Electrical and Computer Engineering
University of Thessaly (UTH)

ECE415 - High Performance Computing (HPC)

Fall Semester — Educational year 2024-2025

Lab04

CUDA Implementation of histogram contrast enhancement for grayscale images

Dimitris Tsalapatas - AEM: 03246

Georgios Kapakos - AEM: 03165

Contents

1 Introduction **3**

2 Evaluation **3**

3 Optimizations **3**
3.1 CPU . . . 3
3.1.1 First spot of divergence . . . 4
3.1.2 Second spot of divergence . . . 4
3.2 Default GPU . . . 5
3.3 Histogram GPU . . . 6
3.3.1 Shared Memory . . . 7
3.3.2 Shared Memory with loop . . . 8
3.4 Histogam Equalization . . . 10
3.5 Pinned Memory . . . 11
3.5.1 Pinned Memory with `cudaHostAllocDefault` . . . 12
3.5.2 Pinned Memory with `cudaHostAllocMapped` . . . 13
3.6 Unified Memory . . . 15
3.7 Constant Memory . . . 16
3.8 Texture Memory . . . 17
3.9 Streams . . . 19

4 Different Implementation Results on Images **22**

5 Conclusion **24**

1 Introduction

The histogram equalization method increases the global contrast of an image, especially when an image is represented by a narrow histogram. The histogram spreads out and and the image becomes more clear, as more pixels are represented. This transforms lower contrast areas into higher contrast ones. It is an especially useful technique for images with background and foreground that are both bright or dark.
The algorithm first calculates, how many pixels are represented in each bin of the histogram. The histogram consists of 256 bins, because we represent grayscale images. We construct a LUT(look-up table), which contains the CDF (cumulative distribution function) of the distribution of the histogram. This is calculated based on the intensity values of the pixels in the image. Lastly, we map the output image's pixels to the input image's pixels, based on the lut created before.

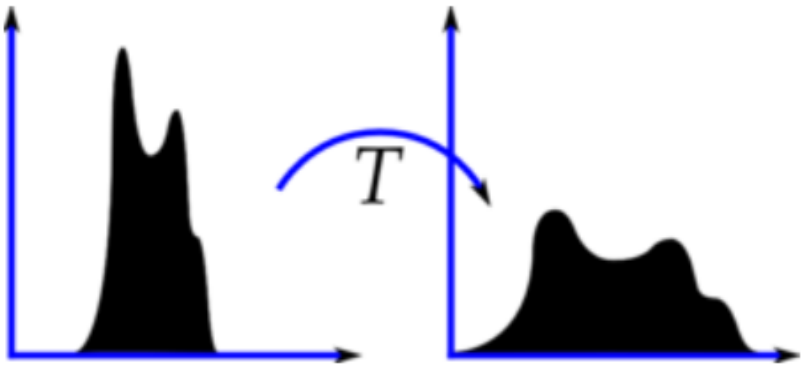

Figure 1: Histogram Equalization

2 Evaluation

We run the optimizations for all the images in the folder. We record each optimization we used even the ones who result in a worse performance. For the evaluation of our code we use diff between the CPU and the GPU generated images. The way we proceed with the evaluation of the images is that we keep the best solution and going to the next optimization. But we kept the optimizations with the shared memory, even though they yielded worse results in the larger image.

3 Optimizations

3.1 CPU

The CPU code traverses in a row-wise fashion the input vector for both the histogram function and the histogram equalization function. It is not an expensive algorithm as one for loop results in O(N) complexity.

We observed a couple spots where divergence is present. Divergence is a significant problem as it can create hick-ups when we will later try to optimize our code, so we altered the code to eliminate whenever divergence is spotted, especially in the histogram_equalization function:

3.1.1 First spot of divergence

In the first for loop where the lut is calculated, the if statement can be avoided. By initializing the the lut vector to zero, until the first non-zero element of the input histogram and then calculating the look-up tables based on the cdf added to the input histogram's value. We don't have to look for values that are negative as cdf will always be larger or equal to the min value we previously calculated.

```
// Finds the first value on the Histogram that isn't 0
    while(min == 0) {
        min = hist_in[i++];
    }
    index = i-1;

    // Calculate the look up table (lut)
    for (i = 0; i < index + 1; i++)  {
        lut[i] = 0;
    }

    d = img_size - min;
    for(i = index; i < nbr_bin; i++) {
        cdf += hist_in[i];
        lut[i] = (int)(((float)cdf - min)*255/d + 0.5);
    }
```

Figure 2: Solving the divergence problem in the first for-loop of the program.

3.1.2 Second spot of divergence

The second spot where divergence is present, is in the second for loop where the output image is calculated depending on the values of the lut, the if-else statement can be avoided. The course we took to avoid the divergence in the second loop, which is calculated for image size times, and therefore plays a significant role in the parallelization of the code later on, is that we threshold the look-up table's values to 255, in a for loop with the size of 256 (number of bins), and then in another loop we assign the output image directly to the value of the look-up table of the input image directly.

```
for(i = 0; i < nbr_bin; i++)  {
   if(lut[i] > 255) {
        lut[i] = 255;
    }
}

/* Get the result image this is the only part of the function to be run in GPU */
for(i = 0; i < img_size; i++) {
    img_out[i] = (unsigned char)lut[img_in[i]];
}
free(lut);
```

Figure 3: Solving the divergence problem in the second for-loop of the program.

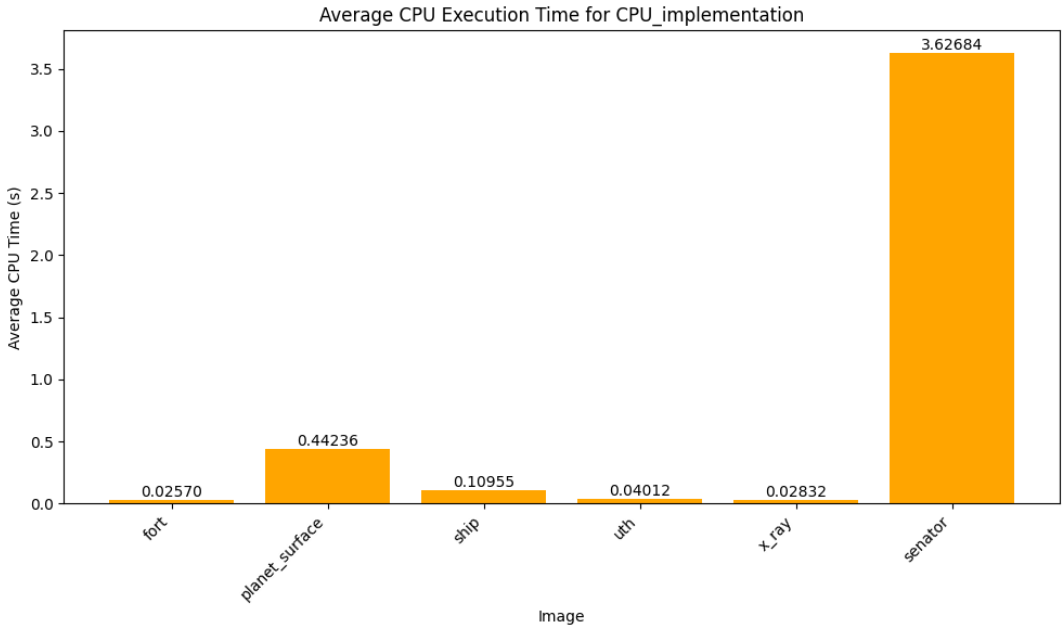

Figure 4: CPU average execution time per image

3.2 Default GPU

At our first attempt using CUDA, we tranfer CPU code to GPU, in some functions, which are worth being optimized in the GPU, in a naive way, with no optimizations. For the GPU code to work we implement atomicAdd and syncthreads to achieve synchronization in the code, atomicAdd is an atomic operation that ensures a read-modify-write cycle is performed without interruption by other threads. It adds a value to a variable in memory and stores the result back in the variable atomically where syncthreads is a synchronization barrier that ensures all threads within a block reach the barrier before any of them can proceed. These functions are the following:

```cpp
__global__ void histogramGPU(int * hist_out, unsigned char * img_in, int imageW, int imageH)
{
    int tx = threadIdx.x;
    int index = blockIdx.x*blockDim.x + tx;

    // Constructs the Histogram Vector
    if (index < imageH*imageW)
    {
        atomicAdd(&hist_out[img_in[index]], 1);
    }
    __syncthreads();

}

__global__ void histogram_equalization_GPU(unsigned char * img_out, unsigned char * img_in, int * lut, int imageW, int imageH)
{
    int index = blockIdx.x*blockDim.x + threadIdx.x;

    /* Get the result image */
    if (index < imageW * imageH)
    {
        img_out[index] = (unsigned char)lut[img_in[index]];
    }
    __syncthreads();
}
```

Figure 5: CUDA implementation, with no optimizations

We decided that the calculation of the histogram and the calculation of the output image from histogram equlization are worth being transfered in the GPU as these are the points of interest as they are executed for the biggest sizes.

