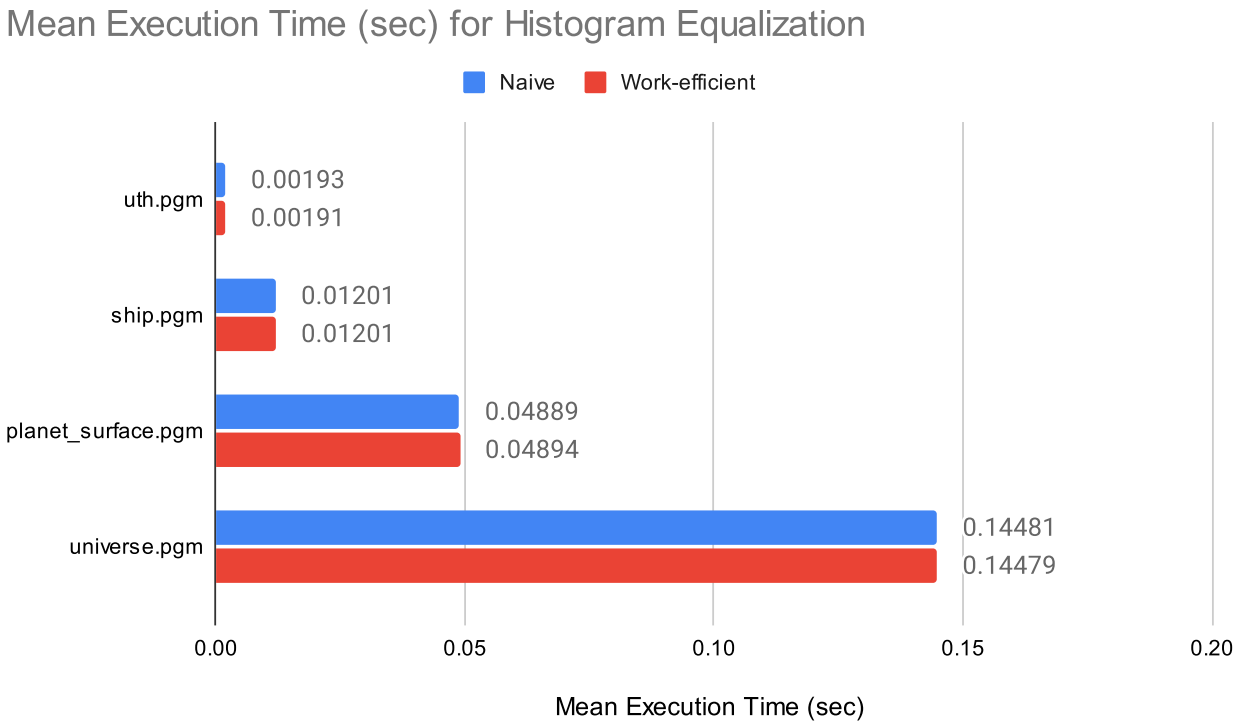

Figure 10: Histogram Equalization Mean Time (sec)

4.3.4 Histogram Equalization on GPU

The second step of the histogram equalization process is the actual replacement of values in the output image. We transferred this function to the GPU using the code listed in 5. The new histogram is the look up table calculated by the cdf function. Since it is frequently access in a non-regular memory access pattern, it is imperative that is first brought into the shared memory by each block of threads. Consulting the CUDA occupancy calculator, we decided to use blocks of 256 threads, so that no thread is idle during copying the histogram and since using the maximum number of threads in a block (1024) results in an under-utilization of the streaming multiprocessors.

```
__global__ void histogram_equalization_gpu(unsigned char *img_out, unsigned
    char *img_in, int *hist_in, int img_size) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;

    extern __shared__ int private_hist[];

    private_hist[threadIdx.x] = hist_in[threadIdx.x];
    __syncthreads();

    img_out[i] = min(255, private_hist[img_in[i]]);
}
```

Listing 5: GPU Histogram Equalization

4.3.5 Results

Here are the results of the GPU Implementation of the Histogram Equalization.

Figure 11 presents the mean execution time for the histogram equalization for the cpu sequential and the gpu parallel implementation. As we also see from figure 12, we gained a significant speedup of around x17 for the larger images. Figure 13 presents the mean total execution time for the baseline CPU implementation, the previous successful optimization (the histogram calculation on the GPU) and the histogram equalization on the GPU. The execution time significantly improved.

