ECE415 - High Performance Computing

– Homework 4 –

Histogram Equalization Acceleration with CUDA

submitted by

Aggeliki-Ostralis Vliora, Spyridon Liaskonis

January 19, 2025

Aggeliki-Ostralis Vliora, Spyridon Liaskonis
avliora@uth.gr, sliaskonis@uth.gr
Student ID: 03140,03381

Contents

1 Introduction **1**

2 CPU Execution Times **1**

3 Histogram Calculation Kernel **1**

3.1 Naive Implementation 2

3.2 Privatization 3

3.3 Aggregation 6

3.3.1 Contiguous Partitioning 6

3.3.2 Interleaved partitioning 7

4 Histogram Equalization Kernel **10**

4.1 Naive Implementation 11

4.2 Privatization 12

5 CDF Kernel **13**

5.1 Privatization 13

5.2 Inplace computation of LUT in histogram 15

6 Memory Optimizations **17**

6.1 Pinned Memory 17

6.2 Zero-copy Memory 18

6.3 Unified Memory 19

6.4 Texture Memory 20

7 Streams **21**

8 Conclusion **22**

A Device Query Outputs **23**

A.0.1 Venus 23

1 Introduction

In this report, we focus on optimizing a program that computes the histogram of a grayscale image and performs histogram equalization utilizing GPUs and CUDA programming.

We explore a variety of optimization techniques, including shared memory utilization, aggregation, privatization, and memory access optimization. Additionally, we investigate the use of streams to overlap data transfers with kernel execution, aiming to maximize GPU efficiency.

The primary objective of this project is to reduce the execution time of histogram calculation and image equalization as much as possible. To achieve this, we also examine the impact of different data partitioning strategies and memory management techniques, ensuring the final implementation delivers peak performance.

2 CPU Execution Times

Figure 1 show the initial execution times of the application run on the CPU for different images and image sizes.

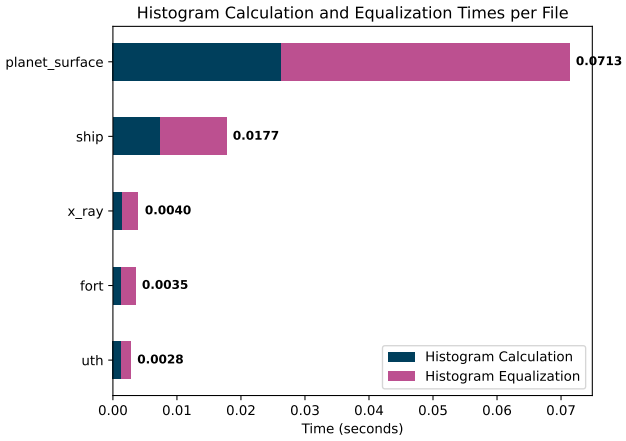

Figure 1: Caption

3 Histogram Calculation Kernel

We will begin by creating a kernel responsible for computing the histogram and explore various implementations to achieve the best possible performance.

3.1 Naive Implementation

In this initial attempt, we implement a naive version of the histogram calculation. Within the kernel, each thread is assigned a unique ID. This ensures that each thread processes a specific pixel from the input image. Using an atomic add operation,each thread inside the kernel increments the corresponding bin in the histogram based on the value of the pixel at `img_in[tid]`, where tid is the thread's unique id. The `atomicAdd()` function ensures thread safety by handling simultaneous updates to the same histogram bin without causing race conditions.

```
1 Histogram calculation: naive implementation
2 __global__ void histogram_calc(int *hist_out, unsigned char *img_in, int img_size, int nbr_bin)
3     int tid = threadIdx.x + blockIdx.x*blockDim.x;
4     atomicAdd(&(hist_out[img_in[tid]]), 1);
5 }
```

Listing 1: Histogram Calculation: Naive Implementation

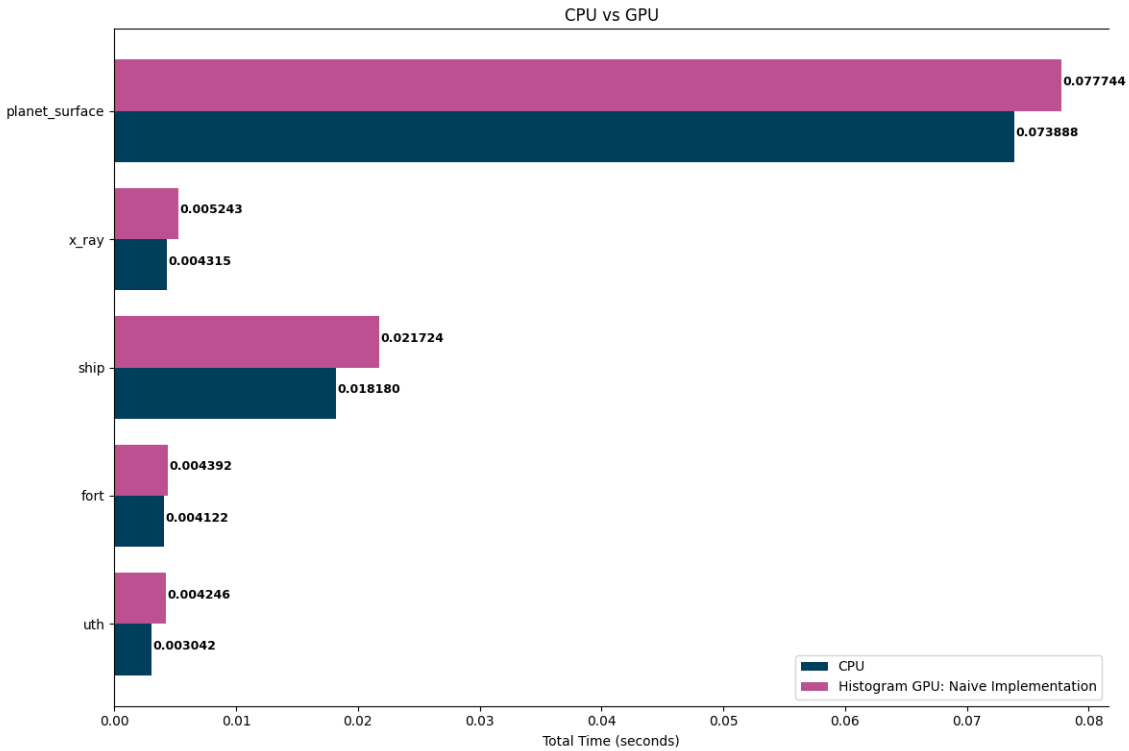

Figure 2: CPU vs GPU Performance using only a kernel for histogram calculation

The main issue with this kernel is that it suffers from congestion due to the use of the atomic add, which creates a bottleneck. Since multiple threads often attempt

to update the same position in the histogram concurrently, they must serialize their accesses to ensure correctness. This serialization significantly reduces the kernel's performance, especially for images with a limited number of unique intensity values (bins), where contention for specific bins becomes more frequent. Also, even though we accelerated the histogram calculation by deploying it on the GPU, there is still a big overhead added by the extra memory transactions from the host to the device as well as from the execution of the histogram equalization which is still being executed in the CPU (as can be seen from Figure 3).