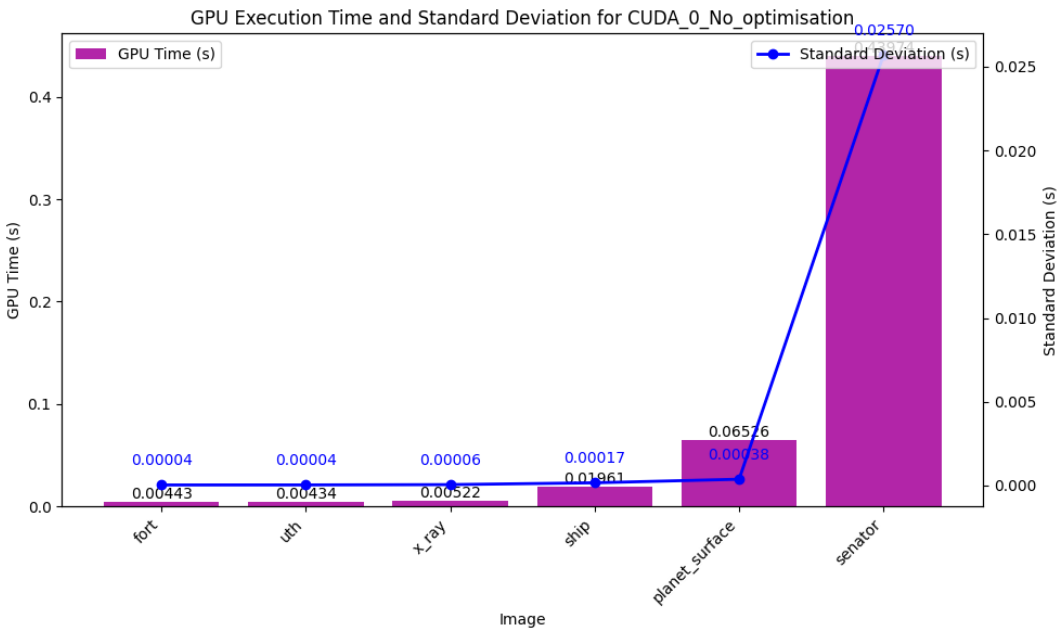

Figure 6: CUDA 0 No optimisation on different images

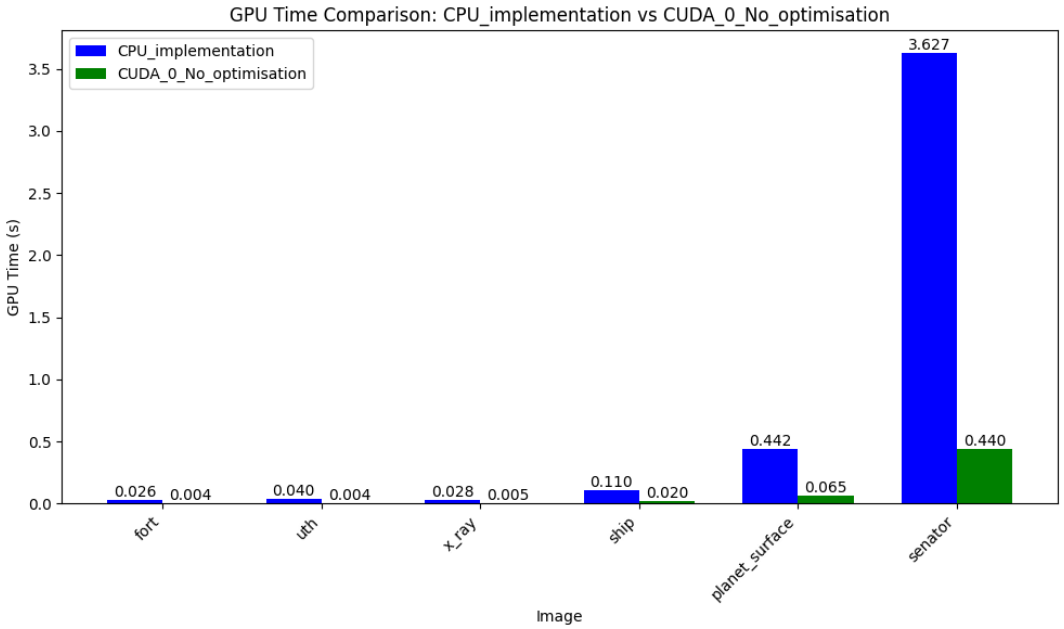

Figure 7: CPU runtime vs CUDA with no optimisations

The GPU implementation yields a great result as the memory pattern in a 1-D matrix is coalesced. A coalesced memory access pattern occurs when threads in a warp access consecutive memory addresses, allowing the GPU to combine multiple memory transactions into a single one, significantly improving memory access efficiency.

3.3 Histogram GPU

In the histogram GPU we take two approaches, an approach using a shared memory block and another using a stride with shared memory.

3.3.1 Shared Memory

Shared memory in CUDA is a **fast memory space** shared by all threads within a block. It resides *on-chip*, making it much faster than global memory but with limited size. Below are some key characteristics:

- **Speed Boost:** Shared memory has significantly lower latency compared to global memory.
- **Collaboration:** Threads within the same block can share data efficiently using shared memory. This is especially useful for algorithms like matrix multiplication and data reduction.
- **Banking:** Shared memory is divided into *banks*, which can be accessed simultaneously by multiple threads. Avoiding *bank conflicts* is critical for performance.
- **Limited Size:** Shared memory size is limited to 49152 bytes per block (48KB).

We noticed that using shared memory to create a private histogram can significantly enhance performance. By assigning 256 threads to calculate the 256 histogram bins in parallel, we can perform these computations much faster within shared memory. Once the calculations are complete, the results are then transferred back to global memory. The choice of 256 threads aligns with the histogram's size (256 bins), ensuring efficient use of shared memory for these calculations.

```cpp
__global__ void histogramGPU(int * hist_out, unsigned char * img_in, int imageW, int imageH)
{
    int tx = threadIdx.x;
    int index = blockIdx.x*blockDim.x + tx;
    extern __shared__ int private_hist[];

    if(tx < 256) {
        private_hist[tx] = 0;
    }

    // Constructs the Histogram Vector
    if (index < imageH*imageW)
    {
        atomicAdd(&private_hist[img_in[index]], 1);
    }
    __syncthreads();

    if(tx < 256) {
        atomicAdd(&hist_out[tx], private_hist[tx]);
    }
}
```

Figure 8: CUDA implementation, with shared memory

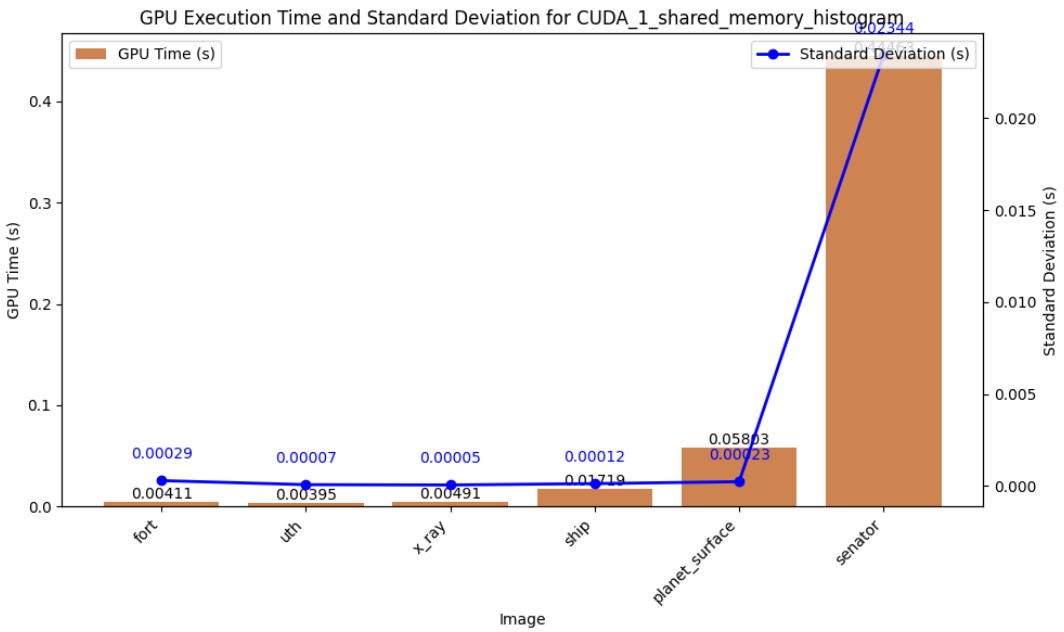

Figure 9: CUDA 1 shared memory on different images

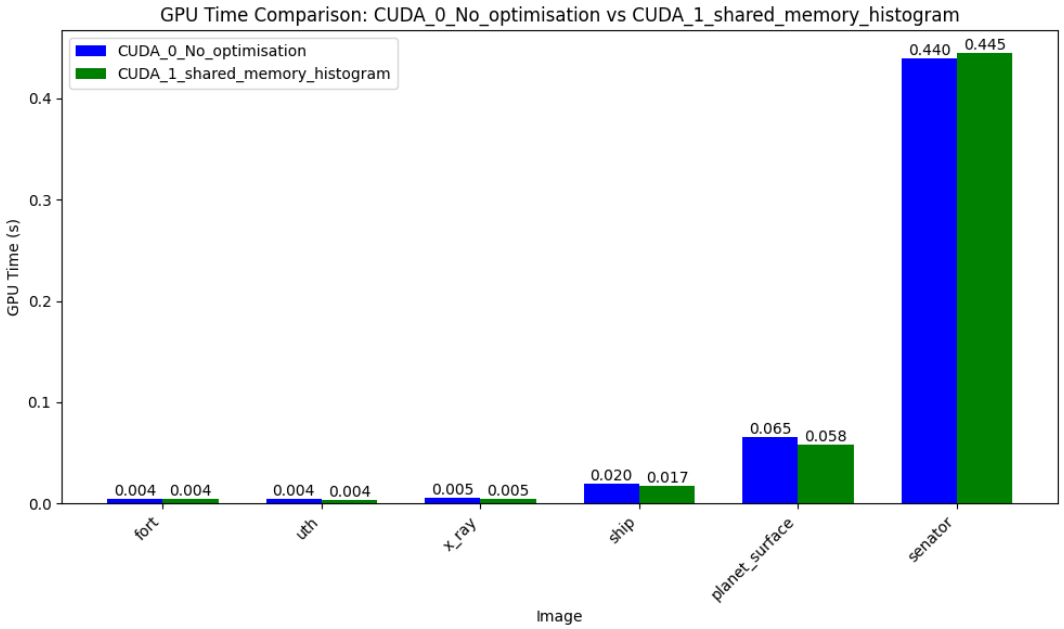

Figure 10: CUDA_0_No_optimisation vs CUDA_1_shared_memory

Shared memory usage can improve performance for smaller images due to fast, low-latency access, but for very large images, but due to increased thread divergence or higher global memory traffic from frequent kernel synchronization or atomic operations, which do not scale well with image size.

3.3.2 Shared Memory with loop

The threads per block remained the same, we again used 256 thread blocks as the private histogram consists of a size of 256. The main reasons we used the shared memory with the stride implementation are:

- **Work Distribution:** Each thread starts at its unique index `index` and processes pixels in strides of `blockDim.x * gridDim.x`.
- **Shared Memory Initialization:** Shared memory, `private_hist`, is initialized by the first 256 threads in each block. This array is used to accumulate partial histogram values.
- **Atomic Operations:** The `atomicAdd` function is used to safely update shared memory and later global memory to avoid race conditions.
- **Thread Reuse:** Threads handle multiple pixels by looping, making efficient use of limited threads for large images.

```cuda
__global__ void histogramGPU(int * hist_out, unsigned char * img_in, int imageW, int imageH)
{
    int tx = threadIdx.x;
    int index = blockIdx.x*blockDim.x + tx;
    extern __shared__ int private_hist[];

    if(tx < 256) {
        private_hist[tx] = 0;
    }

    // Constructs the Histogram Vector
    // Constructs the Histogram Vector
    for(int i = index; i < (imageH * imageW); i += blockDim.x * gridDim.x) {
        atomicAdd(&private_hist[img_in[i]], 1);
    }
    __syncthreads();

    if(tx < 256) {
        atomicAdd(&hist_out[tx], private_hist[tx]);
    }
}
```