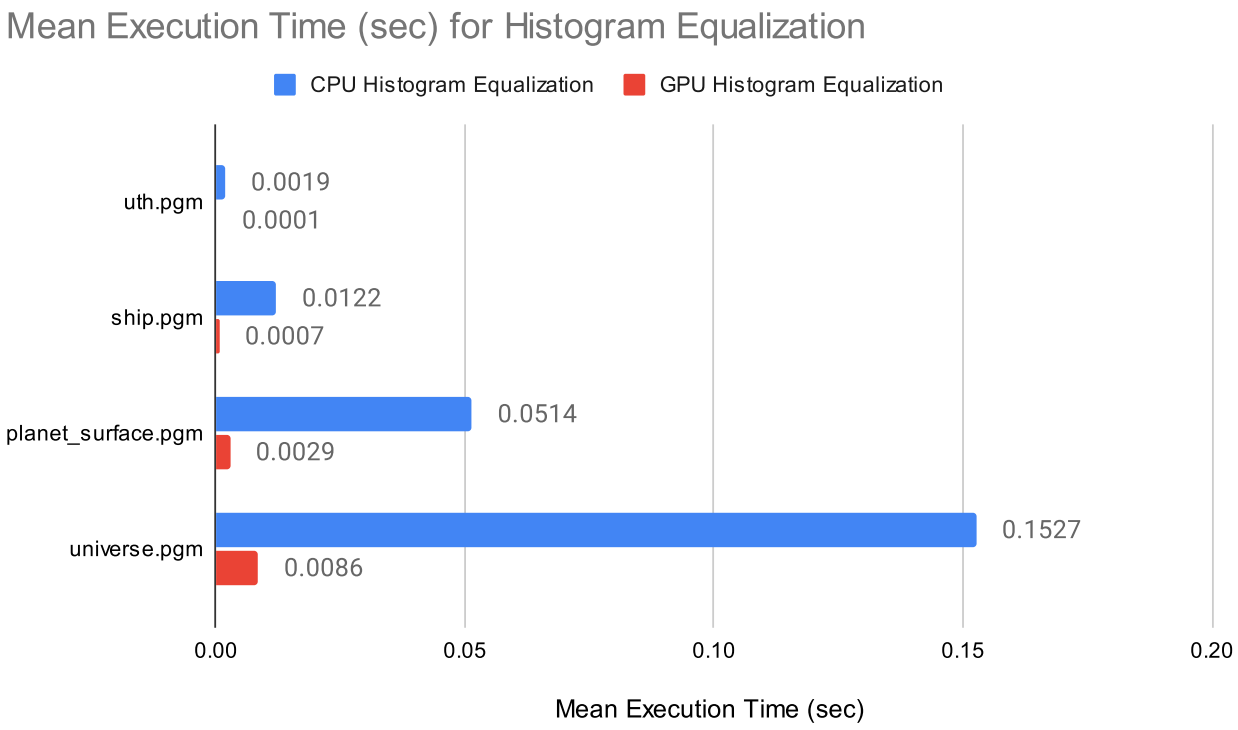

Figure 11: Mean Execution Time for Histogram Equalization

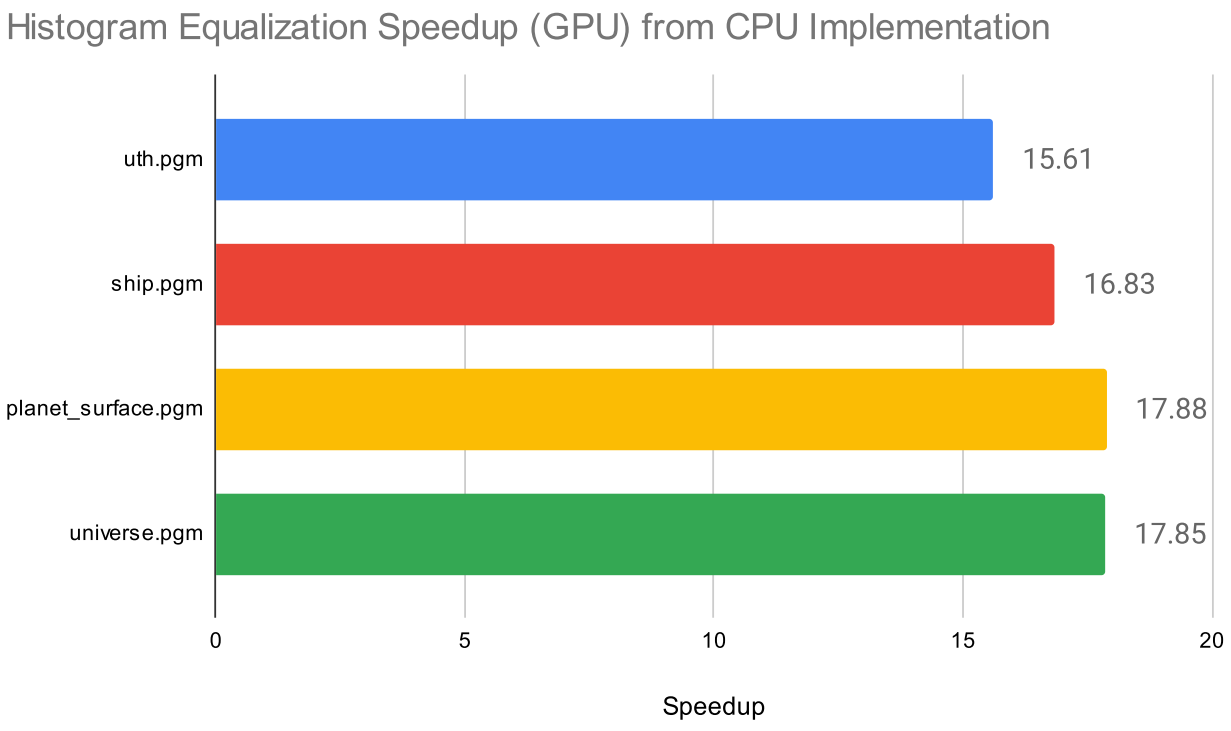

Figure 12: Speedup from CPU Histogram Equalization Implementation

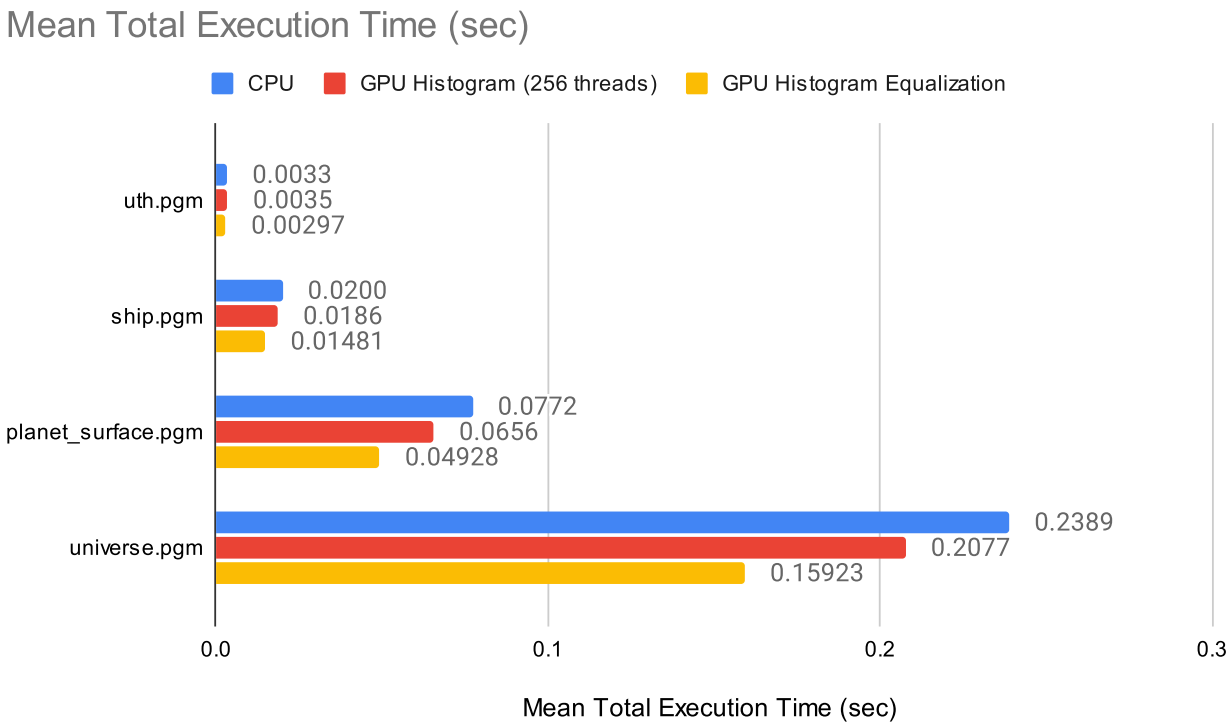

Figure 13: Mean Total Execution Time (sec)

4.4 Page-Locked Memory Allocation

Figure 14 shows that copying the output image back to the host CPU takes up more than 50% now that we have optimized all kernels, which indicates that we must be very conscious about the memory transfers happening under the hood, which transfers have significant execution time penalties. Thus, it is mandatory to investigate smarter memory allocations or techniques in order to minimize the introduced delay.