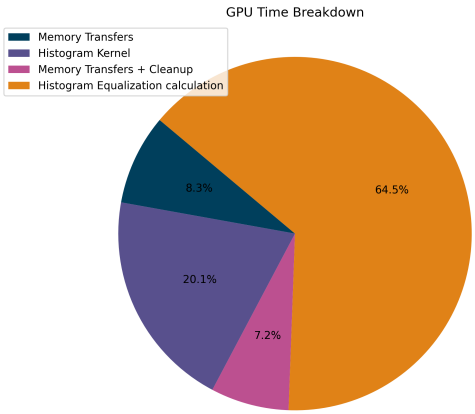

Figure 3: GPU Distribution Time

3.2 Privatization

In our second attempt, we optimize the histogram calculation by introducing a shared memory buffer to reduce global memory contention and improve performance. The kernel operates as follows:

- **Shared Memory Allocation**: A shared memory array `private_hist` is declared, which is local to each block. This array temporarily holds the histogram bins for all threads within the same block. Since shared memory is much faster and local to each block, this significantly reduces the bottleneck caused by frequent atomic operations on global memory.
- **Initialization**: ‘ Number_of_bins threads have to initialize the shared memory array to 0 before the real computation begins. In this implementation, we choose to have 256 threads per block (same as the number of bins for histogram

of a grayscale picture), in order to avoid any case of warp divergence between threads that will have to initialize the histogram array and threads that won't.

- **Histogram Calculation**: For the histogram calculation, we apply interleaved partitioning of threads in order to benefit from coalesced memory accesses from the global memory when accessing the input image. To do this, we divide the number of pixels by a factor of STRIDE, and create this many blocks of 256 threads each (as mentioned before).
- **Reduction to Global Memory**: At the end of the histogram computation, each thread will adds the contents of its corresponding bin in private_hist to the global histogram `hist_out` using an atomic add operation.

Listing 2 show the optimized CUDA implementation for the histogram calculation.

```
__global__ void histogram_calc(int *hist_out, unsigned char *img_in, int img_size, int nbr_bin
    __shared__ int private_hist[256];

   int i = threadIdx.x + blockIdx.x*blockDim.x,
   stride = blockDim.x * gridDim.x;

   private_hist[threadIdx.x] = 0;
   __syncthreads();

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

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

Listing 2: Histogram Calculation: Privatization + Interleaved thread partitioning

Figure 4 show the performance comparison between CPU and GPU times with the new optimized histogram kernel. Once again, we observe that the performance of the histogram calculation has been reduced significantly when compared both to the previous optimization as well as to the CPU histogram calculation time. However, the overall execution time is still bounded by the memory transaction between host and device as we can see from figure 9 as well as the histogram equalization kernel that is still being executed on the CPU. Both the transfers as well as the equalization calculation are now accountable for almost 88% of the total execution time.

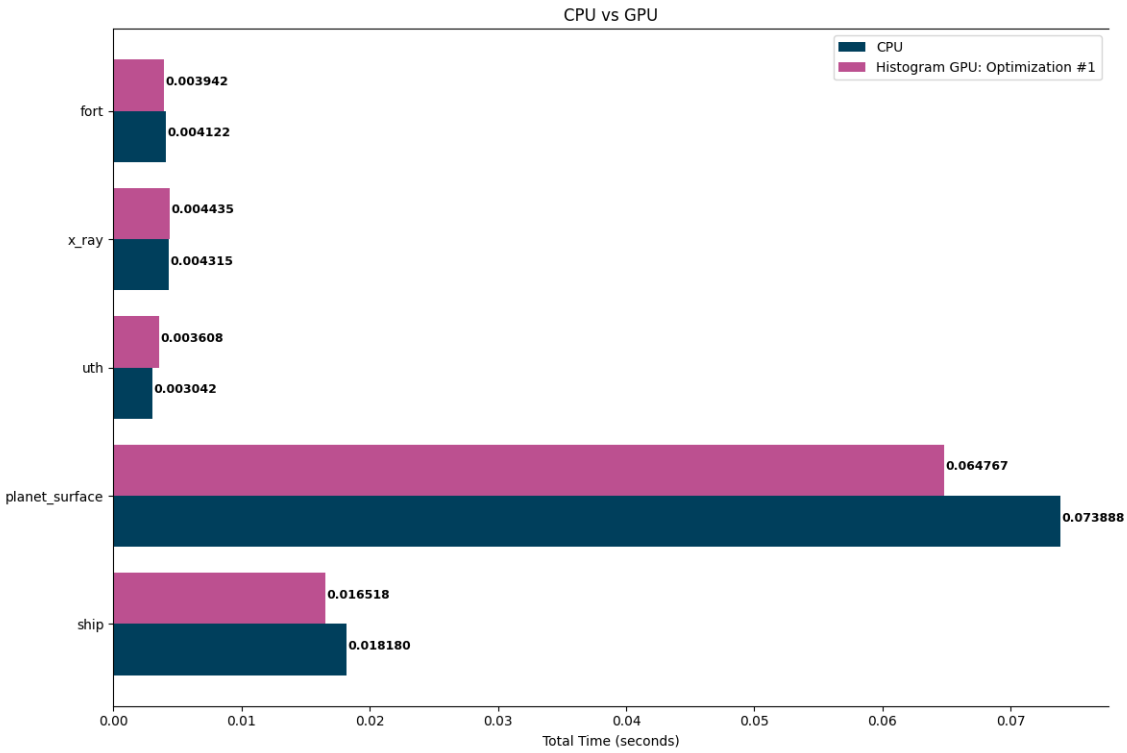

Figure 4: CPU vs GPU with Optimized Histogram Calculation

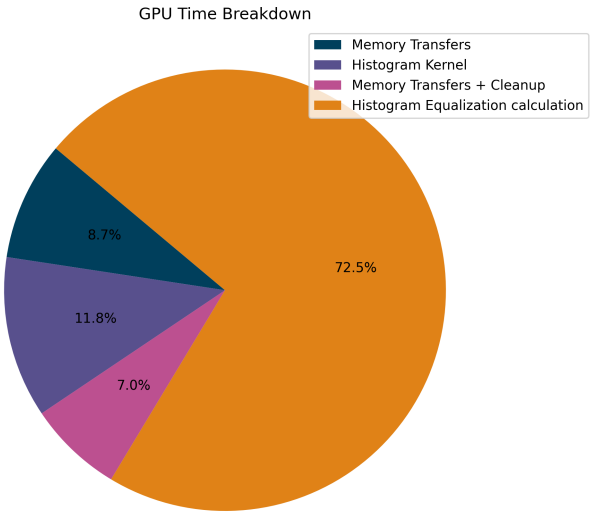

Figure 5: GPU Time distribution

In conclusion, the revised implementation significantly improves the efficiency of the histogram calculation kernel by addressing the bottlenecks present in the naive approach. First, reduced global memory contention is achieved by utilizing a shared

memory array, which enables threads to increment bins locally within each block. This eliminates the frequent atomic operations on global memory , a major source of congestion in the initial implementation. Additionally, increased parallelism is realized by allowing threads within a block to work more independently, minimizing simultaneous access to the same histogram bin and only performing global memory updates once per block. Finally, these optimizations result in improved performance when comparing the histogram calculation times of the CPU with the ones of the GPU (not the overall execution times).

3.3 Aggregation

In our third attempt to optimize the histogram calculation kernel, we introduced aggregation to further enhance performance by reducing the number of atomicAdd operations. Aggregation involves accumulating multiple histogram updates within each thread before performing a single atomic operation, thereby minimizing contention and improving efficiency.

This implementation is particularly effective for our dataset, which consists of images. Since images often contain regions of similar pixel intensity, there is a high likelihood of encountering consecutive pixels with the same value. This natural property of image data significantly benefits from the aggregation strategy, as threads can accumulate counts for these identical pixel values locally before updating the histogram.

To maximize the advantages of aggregation, we implemented Contiguous Partitioning. By ensuring that each thread will process contiguous segments of the image, the probability of encountering consecutive pixels with the same value increases. This design further enhances the efficiency of the aggregation process by reducing the number of atomic operations required, especially in areas of the image with uniform or gradual intensity changes.

3.3.1 Contiguous Partitioning

From Figures 6, 7, we observe that the aggregation optimization resulted in a significant decrease in the computation time of the histogram equalization kernel when compared both to the CPU as well as the previous implementation of the histogram kernel.