Figure 11: CUDA implementation, with shared memory and strides

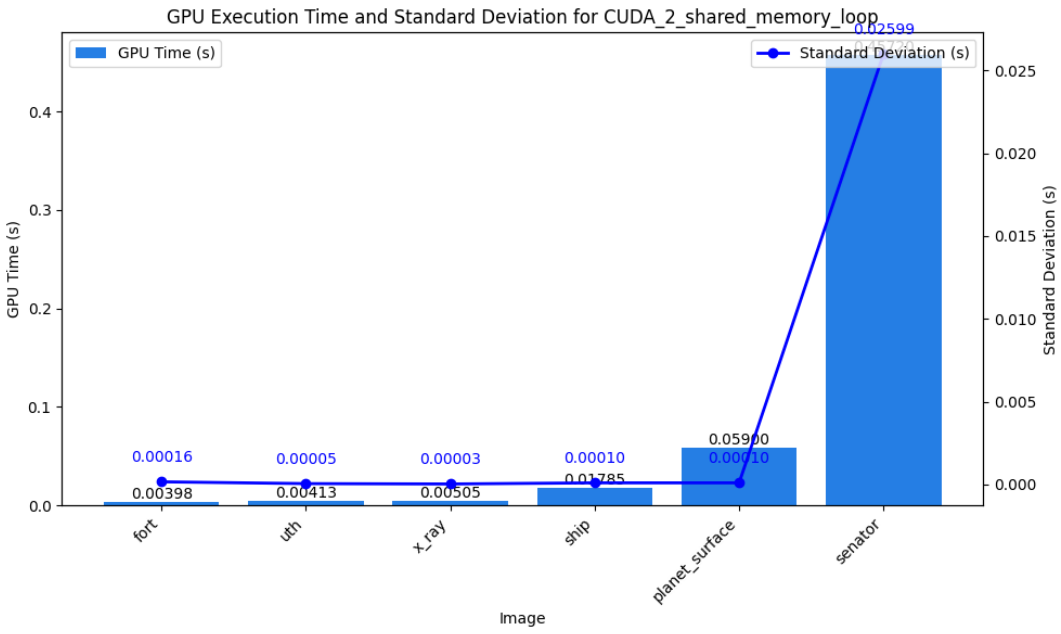

Figure 12: CUDA 2 memory loop on different images

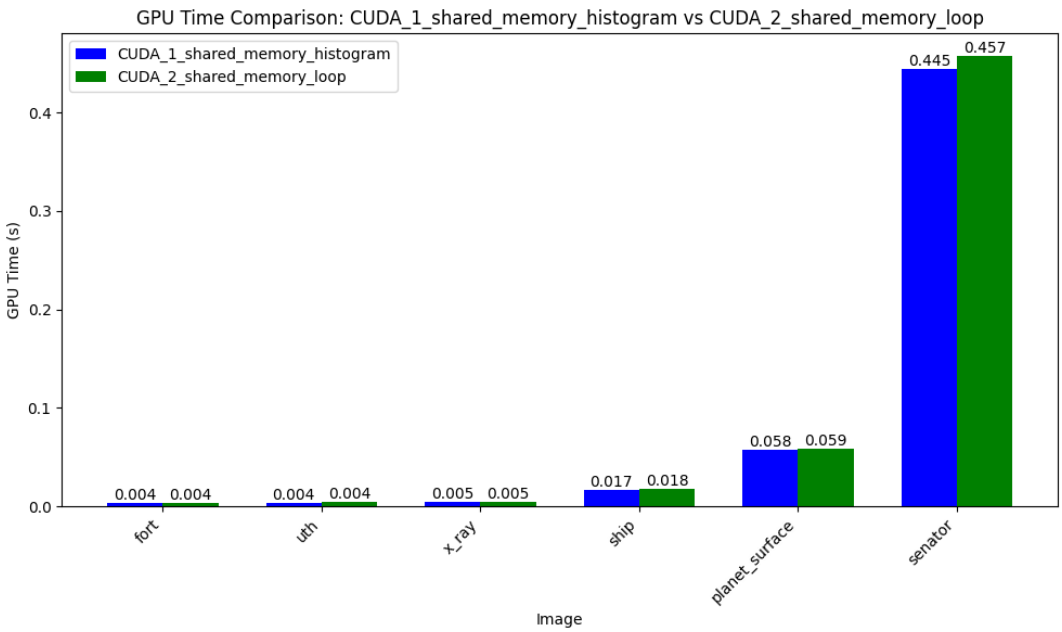

Figure 13: CUDA_1_shared_memory_histogram vs CUDA_2_shared_memory_loop

A loop with shared memory implementation may result in worse performance due to repeated synchronization and increased contention within the shared memory, which can negate the benefits of its fast access.

3.4 Histogam Equalization

In the same fashion as in the previous steps we will try to implement shared memory for the histogram equalization function. This is achieved by transfering a private look-up table into the shared memory space. The private lut will copy the values of the lut in the global memory and then assigning them to the output of the image's values based on this shared memory lut.

```
__global__ void histogram_equalization_GPU(unsigned char * img_out, unsigned char * img_in, int * lut, int imageW, int imageH)
{
    int tx = threadIdx.x;
    int index = blockIdx.x*blockDim.x + tx;
    extern __shared__ int private_lut[];

    if(tx < 256) {
        private_lut[tx] = lut[tx];
    }

    __syncthreads();

    /* Get the result image */
    if (index < imageW * imageH)
    {
        img_out[index] = (unsigned char)private_lut[img_in[index]];
    }
    __syncthreads();
}
```

Figure 14: CUDA implementation, with shared memory in the histogram equalization function

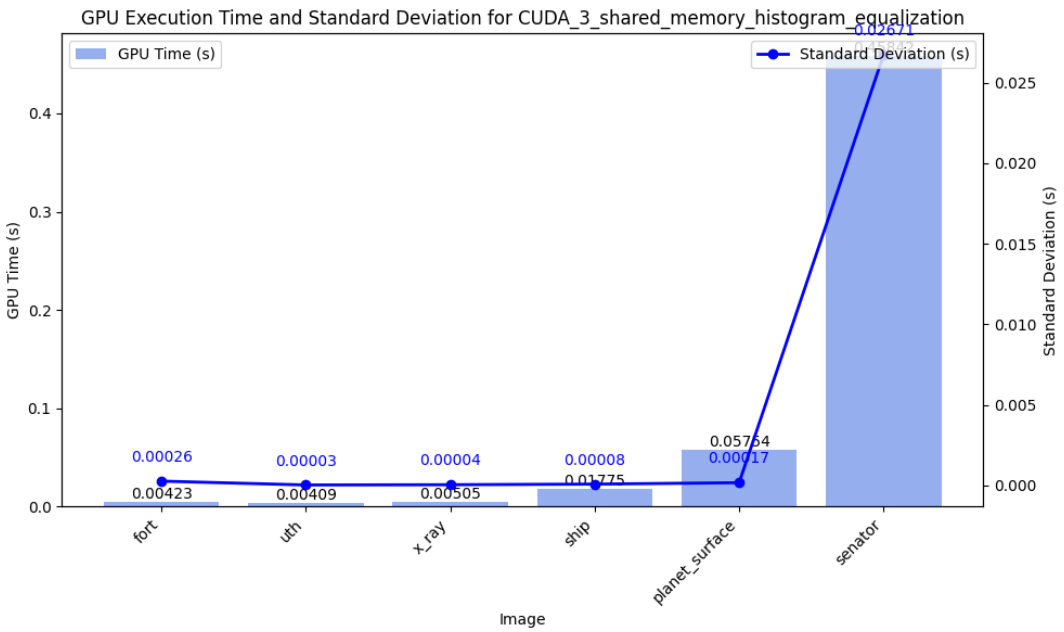

Figure 15: CUDA 3 shared memory histogram equalization on different images

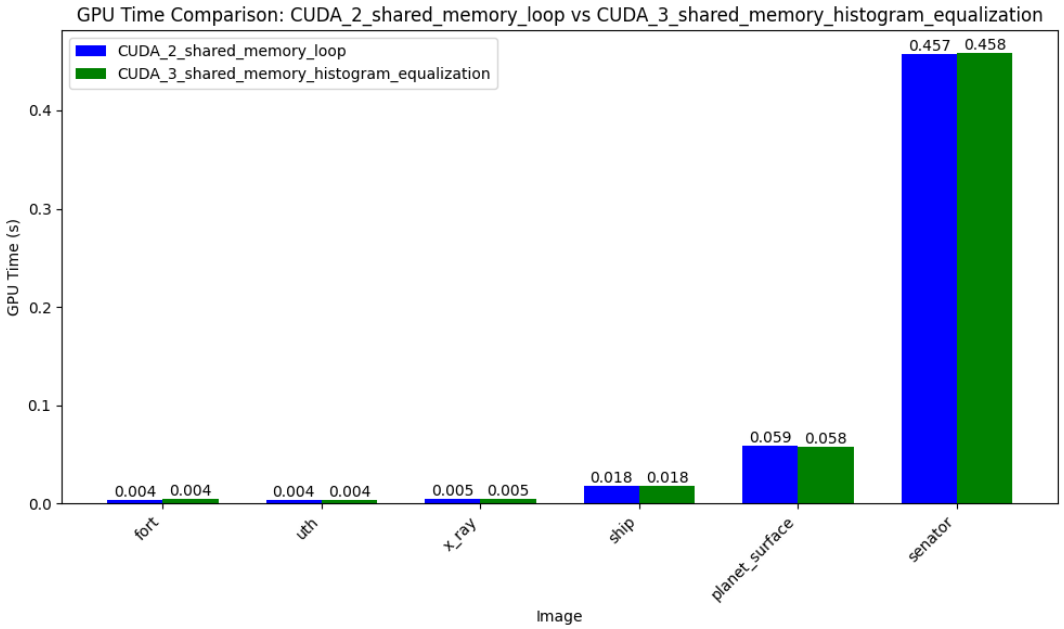

Figure 16: CUDA_2_shared_memory_loop vs CUDA_3_shared_memory_histogram_equalization

In the same fashion as before the shared memory implentation results in better results in the smaller images where in the largest one still there seems to be a discrepancy.

3.5 Pinned Memory

In CUDA, **pinned memory** (or page-locked memory) refers to host memory that is allocated and locked into physical RAM. Unlike pageable memory, pinned memory cannot be swapped out by the operating system. This enables faster and more efficient data transfers between the host and the GPU.

The `cudaHostAlloc()` function is used to allocate pinned memory. Depending on the specified flags, the behavior and accessibility of the pinned memory can vary. Two common flags are `cudaHostAllocDefault` and `cudaHostAllocMapped`.

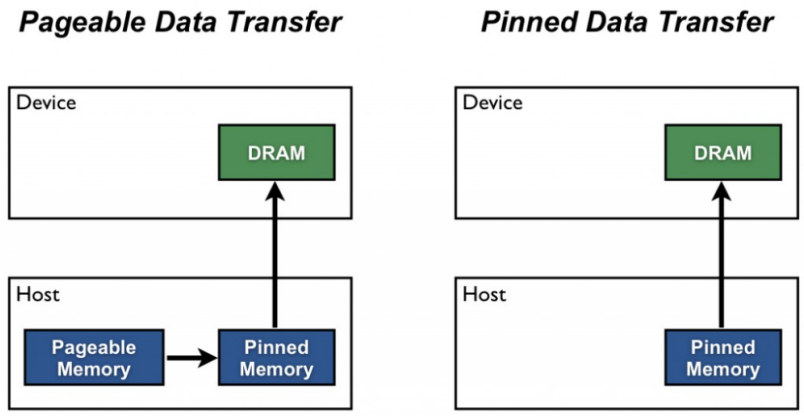

Figure 17: Pinned Memory

3.5.1 Pinned Memory with `cudaHostAllocDefault`

Pinned memory with the `cudaHostAllocDefault` flag:

- The memory resides in host (CPU) memory and is pinned (page-locked).
- This type of pinned memory is primarily used for fast host-to-device and device-to-host data transfers.
- By using pinned memory, the GPU can directly access the memory during transfers without staging through intermediate buffers.
- The transfer bandwidth is significantly higher compared to pageable memory.