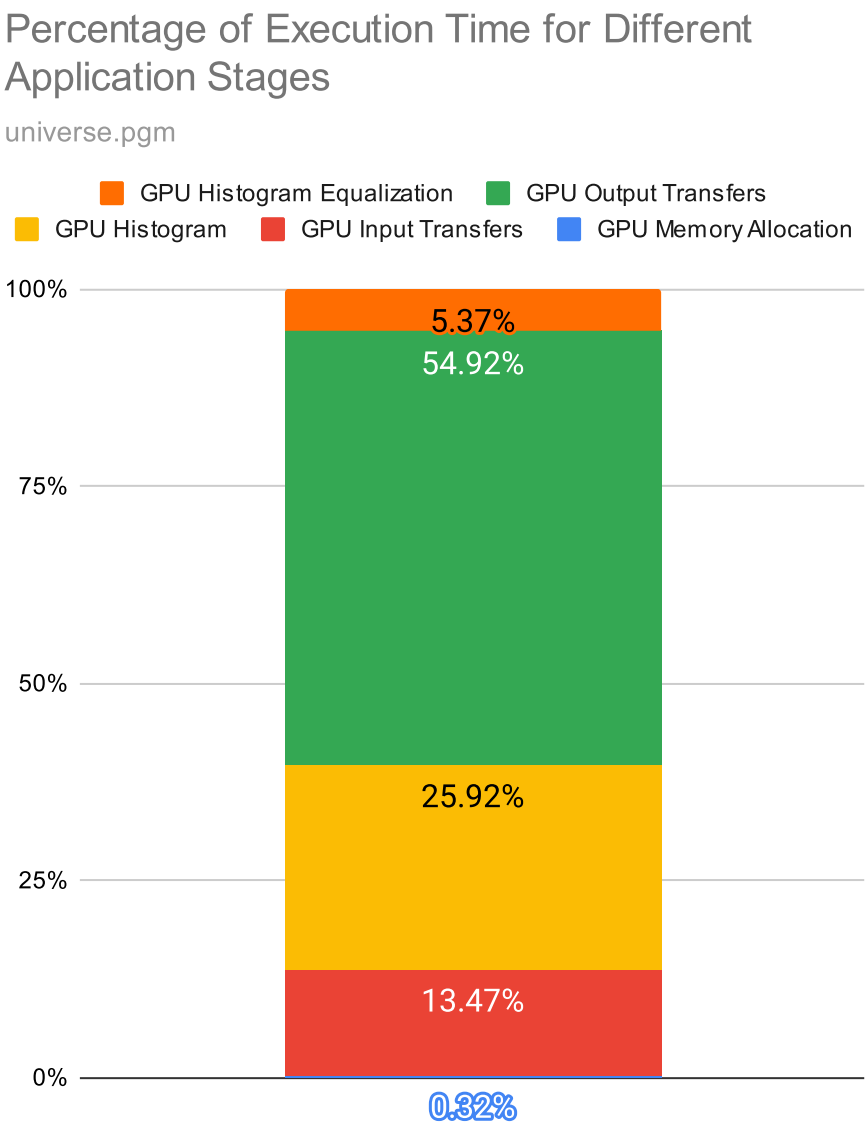

Figure 14: Percentage of the Total Execution Time for the GPU implementation of the previous section for different application stages

We know that the GPU cannot access data directly from host pageable memory. The host data must first be copied to the pinned array, then transfer the data from the pinned array to the device. This is a 2-step process, introducing overhead because the pinned array must be first allocated, then the data must be copied to it and finally reach the GPU. So in order

to avoid this inconvenience, CUDA offers `cudaMallocHost()`, a mechanism to allocate host memory directly to the pinned array eliminating the aforementioned process delay.