```
__global__ void histogram_calc(int *hist_out, unsigned char *img_in, int img_size, int nbr_bin)
    __shared__ int private_hist[256];

    int i = threadIdx.x + blockIdx.x*blockDim.x;
    int it = 0, accum = 0, prev_pixel_val = -1;

    it = i*CFACTOR;
    private_hist[threadIdx.x] = 0;
    __syncthreads();

    while (it < min(img_size, (i+1)*CFACTOR)) {
        int pixel_val = img_in[it];
        if (pixel_val == prev_pixel_val) {
            accum++;
        }
        else{
            if (accum > 0) {
                atomicAdd(&(private_hist[prev_pixel_val]), accum);
            }
            accum = 1;
            prev_pixel_val = pixel_val;
        }
        it++;
    }

    if (accum > 0) {
        atomicAdd(&(private_hist[prev_pixel_val]), accum);
    }

    __syncthreads();

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

Listing 3: Histogram Calculation: Aggregation with Contiguous thread partitioning

3.3.2 Interleaved partitioning

For the next implementation, we experimented with the same aggregation-based kernel but replaced Contiguous Partitioning with Interleaved Partitioning. In this approach, each thread processes non-contiguous pixels, distributed across the image in an interleaved manner. This modification allowed us to analyze the impact of partitioning strategies on performance and better understand the advantages of Contiguous Partitioning for our dataset.

By using Interleaved Partitioning, the likelihood of encountering consecutive pixels with the same value was reduced, especially in regions of the image with uniform intensity. This change highlighted how Contiguous Partitioning leverages the spatial

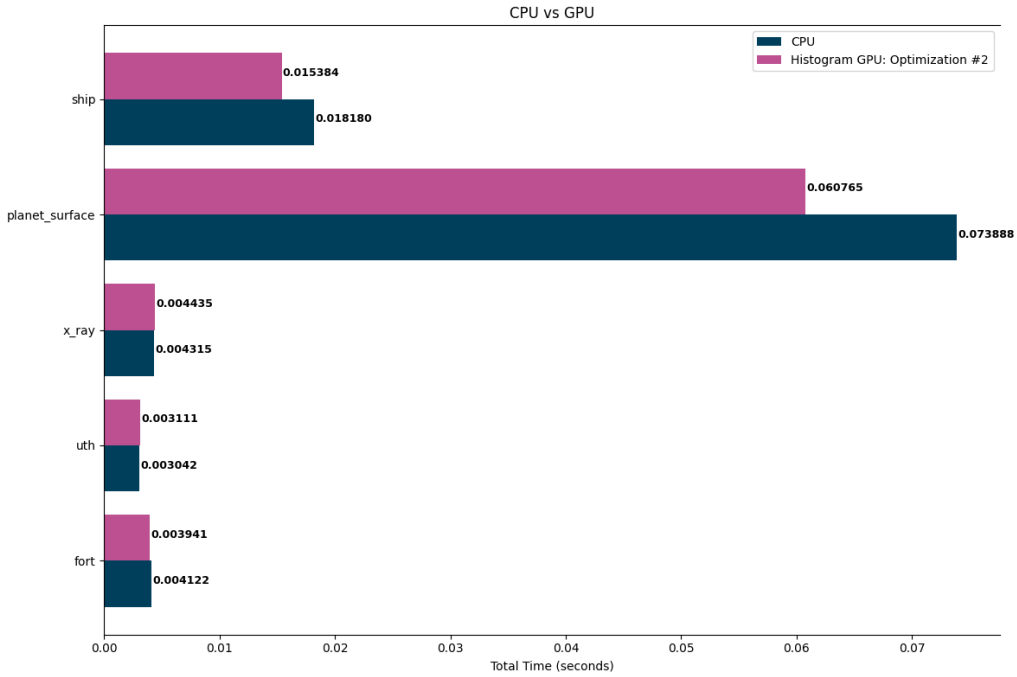

Figure 6: Average execution time comparison between CPU and GPU with optimized histogram kernel

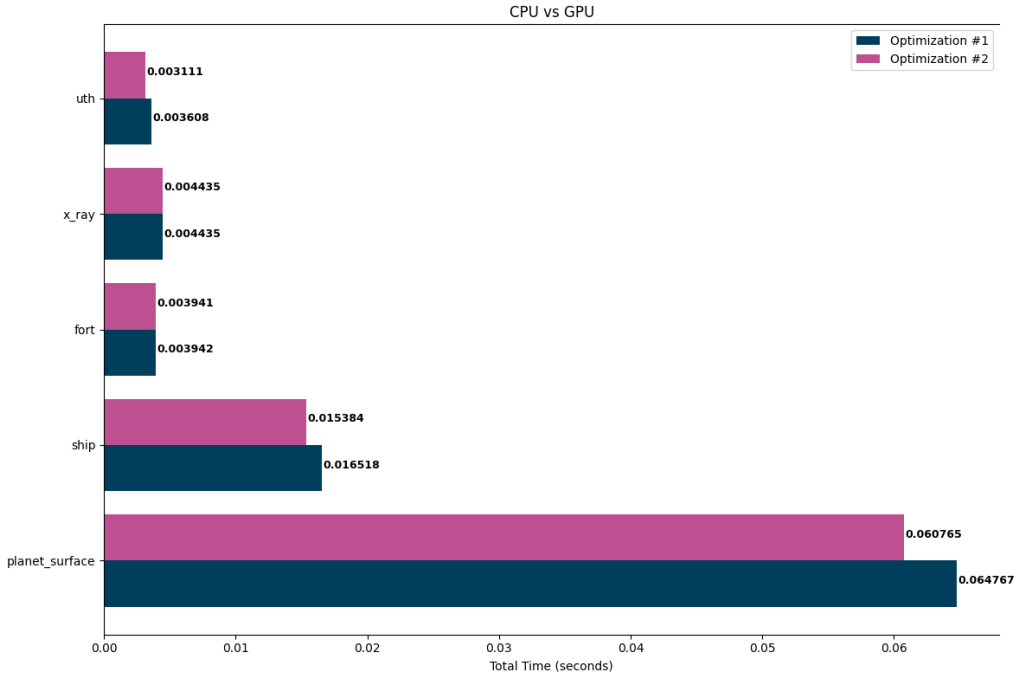

Figure 7: Average GPU execution time comparison between Privatization and Aggregation Optimizations

coherence of image data to optimize aggregation, demonstrating the significant benefit of aligning the partitioning strategy with the dataset's inherent properties.