In the code we implemented the pinned memory to allocate memory for the input image and also for the allocation of the look-up tables.

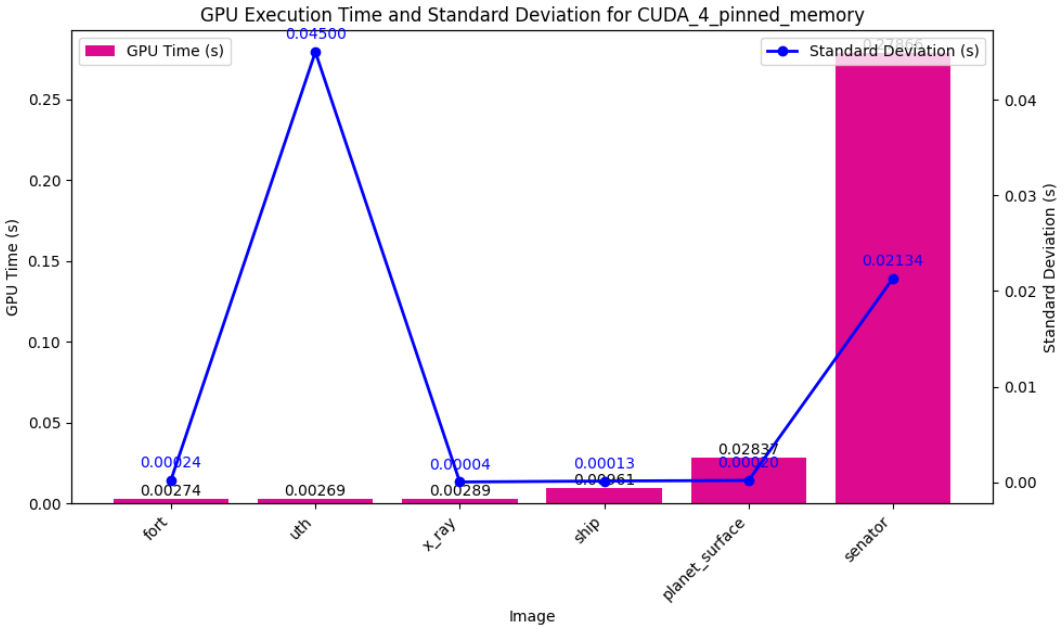

Figure 18: CUDA 4 pinned memory on different images

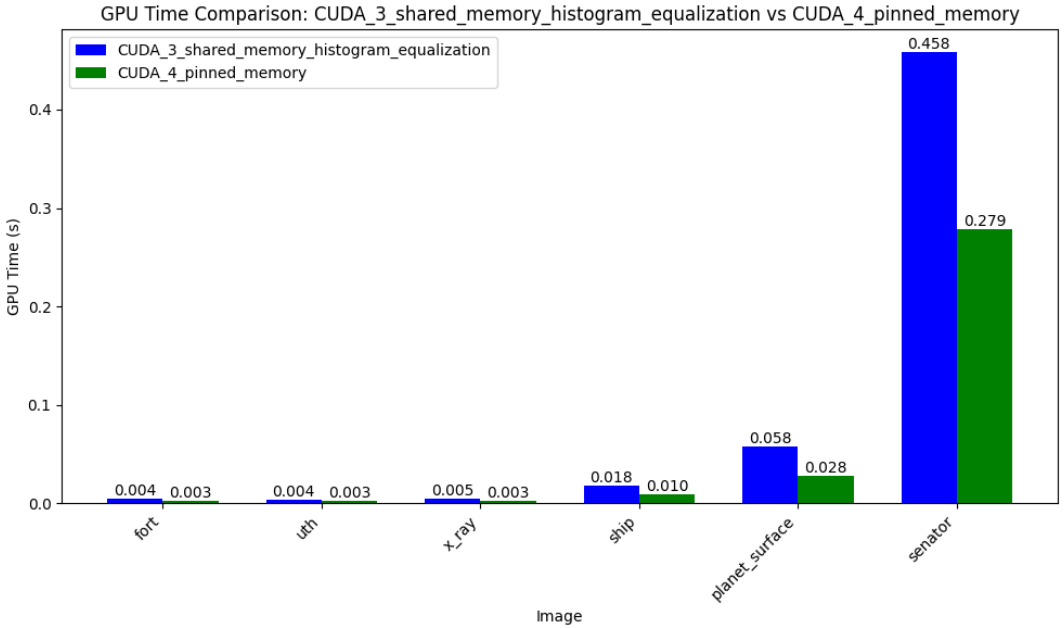

Figure 19: CUDA_3_shared_memory_histogram_equalization vs CUDA_4_pinned_memory

Pinned memory with the default flag optimizes performance by enabling faster and more efficient data transfers between host and device through direct access to the host memory without the need for intermediate buffering, as seen by the results in all the images.

3.5.2 Pinned Memory with `cudaHostAllocMapped`

Pinned memory with the `cudaHostAllocMapped` flag:

- The allocated memory is pinned and also mapped into the GPU's address space.

- Threads running on the GPU can directly access this memory (DMA) using a device pointer, eliminating the need for explicit data transfers.
- This feature is useful for applications where host memory needs to be shared dynamically with the GPU, such as zero-copy memory.
- However, the performance of direct access is typically lower than accessing global memory on the GPU, due to the latency of PCIe or other host-GPU interconnects.

The code doesn't change much as only the flag is altered. Also the device pointer to the memory may be obtained by calling `cudaHostGetDevicePointer()`.

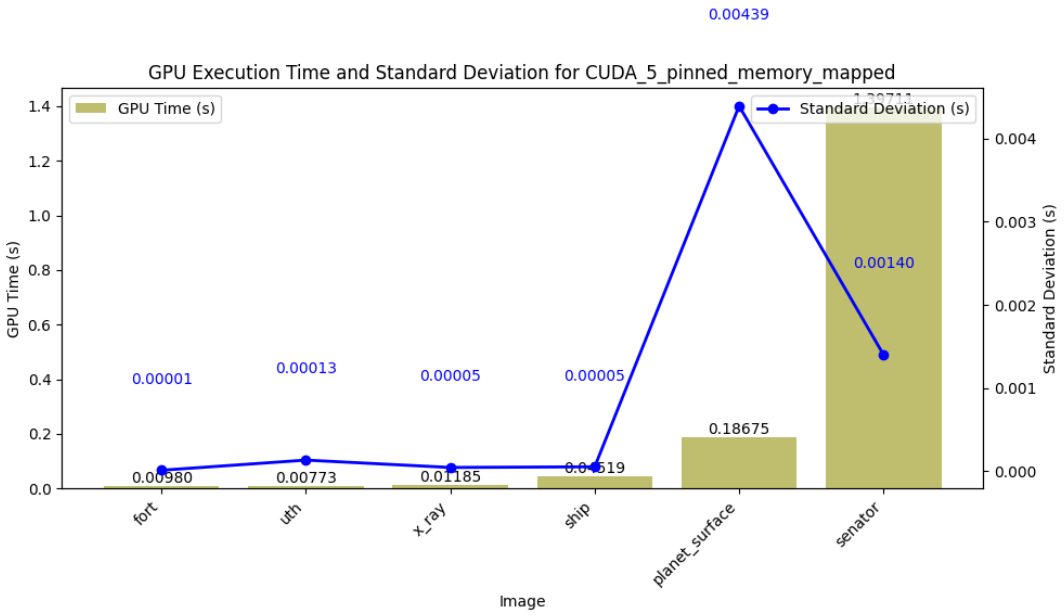

Figure 20: CUDA 5 pinned memory mapped on different images

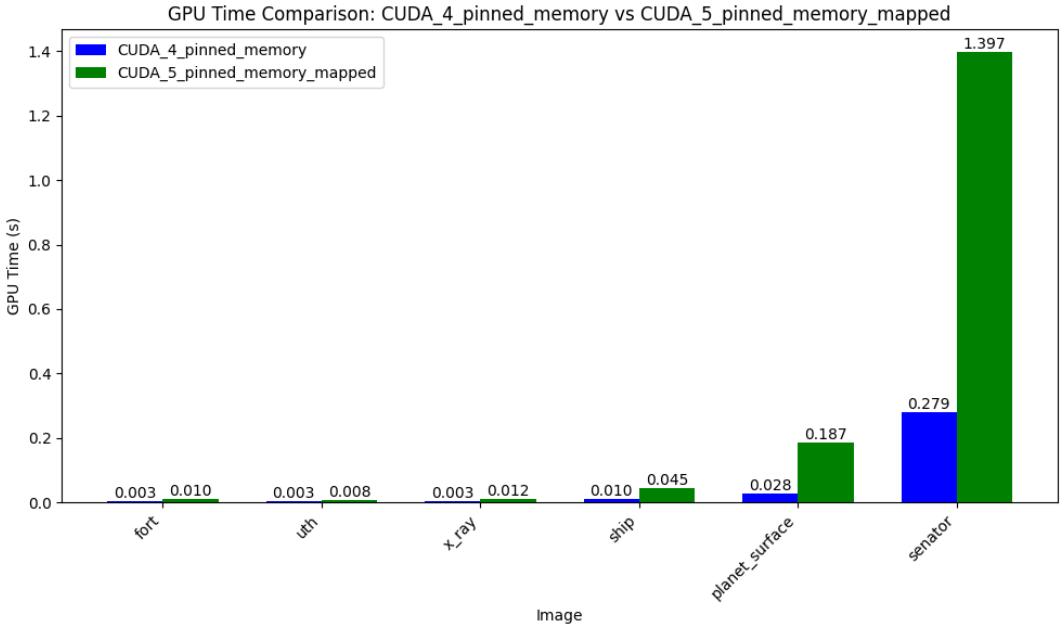

Figure 21: CUDA_4_pinned_memory vs CUDA_5_pinned_memory_mapped

Mapped memory can be worse than the default pinned memory because direct GPU access to mapped host memory incurs higher latency and lower bandwidth due to the PCIe interconnect, making it slower than transferring data to GPU global memory first.

3.6 Unified Memory

Unified Memory is a memory management model that allows both the CPU and GPU to share a single memory space. This makes it easier to go from the device to the host and vice-versa by removing the need for explicit memory copies between host and device.

Unified Memory is a single memory address space accessible from any processor in a system. This hardware/software technology allows applications to allocate data that can be read or written from code running on either CPUs or GPUs. Allocating Unified Memory is as simple as replacing calls to `malloc()` or new with calls to `cudaMallocManaged()`, an allocation function that returns a pointer accessible from any processor (ptr in the following).

When code running on a CPU or GPU accesses data allocated this way (often called CUDA managed data), the CUDA system software and/or the hardware takes care of migrating memory pages to the memory of the accessing processor.