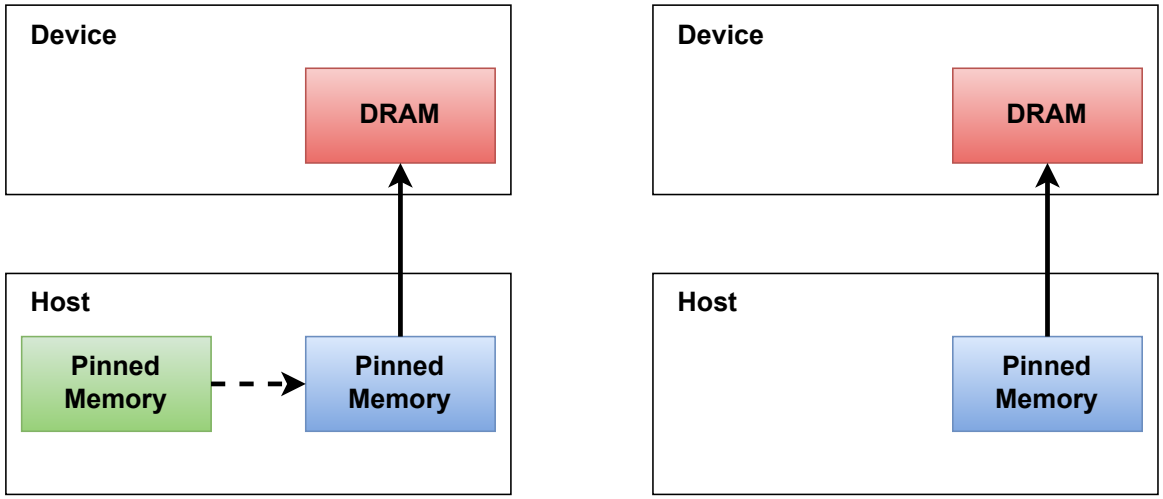

Figure 15: On the left we can see the 2-step data transfer procedure and at right we can see how this procedure changes when using `cudaMallocHost()`

We experimented with allocating the input and output image with cudaHostAlloc with two combinations of parameters:

1. cudaHostAllocDefault: If the host memory was not pageable, we expect to see an improvement in the input and output data transfers.

2. cudaHostAllocMapped + cudaHostAllocWriteCombined: The first flag maps the memory allocated to the CUDA address space and eliminates the need for explicit memory copies (the data is still moved through the PCI pipe, albeit in a transparent fashion). The second flag is used to speedup transfers over the PCI Express bus.

4.4.1 Results

Figure presents the improvement in the speed of memory copies using cudaHostAlloc (with the default flag), especially for the output transfers, which indicates that the output image was not pageable at that time. The Mean Total Execution time (sec) with and without cudaHostAlloc is presented in figure 17. Using the default flag for the cudaHostAlloc yielded significant improvement in the execution time, whereas the Mapped + WriteCombined flag performed poorly due to the many transfers over the PCI bus midst kernel execution.