Listing 4 shows the CUDA implementation of aggregation with interleaved threada partitioning.

```
__global__ void histogram_calc(int *hist_out, unsigned char *img_in, int img_size, int nbr_
    __shared__ int private_hist[256];

    int i = threadIdx.x + blockIdx.x*blockDim.x, stride = blockDim.x * gridDim.x;
    int it = 0, accum = 0, prev_pixel_val = -1;

    it = i;
    private_hist[threadIdx.x] = 0;
    __syncthreads();

    while (it < img_size) {
        int pixel_val = img_in[it];
        if (pixel_val == prev_pixel_val) {
            accum++;
        }
        else{
            if (accum > 0) {
                atomicAdd(&(private_hist[prev_pixel_val]), accum);
            }
            accum = 1;
            prev_pixel_val = pixel_val;
        }
        it+=stride;
    }

    if (accum > 0) {
        atomicAdd(&(private_hist[prev_pixel_val]), accum);
    }

    __syncthreads();

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

Listing 4: Histogram Calculation: Aggregation with Interleaved Thread Partitioning

As anticipated, the interleaved partitioning method did not perform as well as the contiguous partitioning. This difference in performance is likely attributed to the characteristics of our dataset.

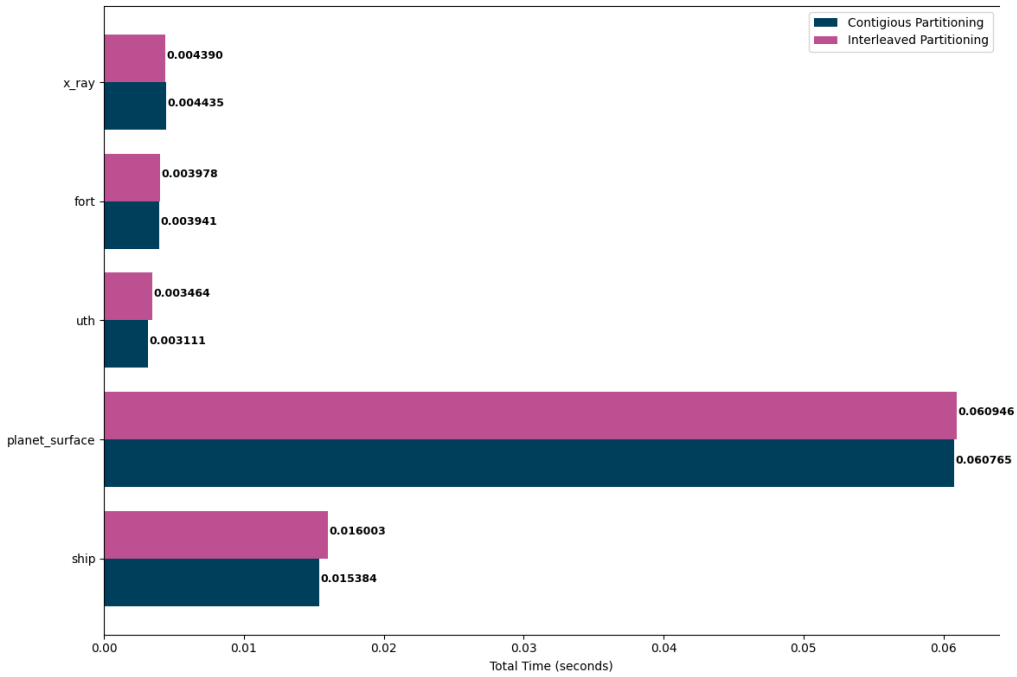

Figure 8: Comparison of Aggregation with Contiguous vs Interleaved partitioning

4 Histogram Equalization Kernel

At this point, we have optimized the histogram calculation kernel and as seen in Figure 9, the histogram equalization calculation takes up 72% of the total execution time since it is not executed on the GPU. In this section, we implement and optimize a new kernel that executes the histogram equalization kernel on the GPU.

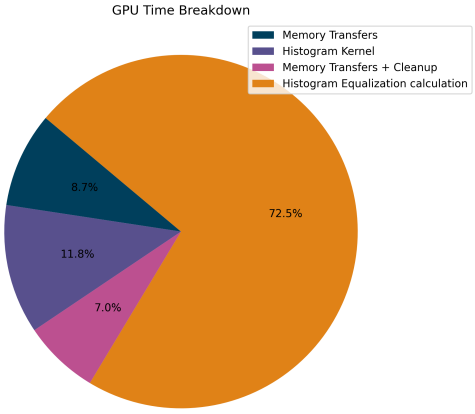

Figure 9: GPU Time distribution

4.1 Naive Implementation

In this kernel, we implement the basic functionality for histogram equalization by applying a lookup table (LUT) to adjust the pixel values of an input image (d_img_in) and store the results in the output image (d_img_out). Each thread is assigned to process a single pixel using the unique thread ID (tid).

```
__global__ void histogram_equ(unsigned char *d_img_out, unsigned char *d_img_in, int *d_lut) {
    int tid = threadIdx.x + blockIdx.x*blockDim.x;

    if (d_lut[d_img_in[tid]] > 255) {
        d_img_out[tid] = 255;
    }
    else {
        d_img_out[tid] = (unsigned char)d_lut[d_img_in[tid]];
    }
}
```

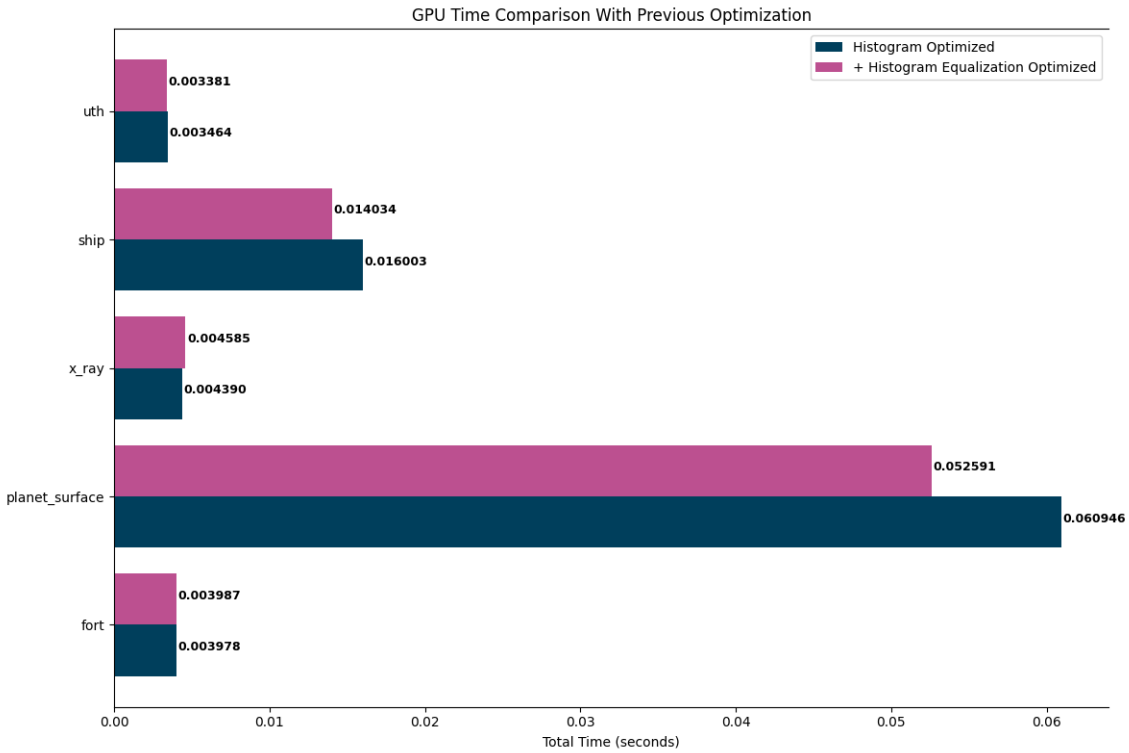

Figure 10: Performance comparison of GPU code with and without Histogram Equalization Kernel

As expected, executing the histogram equalization on the gpu, led to better execution times, as can be seen from Figure 10. Also, from Figure 11, we can see the difference

that the new kernel made in the overall GPU time distribution since now it takes even less than time than the histogram kernel. The new overhead now is added by the new memory transactions.

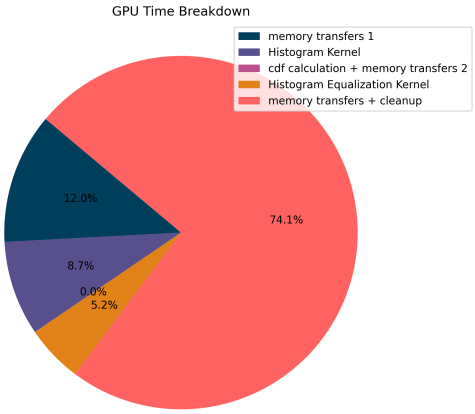

Figure 11: GPU Time distribution

4.2 Privatization

In this stage, we optimize the histogram equalization kernel using privatization and interleaved partitioning of threads to improve memory access patterns and overall performance. The main enhancement involves the use of shared memory to reduce frequent accesses to global memory.

- **Exclude img_out**: To eliminate unnecessary memory operations, we stop using a separate output array (img_out) and instead make the changes directly in the input array (img_in). This avoids redundant memory allocations and the overhead of copying data back and forth between arrays, further improving efficiency.

- **Shared Memory Allocation**: The main enhancement involves the use of shared memory to reduce frequent accesses to global memory. A private lookup table (priv_lut) is created in shared memory, where each thread in the block loads a portion of the global LUT (d_lut). This ensures coalesced memory access during the initialization phase and minimizes global memory access during computation. The condition if (threadIdx.x ¡ 256) ensures that only the first 256 threads in the block participate in loading the LUT.

- **Interleaved partitioning**: Next, each thread processes a portion of the input image, using a stride, exactly as implemented in section 3.2. This approach ensures that threads access image pixels spaced evenly across the entire image, promoting coalesced memory access when reading and writing.