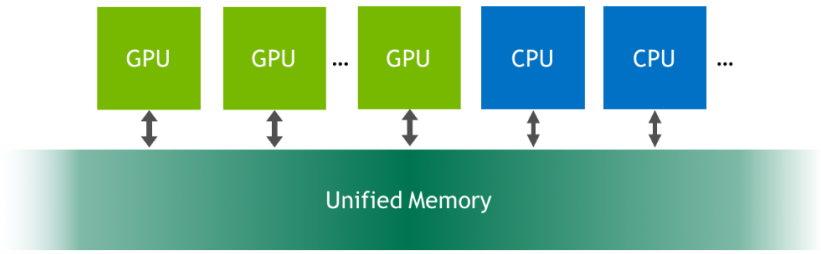

Figure 22: Unified Memory

In the code for unified memory we used the `cudaMallocManaged()`, for the allocation of the input image and the look-up table vector.

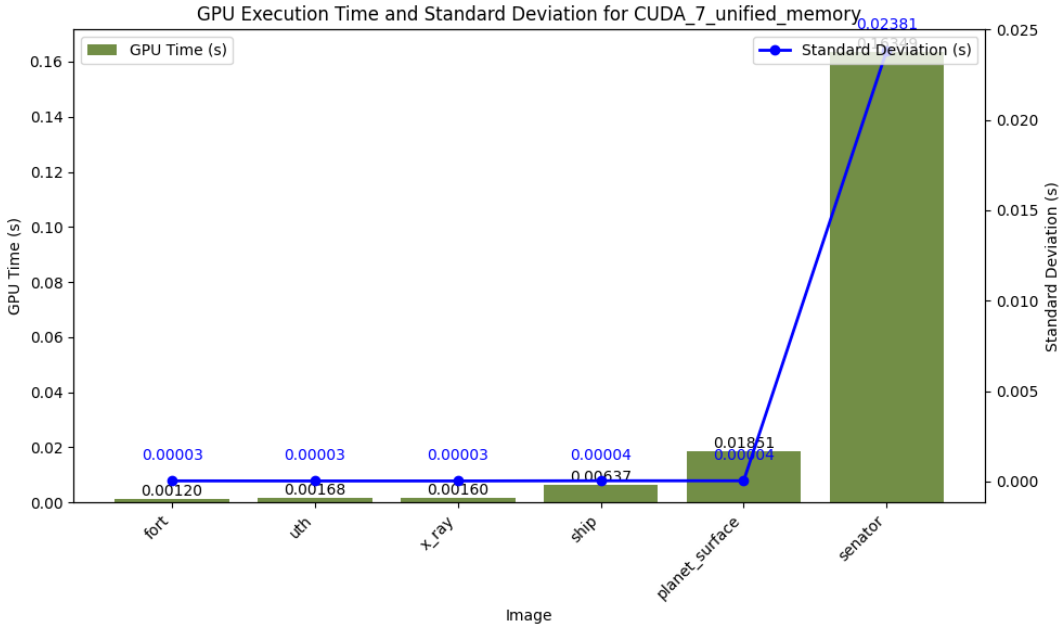

Figure 23: CUDA 7 Unified memory on different images

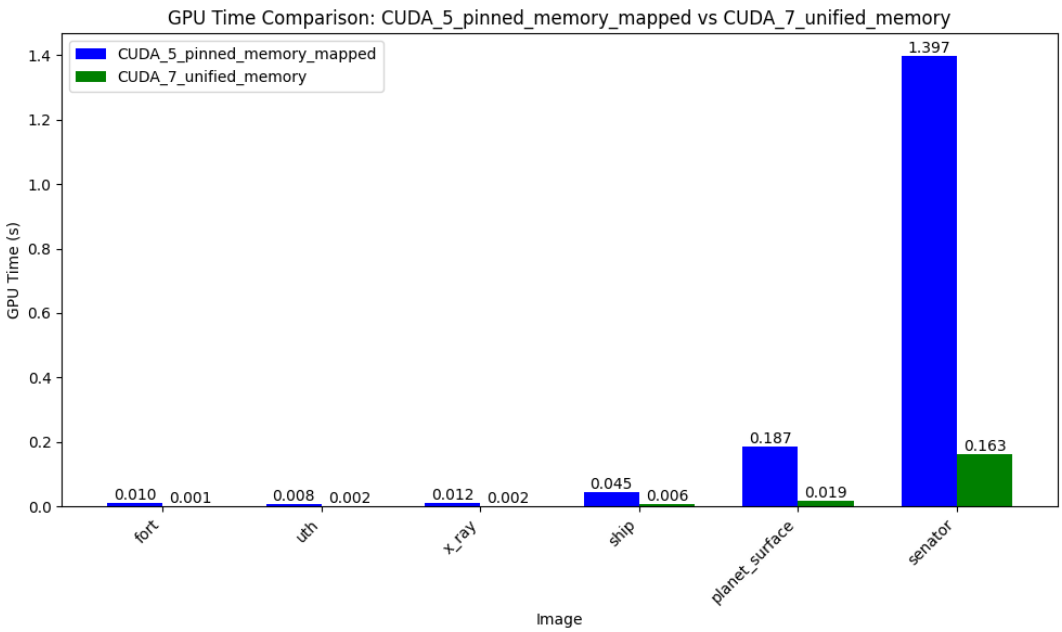

Figure 24: CUDA_5_pinned_memory_mapped vs CUDA_7_unified_memory

Unified memory is better than pinned memory in terms of ease of use because it automatically manages data movement between host and device, simplifying programming and eliminating the need for explicit memory transfers.

3.7 Constant Memory

- **Read-Only Access:** Constant memory is only accessible for read operations. Any attempt to write to it from the host or device will result in an error.
- **Cached Memory:** Constant memory is cached, which means that it can be accessed more quickly than global memory. The data is cached on-chip, making access very fast for threads in a warp.
- **Access Pattern:** To maximize performance, the access pattern should be coalesced. All threads in a warp should access the same memory location to fully utilize the cache.
- **Size Limitations:** The size of constant memory is limited, 64 KB.

```
__constant__ int constant_lut[256] ;
__global__ void histogram_equalization_GPU(unsigned char * img_out, unsigned char * img_in, int * lut, int imageW, int imageH) {
    int index = blockIdx.x*blockDim.x + threadIdx.x;

    /* Get the result image */
    if (index < imageW * imageH)  {
        img_out[index] = constant_lut[img_in[index]];
    }
}
```

Figure 25: Cuda Optimization with the use of constant memory

We implemented constant memory in the histogram equalization function. We declare the lut table in the constant memory. The memory access pattern is convenient as the matrix is 1-D, so the memory is already coalesced, meaning that the constant memory will perform very well.

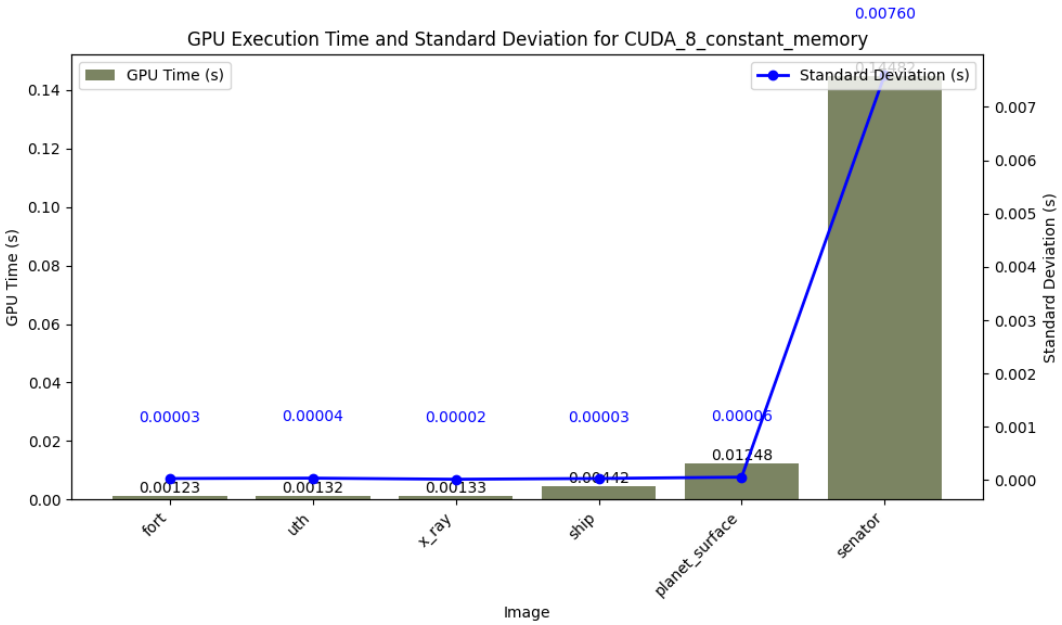

Figure 26: CUDA 8 constant memory on different images

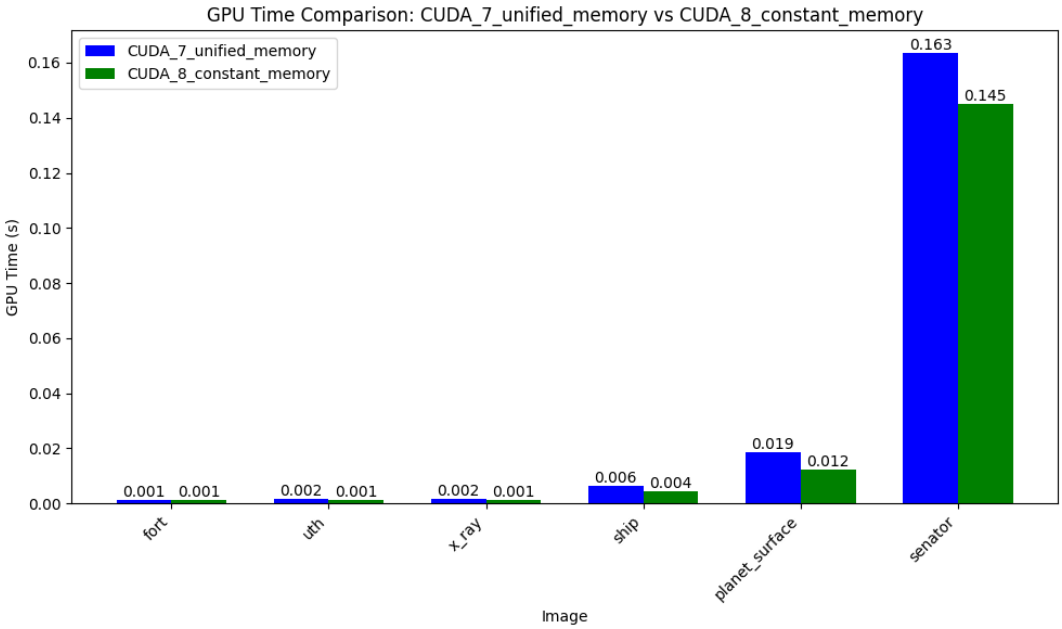

Figure 27: CUDA_7_unified_memory vs CUDA_8_constant_memory

Constant memory is better than shared memory for read-only data that is accessed by all threads because it is cached and optimized for such use, providing faster access and reducing contention, whereas shared memory is limited in size and can only be accessed by threads within the same block.