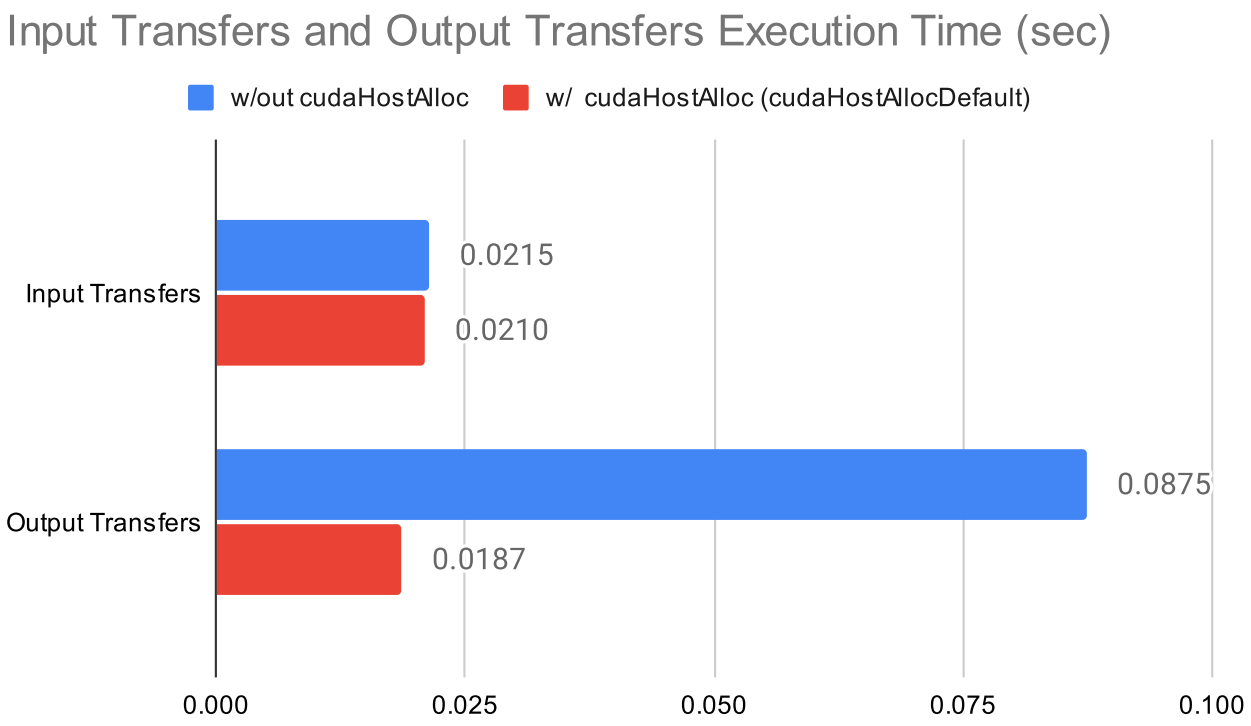

Figure 16: Mean Execution time (sec) for Input and Output Transfers

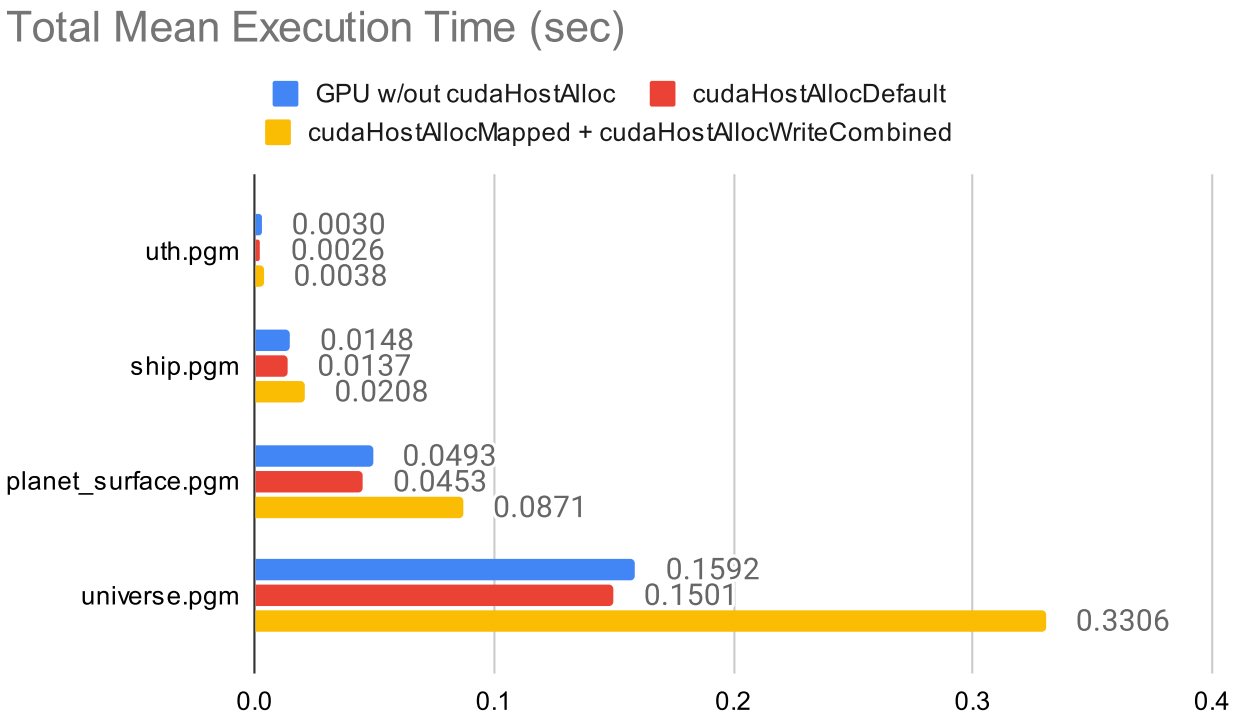

Figure 17: Mean Total Execution Time (sec)

4.5 CUDA Streams

Since our GPU supports concurrent copy and kernel execution, we thought that we would take advantage of CUDA streams.

We created two streams to enable overlapping memory copies and kernel execution. We split the input image to two device vectors. Once the first half is transferred by stream0 to the device, we execute the histogram calculation kernel for the first half of the input image. The other half of the input image is transferred to the kernel simultaneously. Then, we execute the histogram calculation kernel for this second half. Both kernels perform reduction on the same histogram. After the cdf computation, in a similar fashion, we update the first half of the image with the new values, executing the histogram equalization kernel. Then, the first half is transferred back to the CPU and the histogram equalization updates the second half of the image in parallel. Finally, the second half of the image is transferred back to the host. Figure presents the timeline of the above description.

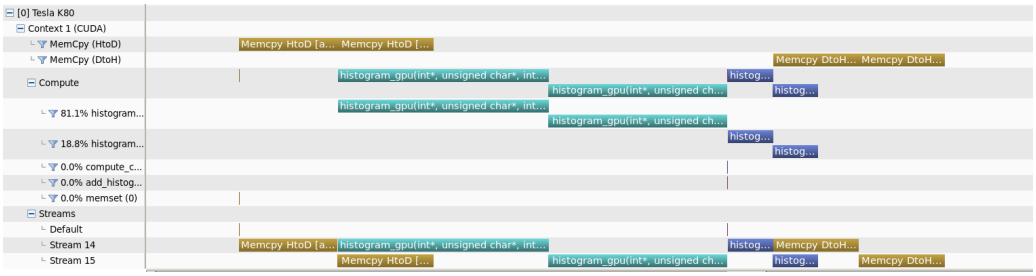

Figure 18: Timeline of the application execution using 2 streams

4.5.1 Results

Figure 19: Mean Total Execution Time (sec) for the previous and current optimization

All the images, except for the smallest one, uth.pgm, reported a reduction in execution time.