```
__global__ void histogram_equ(unsigned char *d_img_in, int *d_lut, int img_size) {
    int tid = threadIdx.x + blockIdx.x*blockDim.x,
        stride = blockDim.x * gridDim.x;
    __shared__ int priv_lut[256];

    if (threadIdx.x < 256) {
        priv_lut[threadIdx.x] = d_lut[threadIdx.x];
    }
    __syncthreads();

    while (tid < img_size) {
        d_img_in[tid] = (unsigned char)priv_lut[d_img_in[tid]];
        tid += stride;
    }
}
```

As we can see from Figure 12, the execution time of the optimized kernel is indeed superior to the previous one.

5 CDF Kernel

Finally, we choose to implement the CDF that creates the LUT (needed by the histogram equalization kernel) on the GPU. The execution time of the CDF on the CPU is already really low, however, by creating a new kernel and computing the LUT on device, we can further reduce host-device memory transfers while also keeping the cost of the CDF calculation relatively low.

5.1 Privatization

The CDF kernel is compact and efficient, requiring only number_bin threads to compute the complete CDF and LUT. To achieve this, we configure a single block with 256 threads. Shared memory (priv_hist) is employed to store histogram values, providing faster access for all threads within the block. Each thread performs a

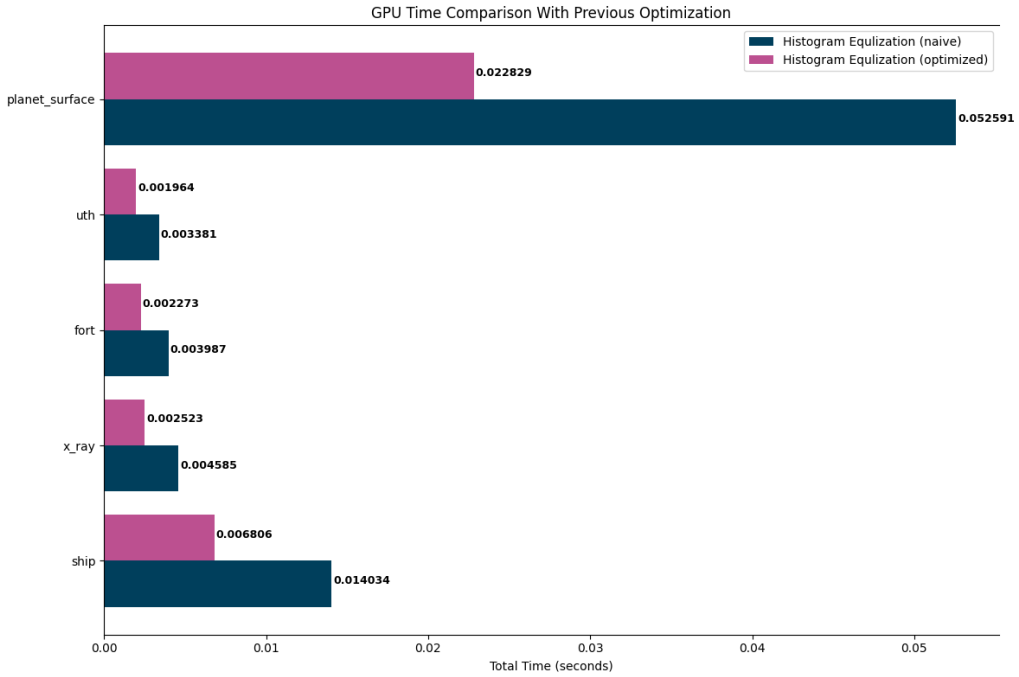

Figure 12: Average GPU time comparison between naive and optimized histogram equalization kernels

prefix sum operation to calculate its portion of the CDF, which is then utilized to compute a corresponding value in the LUT.

```
1  // CDF kernel
2  __global__ void cdf_calc(int *d_lut, int *d_hist, int img_size, int nbr_bin) {
3      int min = 0, d, cdf = 0, idx = 0;
4
5      __shared__ int priv_hist[256];
6
7      if (threadIdx.x < 256) {
8          priv_hist[threadIdx.x] = d_hist[threadIdx.x];
9      }
10     __syncthreads();
11
12     while (min == 0) {
13         min = priv_hist[idx++];
14     }
15     d = img_size - min;
16
17     for (int i = 0; i <= threadIdx.x; i++) {
18         cdf += priv_hist[i];
19     }
```

```
20
21      d_lut[threadIdx.x] = max((int)(((float)cdf - min)*255/d + 0.5), 0);
22  }
```

As expected, the CDF kernel doesn't have a big impact on performance, when compared with the previous implementation, but now we have less memory transactions between host and device. Figure 13 shows the results for this optimization.

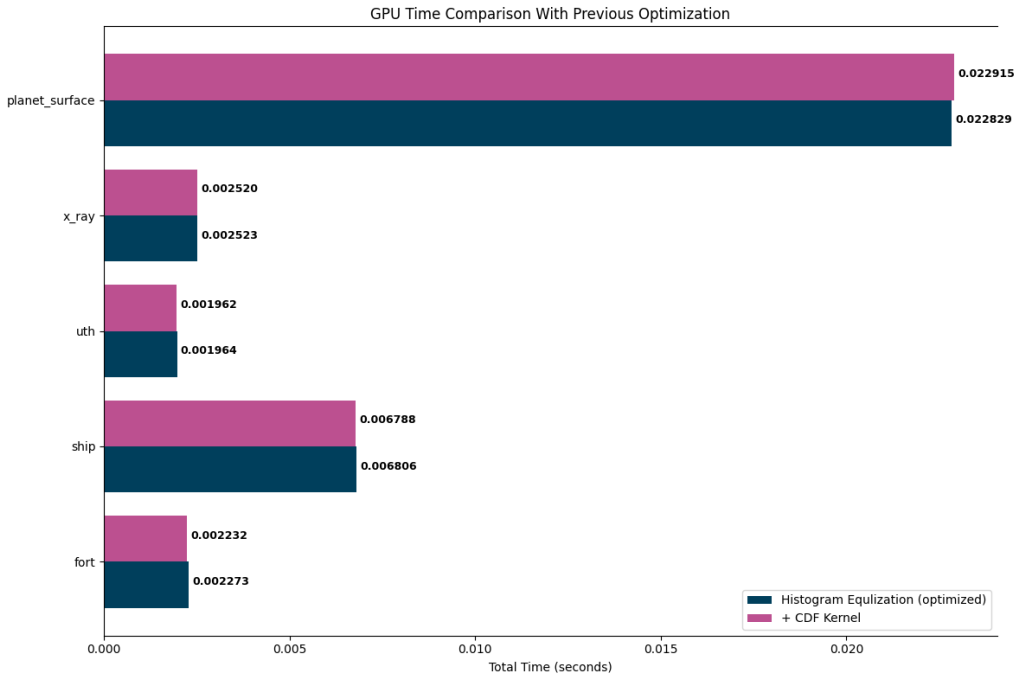

Figure 13: Comparison between previous implementation and CDF Kernel privatization

5.2 Inplace computation of LUT in histogram

In the final implementation of the CDF kernel, we further optimize the computation by eliminating unnecessary memory operations. Instead of using a separate output array (lut), we directly modify the d_hist array. This further reduces memory allocations and transfers, leaving us only with one memory transfer of the input image which we will try to optimize in the next sections.