3.8 Texture Memory

Like constant memory, texture memory is cached on chip, so in some situations it will provide higher effective bandwidth by reducing memory requests to off-chip DRAM. Specifically,

texture caches are designed for graphics applications where memory access patterns exhibit a great deal of spatial locality. In a computing application, this roughly implies that a thread is likely to read from an address “near” the address that nearby threads read.

The read-only texture memory space is cached. Therefore, a texture fetch costs one device memory read only on a cache miss. Otherwise, it just costs one read from the texture cache.

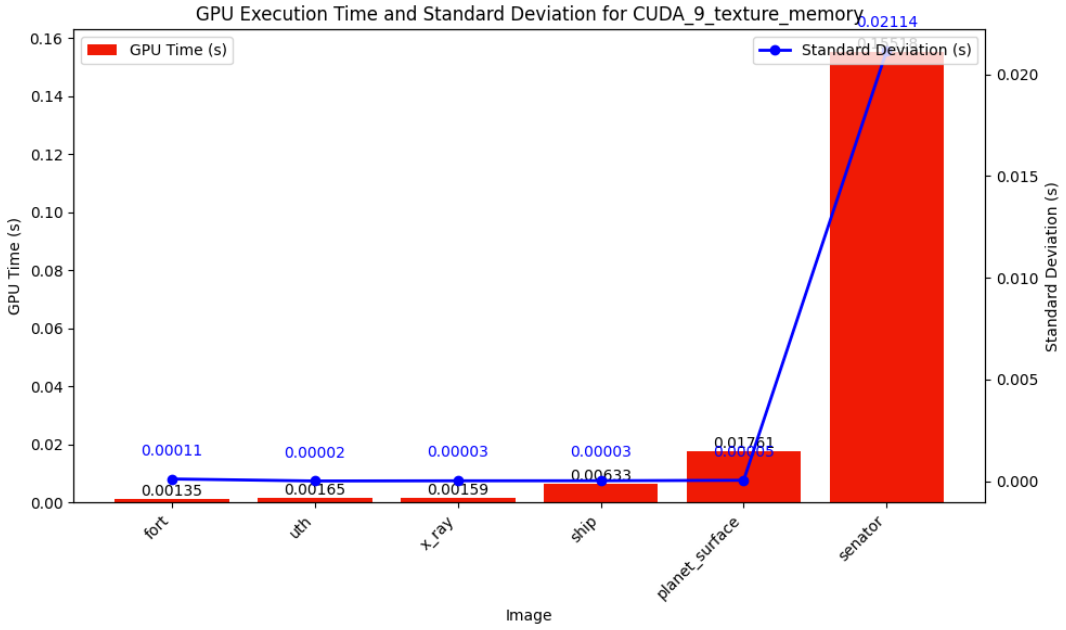

Figure 28: CUDA 9 texture memory on different images

```
texture<int, cudaTextureType1D, cudaReadModeElementType> lut_texture; // Bind the 1D texture

__global__ void histogram_equalization_GPU(unsigned char * img_out, unsigned char * img_in, int * lut, int imageW, int imageH) {
    int index = blockIdx.x*blockDim.x + threadIdx.x;

    /* Get the result image */
    if (index < imageW * imageH) {
        img_out[index] = tex1Dfetch(lut_texture, img_in[index]);
    }
}
```

Figure 29: Cuda Optimization with the use of texture memory

We used the texture memory in the histogram equalization function, when declared the lut table in the texture memory, and we fetch it from there to update the values of the output image.

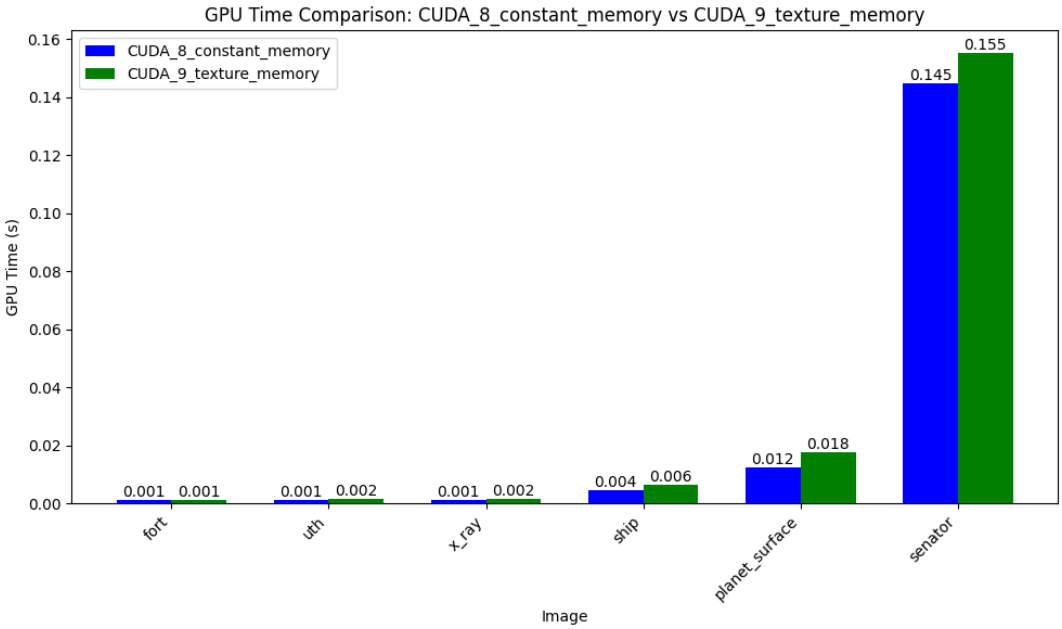

Figure 30: CUDA_8_constant_memory vs CUDA_9_texture_memory

Texture memory can be a bit worse than constant memory because, while it offers spatial locality optimizations (like caching for 2D/3D data), it has higher latency for simple, one-dimensional read-only data, and its access pattern benefits are less efficient compared to the highly optimized caching of constant memory for uniform data access.

3.9 Streams

- The real power of streams is that they can execute concurrently on the GPU.
- Making use of multiple streams can improve the efficiency as the GPU can schedule multiple kernels at once.
- The overlap functionality is critical: The GPU can be running a kernel while copying items between the host and the GPU.
- **Asynchronous Operations:** CUDA operations launched in different streams can execute asynchronously with respect to each other.
- **Independent Execution:** Each stream operates independently, enabling parallel execution of multiple kernels or memory operations.

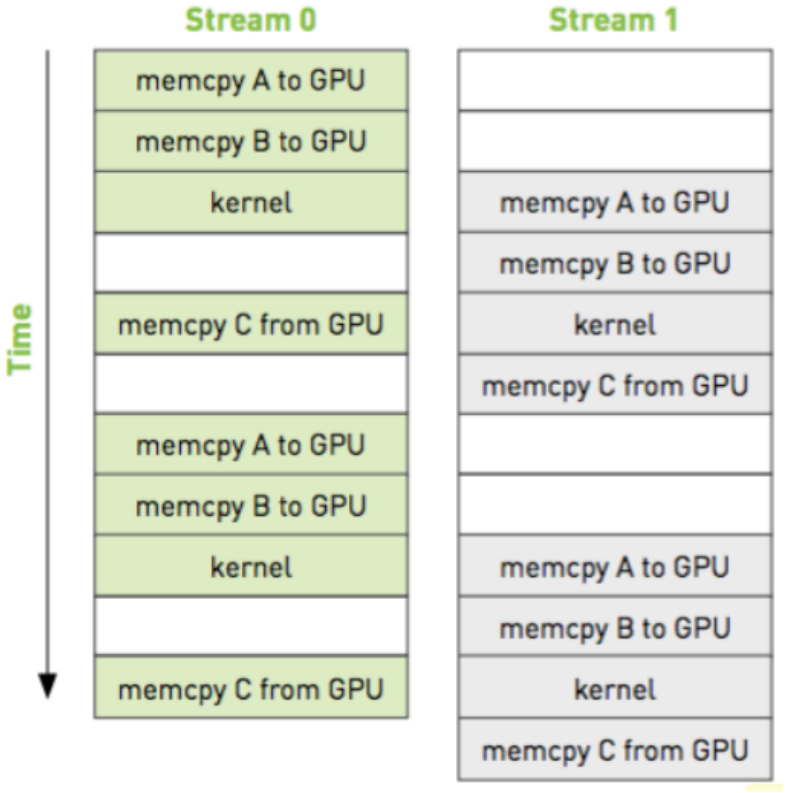

Figure 31: Cuda Optimization with streams

```
for (int i=0; i < 2; i++)  {
        int offset = i * seg_size;
        cudaStreamCreate(&stream[i]);

        histogramGPU<<<((seg_size)/256), 256, 256*sizeof(int), stream[i] >>>(d_hist, img_in.img+offset, seg_size);

        cudaMemcpyAsync(hist, d_hist, 256*sizeof(int), cudaMemcpyDeviceToHost, stream[i]);
    }
```

Figure 32: Cuda Optimization with streams

In the code we add a second stream that is identical to the first. The second steam will need its own memory for the calculations. The first stream calculates the first half of the image where the second image the other half of the operations. This splits the calculations in half. Each streaming multiprocessor (SM) will do its part.