4.6 GPU Caches

To further reduce the execution time, we thought that we could improve the bandwidth from read-only data by storing them in read-only memory. The two available read-only memories we have at our disposal are constant and texture memory. Their size is small though (only 65536) bytes and therefore we cannot transfer the input image in read-only memory and we cannot write on read-only memory again the image in tiles. So, we decided to move the histogram and the look up table after their computation to the constant or texture memory. This way, instead of reading the element from global memory, we now read them from one of the caches (constant or texture), while paying the cost of copying to constant memory or binding to texture memory.

4.7 Results

The results from figure 20 indicate that the transferring the histogram and look up table to the constant memory did not aid performance. Moving them to texture memory slightly improved execution time for all images, so we can consider it a successful optimization.

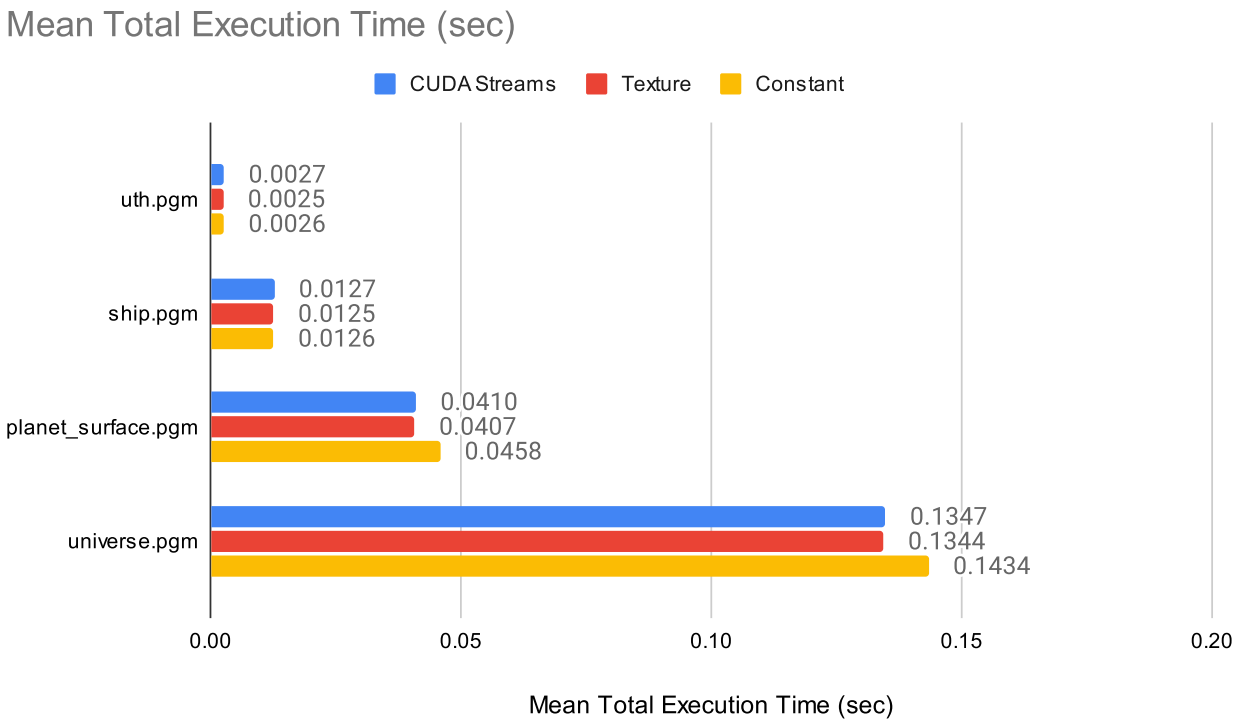

Figure 20: Mean Total Execution Time in (sec)

4.8 Unified Memory

Unified memory is a memory address space accessible from both the CPU and GPU. This technology allows us to allocate data that can be accessed and modified from code running on either the host or the device. It offers simpler programming and memory models, as well as performance enhancement through data locality. Although data are still being transferred from and to the device through the PCI-E interface, as long as pages are accessed in order prefetching mechanisms increase the code performance.

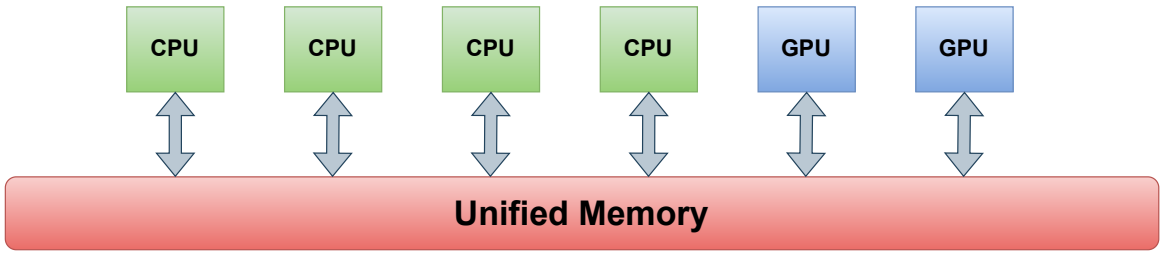

Figure 21: Unified memory demonstration

During the attempts to optimize the code, we allocated the resulting image in the unified memory using the `cudaMallocManaged()` CUDA function. So now there was no need to perform any memory copies from the device to the host to obtain the resulting image. We did not allocate the input image in the unified memory, because it's used by two kernels (histogram_gpu and

histogram_equalization_gpu) so data would potentially need to be transferred twice.

Figure 22 present the mean execution time for the GPU with textures and the three attempts we made to exploit unified memory. The first attempt was to allocate only the resulting image in the unified memory (*Malloc Managed (output)*). The second attempt was to allocate both the input and the resulting images using `cudaMallocManaged()` (*Malloc Managed (input + output), no Streams*). Finally, the last version optimization was to invoke streams along with unified memory (*Malloc Managed (input + output) + Streams*). As we can see from the diagram below, the unified memory improved the overall execution time of the program. Moreover, these implementations run faster than the previous one with the texture, but the optimal performance is achieved when only the output is in unified memory and the input image is pinned. So this concluded our final optimization.