```
1  // CDF kernel
2  __global__ void cdf_calc(int *d_hist, int img_size, int nbr_bin) {
3      int min = 0, d, cdf = 0, idx = 0;
4
```

```
    __shared__ int priv_hist[256];

    if (threadIdx.x < 256) {
        priv_hist[threadIdx.x] = d_hist[threadIdx.x];
    }
    __syncthreads();

    while (min == 0) {
        min = priv_hist[idx++];
    }
    d = img_size - min;

    for (int i = 0; i <= threadIdx.x; i++) {
        cdf += priv_hist[i];
    }

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

Again, as expected, performing an in place computation of the LUT inside the histogram variable, further reduces the total execution time since it eliminates one more host-device memory transaction.

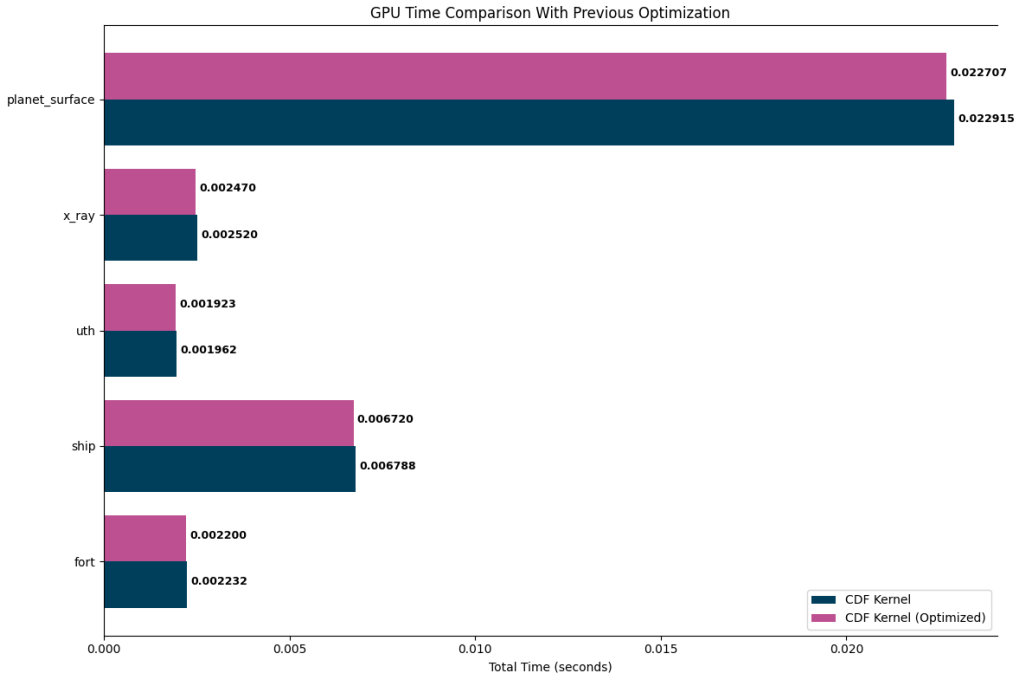

Figure 14: Previous implementation vs Inplace calculation of LUT

6 Memory Optimizations

After optimizing all kernels, we observe that the bottleneck is now at the memory transactions between the device and host. We have already minimized some memory transactions by implementing in-place algorithms in each kernel. However, a big fraction of the execution time still belongs to copying the input image from the host to the device. In order to minimize the time of this transactions we implement the following memory optimizations.

6.1 Pinned Memory

First, we use the pinned memory in order to allocate space for the input image on the host. Pinned memory is a region of host memory that cannot be paged out to disk by the OS, allowing faster data transfers to and from the GPU. In this section, we replace malloc calls for the input image memory allocations with the cudaAllocHost API function. Doing so, the input image is now allocated in pinned memory.

Figure 15, shows the average execution times for different input images with and without pinned memory. As expected, storing the input image in pinned memory gave us a small performance gain.

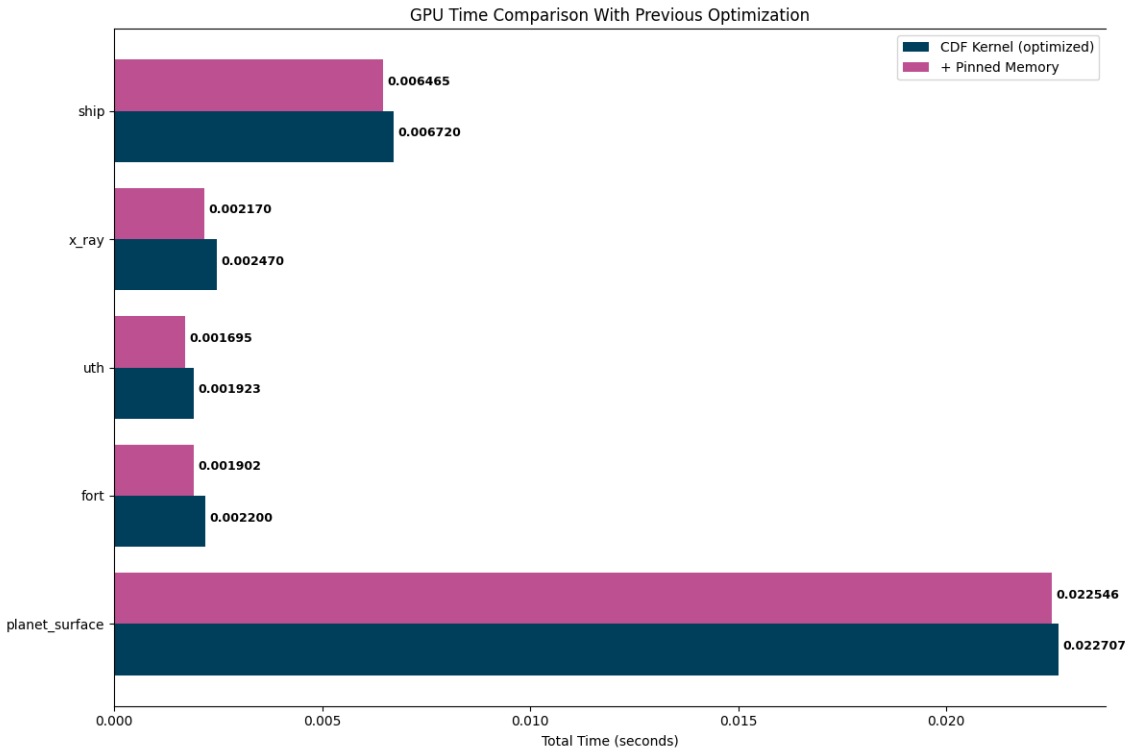

Figure 15: GPU time comparison with and without Pinned Memory

6.2 Zero-copy Memory

Next, we attempt to completely eliminate the memory transfer of the input image from the host to the device by using zero-copy memory. Here, we still allocate the input image on pinned memory on the host using cudaHostAlloc but we also use two additional flags provided by the cuda API, cudaHostAllocMapped which maps the memory allocated on the host in the memory space of the device allowing direct access, and cudaHostAllocWriteCombined which allows multiple small writes to memory to be combined together into a single burst transaction when sending data over the PCI Express (PCIe) bus to the GPU. Doing so, the device can now access memory thats allocated on the host by using a pointer to this memory.