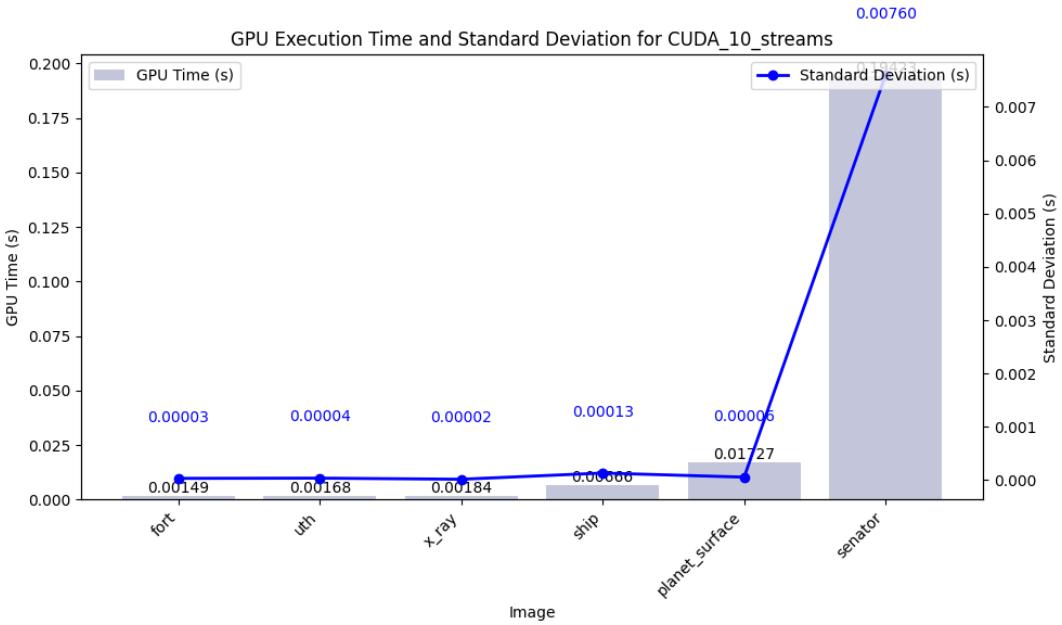

Figure 33: CUDA 10 streams on different images

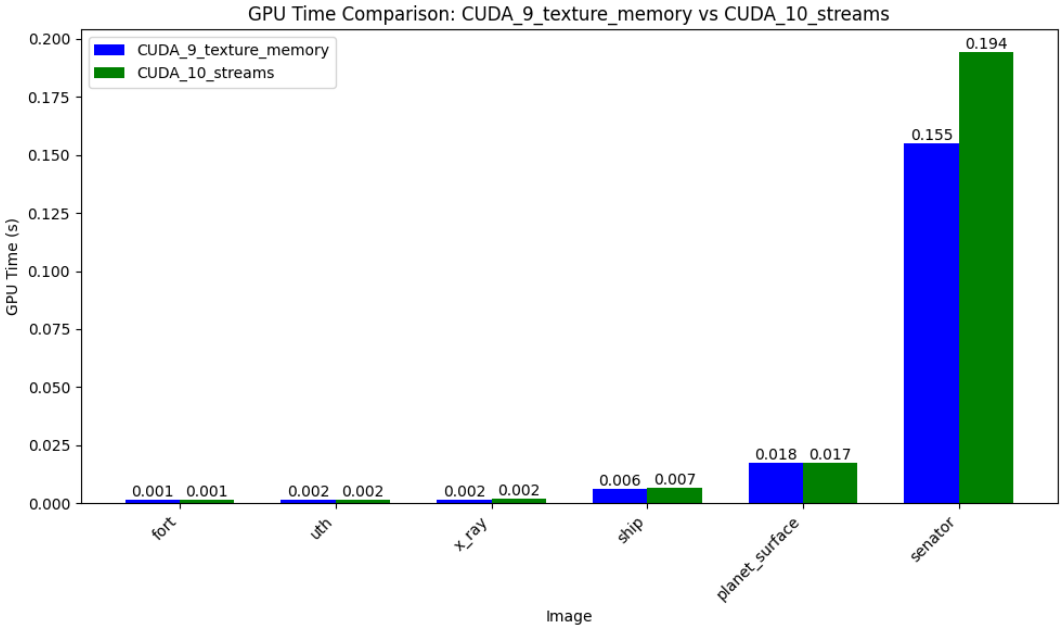

Figure 34: CUDA_9_texture_memory vs CUDA_10_streams

Two streams may perform worse than one due to excessive contention for GPU resources, such as memory or compute units, leading to inefficiencies in execution. The two streams may not overlap effectively.