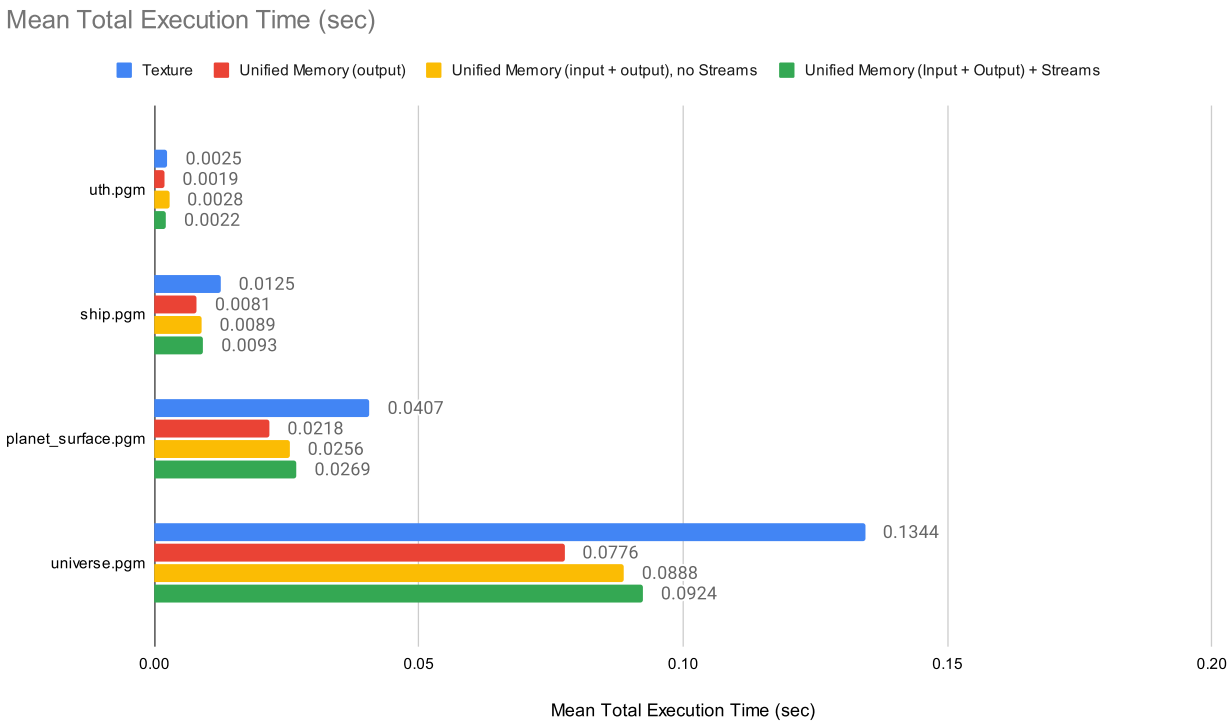

Figure 22: Total Mean Execution Time (sec)

5 Conclusions

In this assignment, we had to optimize a sequential CPU implementation of a contrast enhancement algorithm for grayscale image using CUDA. We managed to gain a moderate speedup of x3.08.

Our findings demonstrated that we should always consult the CUDA Occupancy Calculator to choose our geometry since increased parallelization does not always equate to increased speed. Our target application was not computationally intensive, so there is little room for performance