Figure 16 shows the execution times after completely eliminating the memory transfer of the input image from the host to the device. Here, we observe a big performance hit since now every time the device needs access to an element from the input image, the data has to be transferred through the PCIe bus.

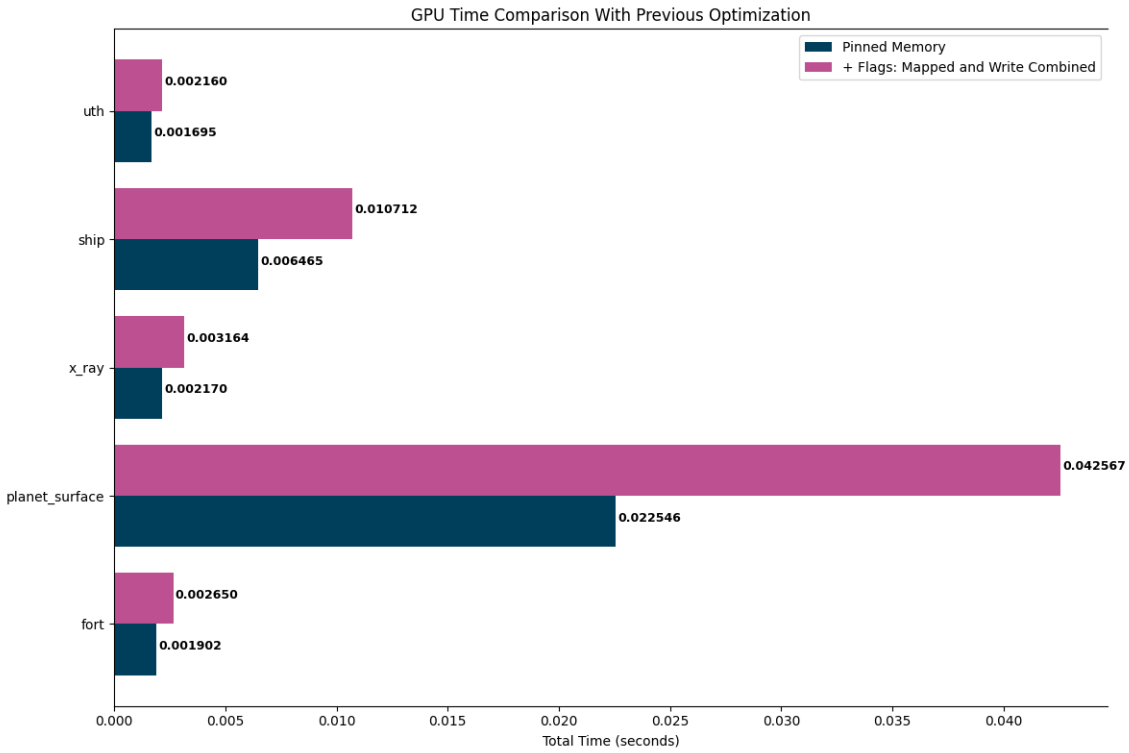

Figure 16: GPU time comparison Pinned Memory VS Zero-copy Memory

6.3 Unified Memory

Unified memory provides a single memory space that is shared between the CPU and GPU, allowing both to access and operate on the same data without requiring explicit memory copies or manual synchronization. Besides the shared memory address space, unified memory also provides automatic data migration. When the GPU accesses data located in Unified Memory for the first time, the necessary data is automatically moved from the CPU to the GPU. The data is still moved through the PCIe bus but cuda runtime handles these migrations seamlessly with a combination of page-faulting and pre-fetching. For this optimization, we store the input image in unified memory using cudaMallocManaged, and thus completely eliminating the need of copying the input image to device memory.

Figure 17 shows the performance boost gained by unified memory. By eliminating all memory transfers between host and device and using the automatic data migration of unified memory, this optimization outperforms all previous ones (pinned and standard memory) and gives us a speedup of up to x1.9 when compared to the previous optimization for larger images.

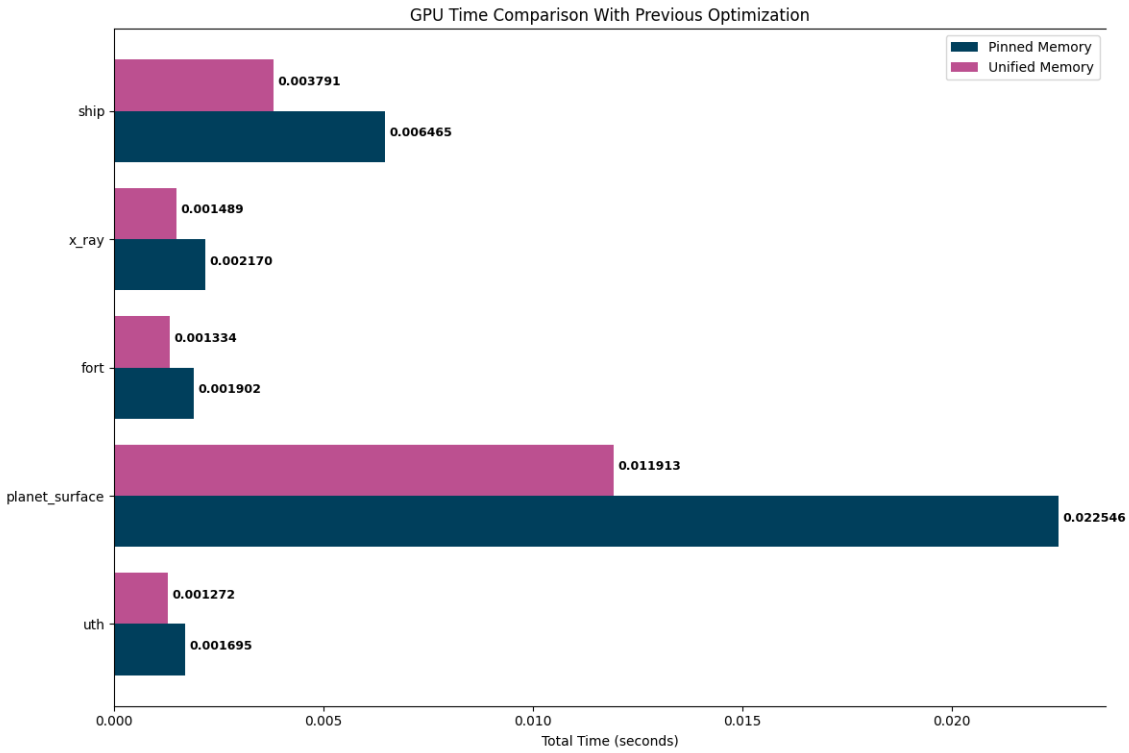

Figure 17: Unified Memory Performance Comparison with Pinned Memory

6.4 Texture Memory

Finally, for the last memory optimization, we chose to store data on the texture memory. Even though texture memory is part of the global memory on the GPU, it benefits from a specialized texture cache, thus making it the perfect candidate for efficient read-only memory accesses. Since here we cannot write in this type of memory, we are limited to storing only elements that we use for reading such as the LUT after it is calculated by the cdf kernel. By storing the LUT in texture memory, we benefit from less memory accesses to global memory (since most of the values will be stored in caches).