4 Different Implementation Results on Images

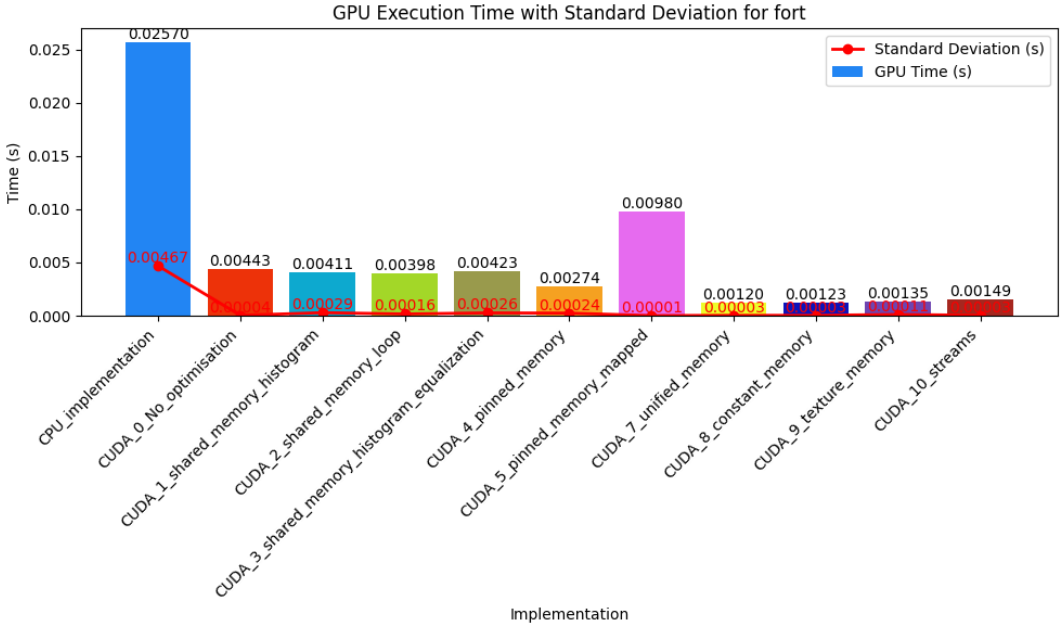

Figure 35: Different Implementations Runtime on fort image

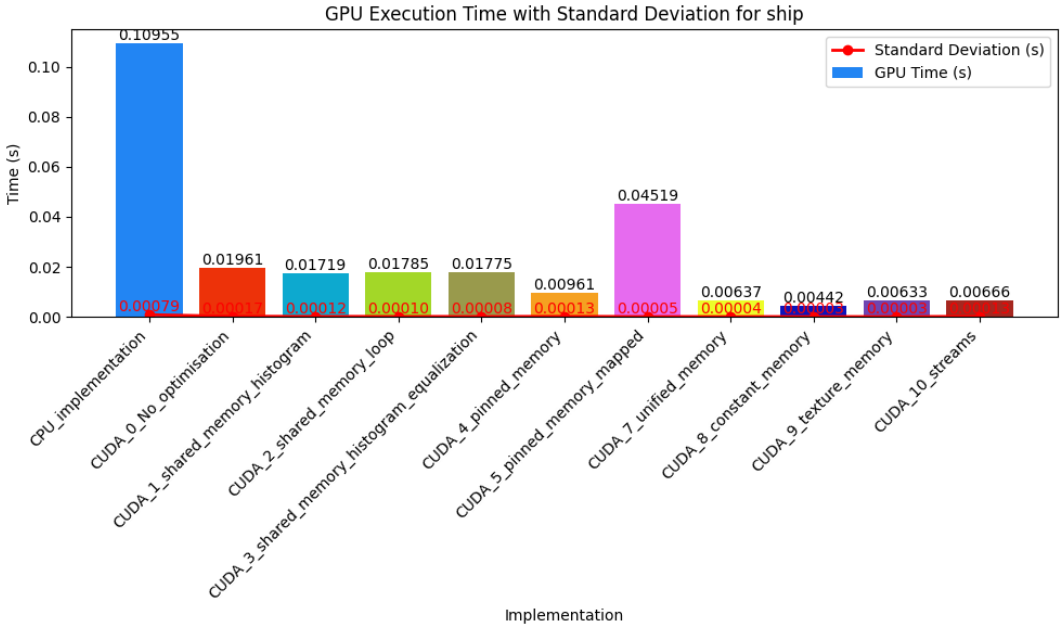

Figure 36: Different Implementations Runtime on ship image

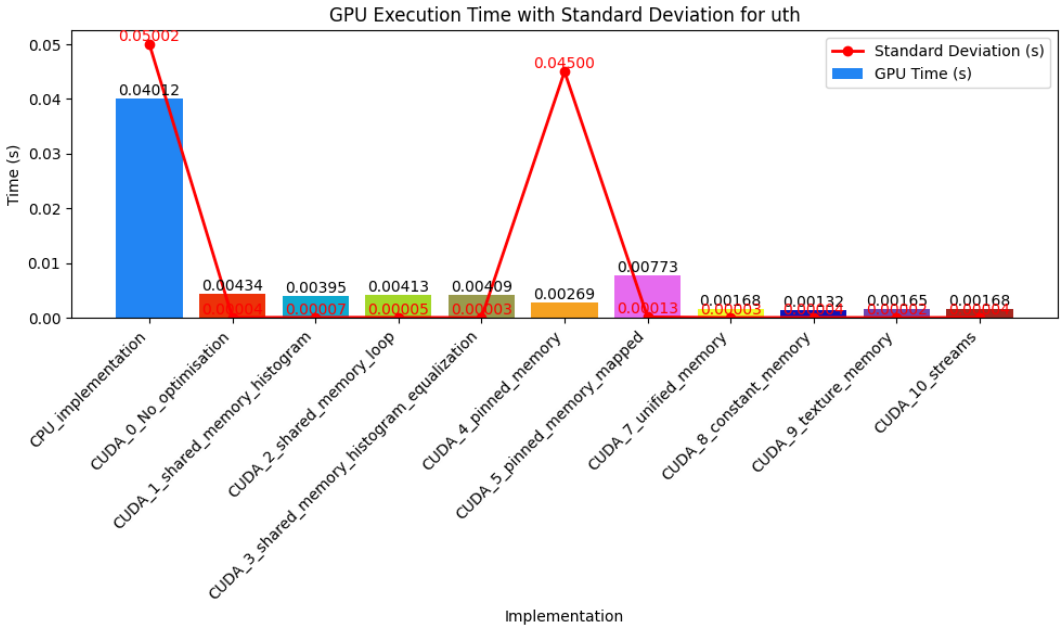

Figure 37: Different Implementations Runtime on UTH image

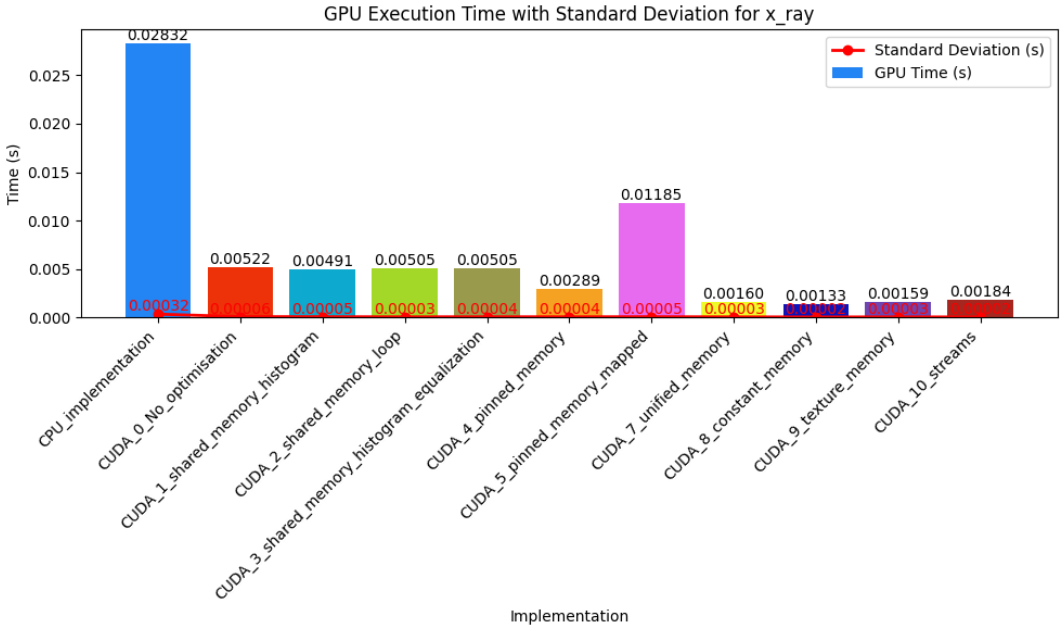

Figure 38: Different Implementations Runtime on x_ray image

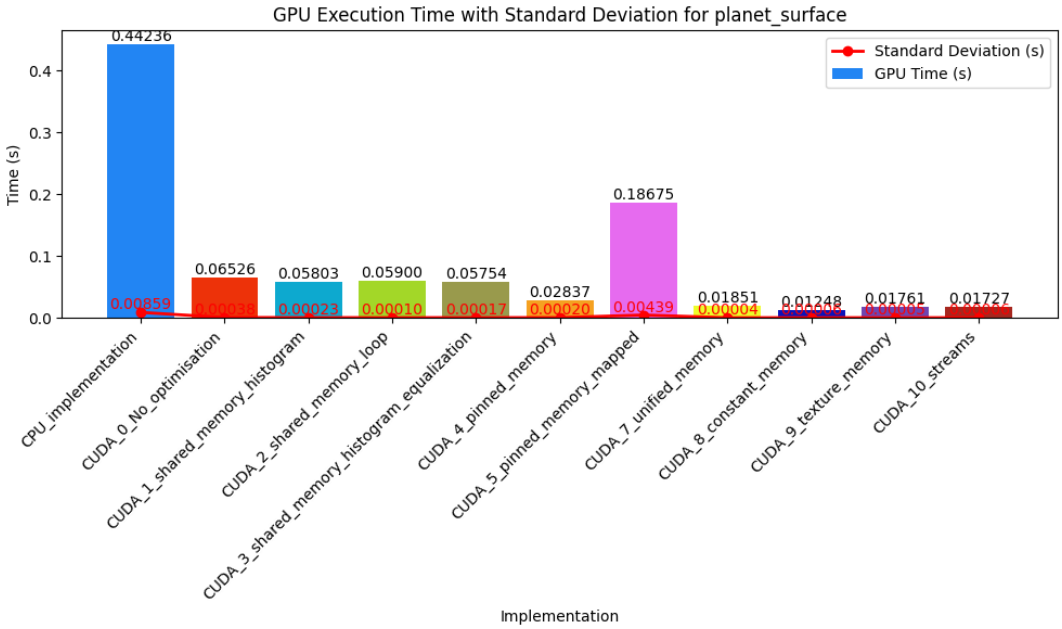

Figure 39: Different Implementations Runtime on planet_surface image

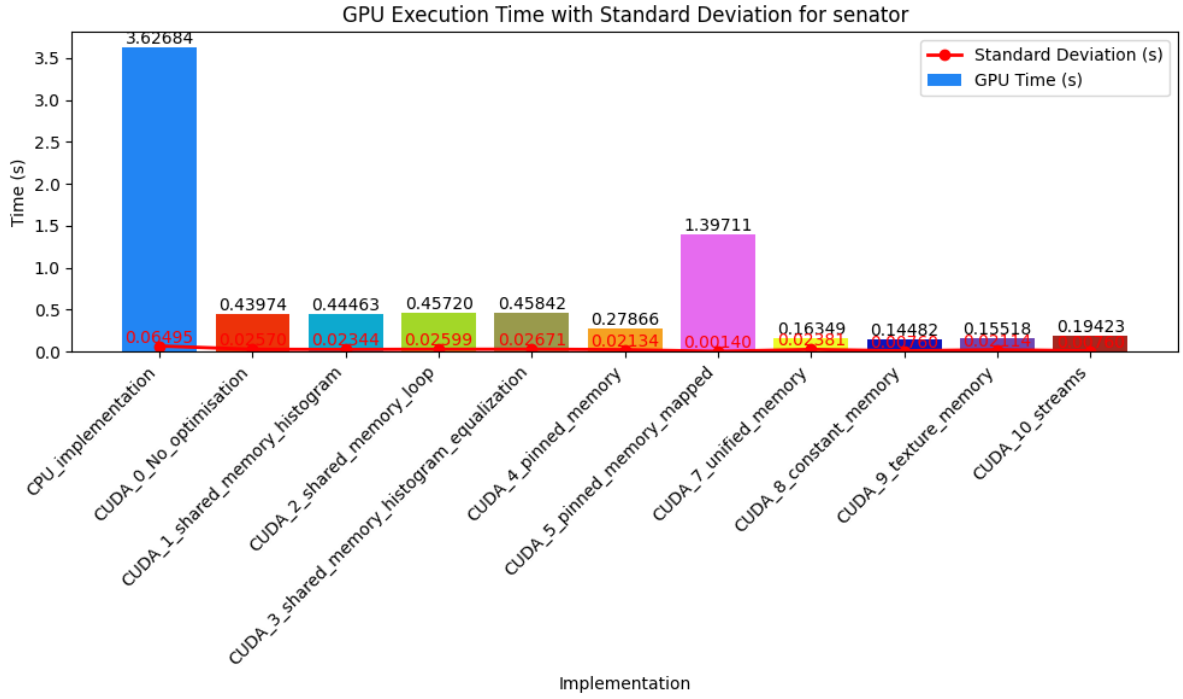

Figure 40: Different Implementations Runtime on Senator image

5 Conclusion

The best testcase to compare the different implementations of our code is **Senator** because it is much bigger than the others. We focus on this image and the runtimes in different implementations as shown in Figure 40. So, the best implementation of our code is version **CUDA_8_constant_memory** which has a runtime of **0.14482 sec** for this image.

This version includes a unified memory implementation for the input image, a shared memory implementation for the histogram function and a constant memory implementation inside the histogram equlization function. Here we can see the picture that includes all runtimes on different images for this final implementation.

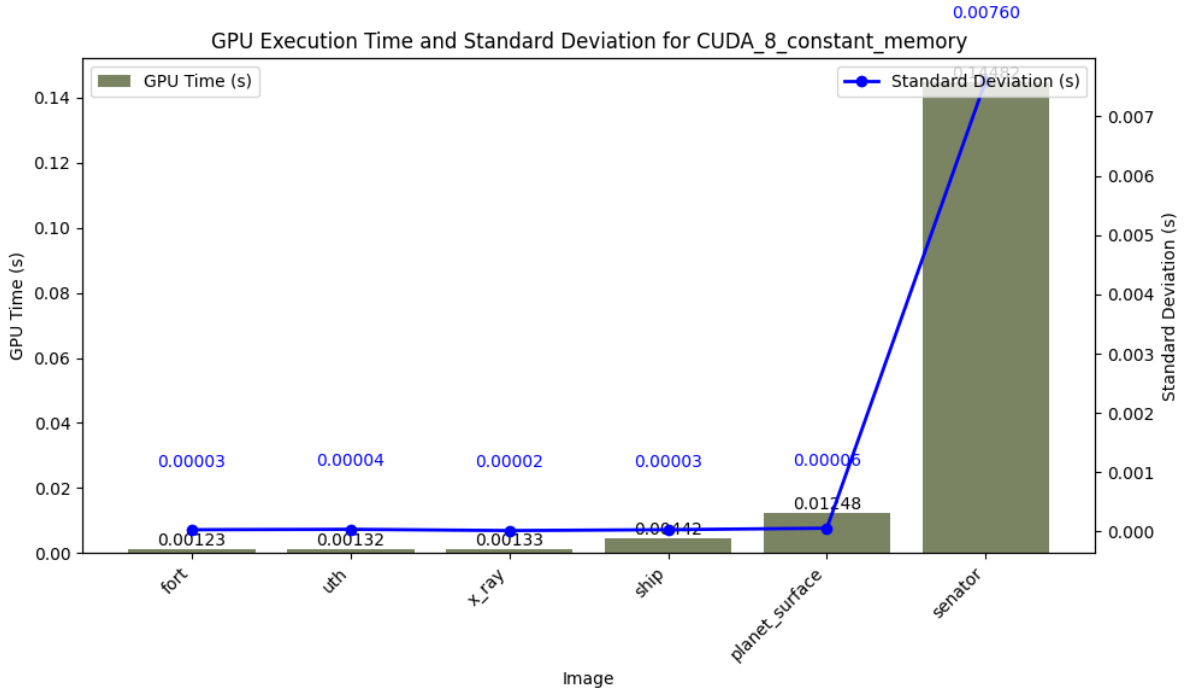

Figure 41: Different Runtimes on Images with final CUDA implementation

Finally, we present an image that analyse the speedups of each implementation according toy CPU runtime on testcase **Senator** image.

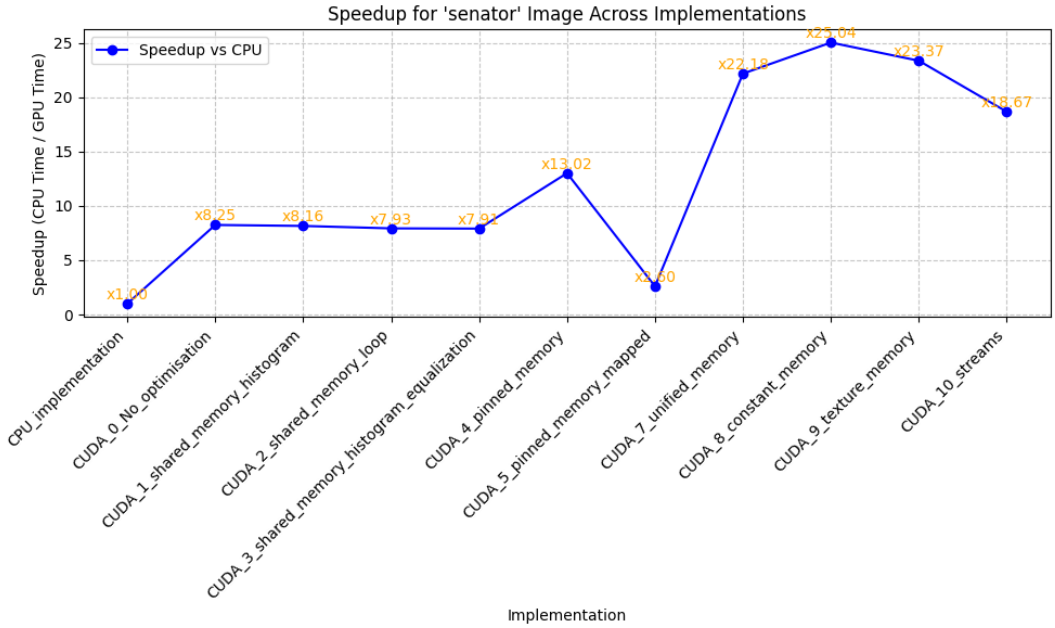

Figure 42: Runtime comparison between different implementations and CPU on Senator Image