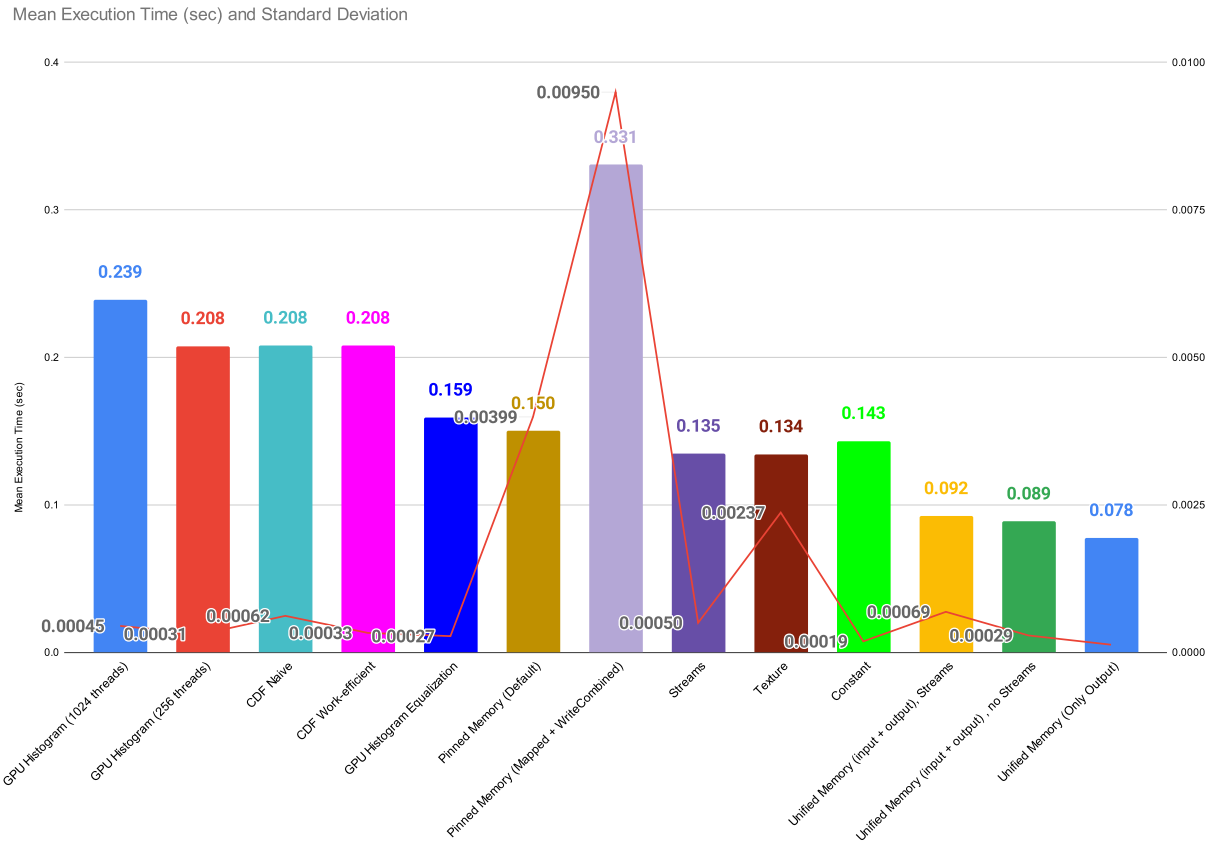

Figure 23: Mean Total Execution Time and Standard deviation for all configurations

gain due to the hardware resources of the GPU. In order to gain sufficient speedup, we had to take great care of where are data are residing at any moment. We also had to exploit task parallelism using CUDA streams to mitigate the effects of data transfers.

Overall, each optimization we did contributed only slightly to the performance. The decrease in execution time was gradual and required a lot of steps, as shown in figure 23. To enhance this argument, we include figure 24 which demonstrates the speedup for each optimization attempt we made.

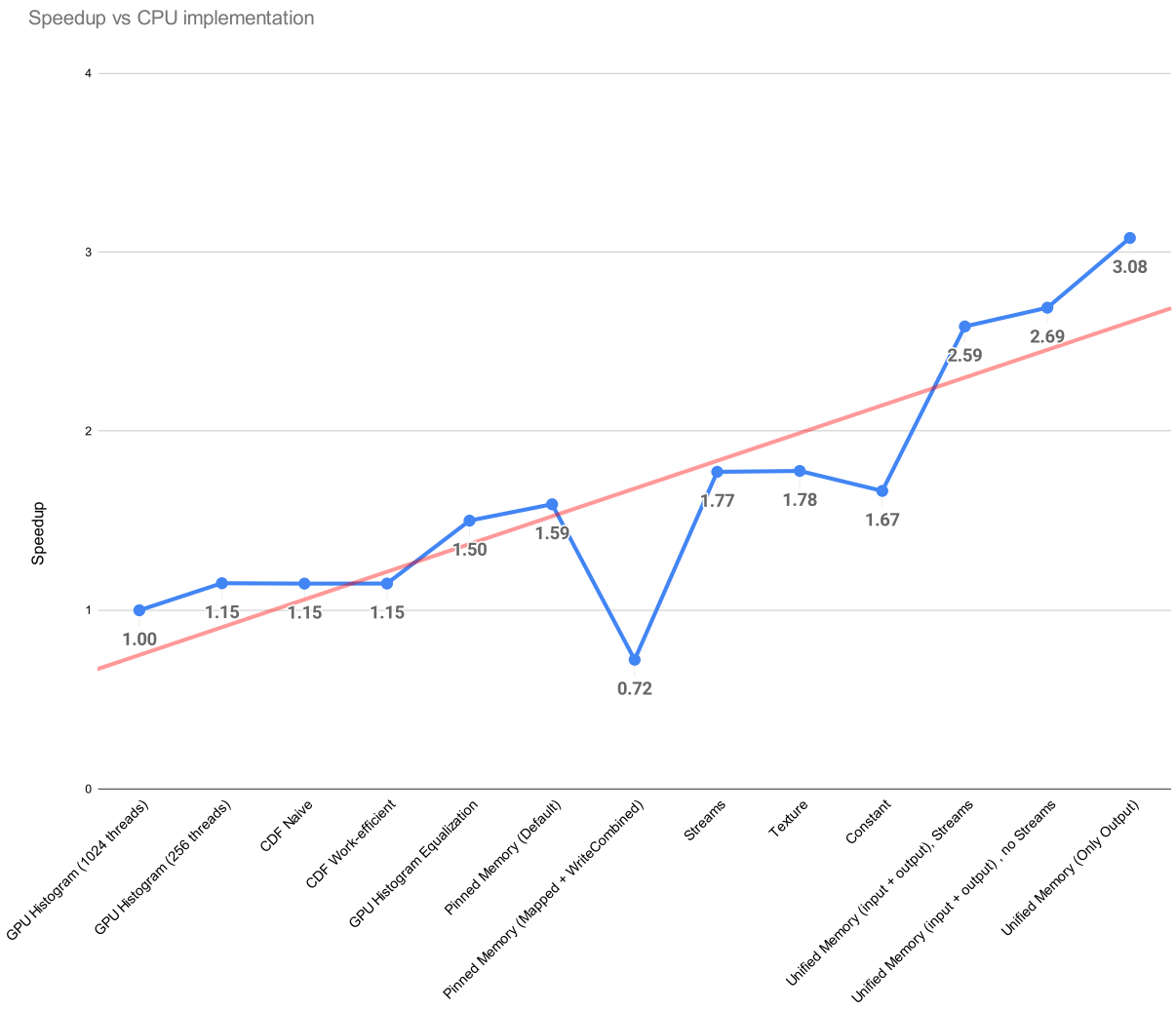

Figure 24: Speedup across our optimization attempts