Figure 18 shows the results of this optimization. As we can see, texture memory didn't have an impact on the overall performance for the set of images used for testing. When executed for an even larger image than the ones used here we observed a small speedup x1.005 which we however discarded due to a slightly larger standard deviation in the final results for that specific experiment.

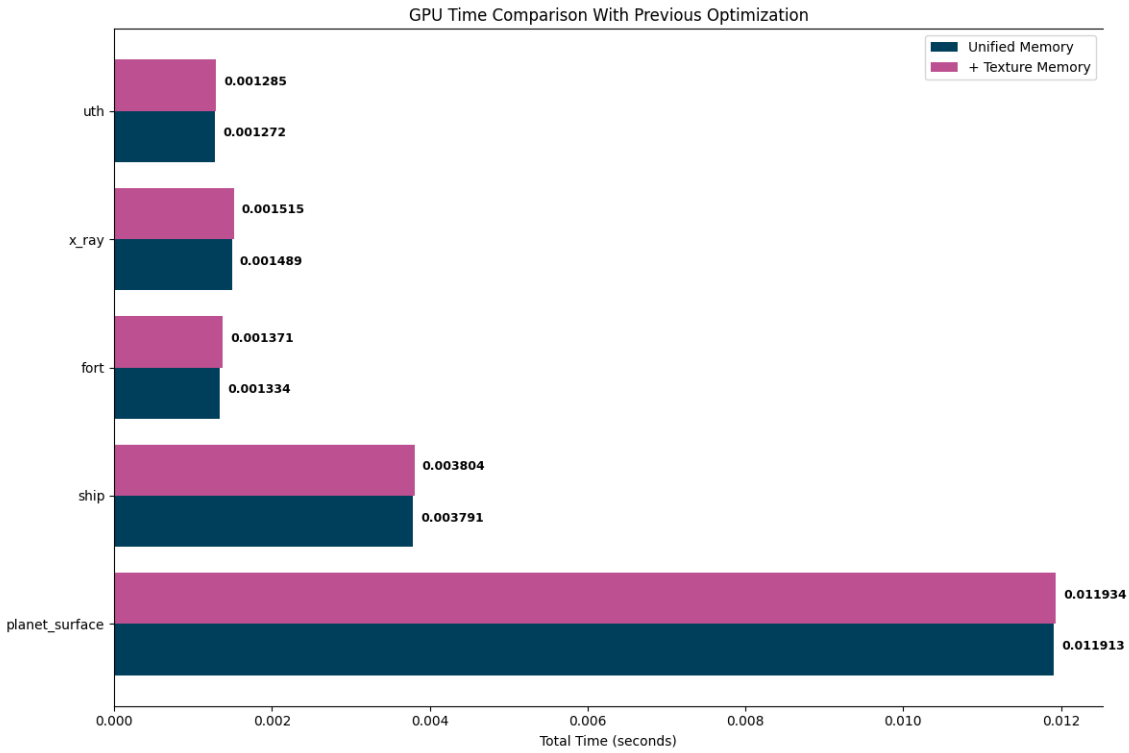

Figure 18: Comparison between Unified and Unified+Texture memory optimizations

7 Streams

In our next optimization attempt, we leveraged two CUDA streams to perform simultaneous memory copy operations using cudaMemcpyAsync() while running our kernels. Specifically, we divided the input image (img_in) into two separate images and transferred these parts concurrently from the CPU to the GPU and vice versa. By utilizing multiple streams for memory transfers, we aimed to overlap the data transfer with kernel execution, improving overall throughput and reducing the time spent on memory transfers.

Although this approach resulted in some improvement compared to using global memory, it did not outperform the unified memory model, as we can see from figures 20 and 21 respectively. The reason for this is that unified memory allows the GPU and CPU to share a single memory space, eliminating the need for explicit memory copying between the two. With unified memory, the system automatically handles memory transfers, ensuring that the data is available where it is needed without requiring manual intervention or additional overhead from stream management.

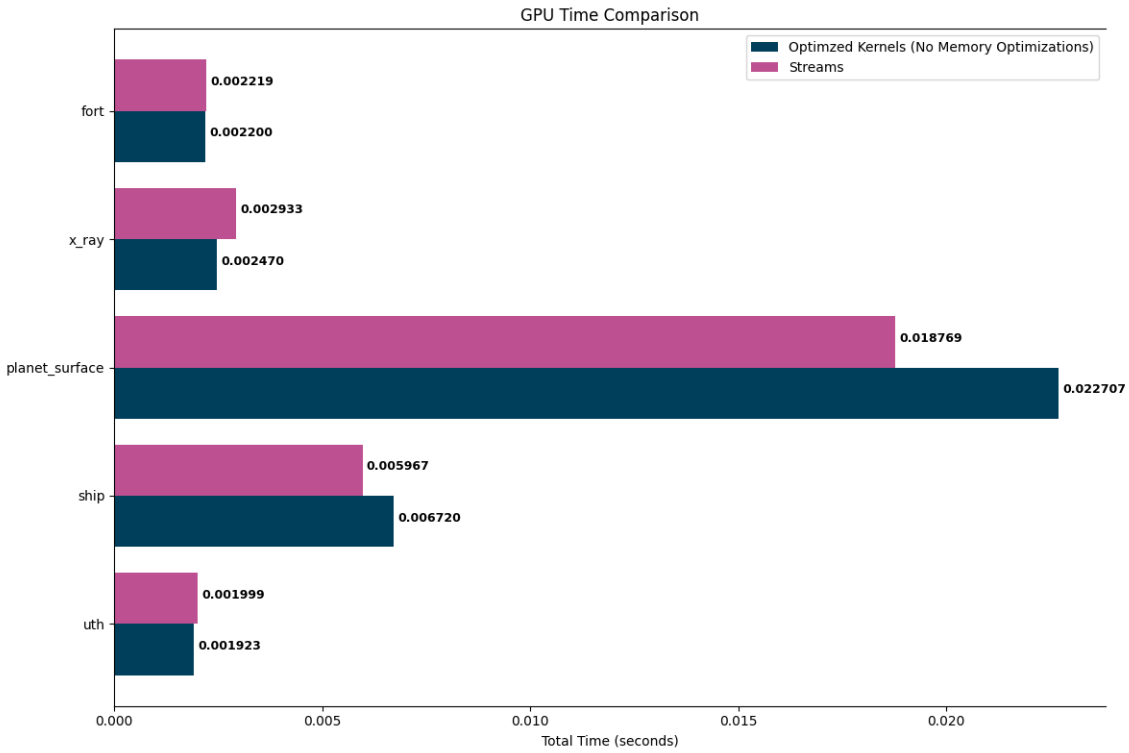

Figure 19: Performance comparison of serialized vs overlapped memory transfer and kernel execution using streams

8 Conclusion

In conclusion, our optimizations for histogram equalization on the GPU demonstrated significant performance improvements through careful kernel design and memory optimizations. By leveraging contiguous partitioning, aggregation techniques, and privatization, we reduced the number of atomic operations and improved memory access patterns, achieving faster execution times. Memory optimizations, including pinned memory, zero-copy memory, unified memory, and texture memory, further enhanced performance by minimizing host-device memory transfers and leveraging efficient caching mechanisms. Unified memory proved particularly effective, offering a speedup of up to 1.9x for larger images. These results underscore the importance of tailoring memory and computational strategies to the specific characteristics of the dataset and GPU architecture for achieving optimal performance. In Figure 21 we present a final graph of all the optimizations performed. The graphs shows the mean execution time of the gpu application for all optimizations. When compared to the CPU, our implementation achieved a significant speedup of x6.7 times faster

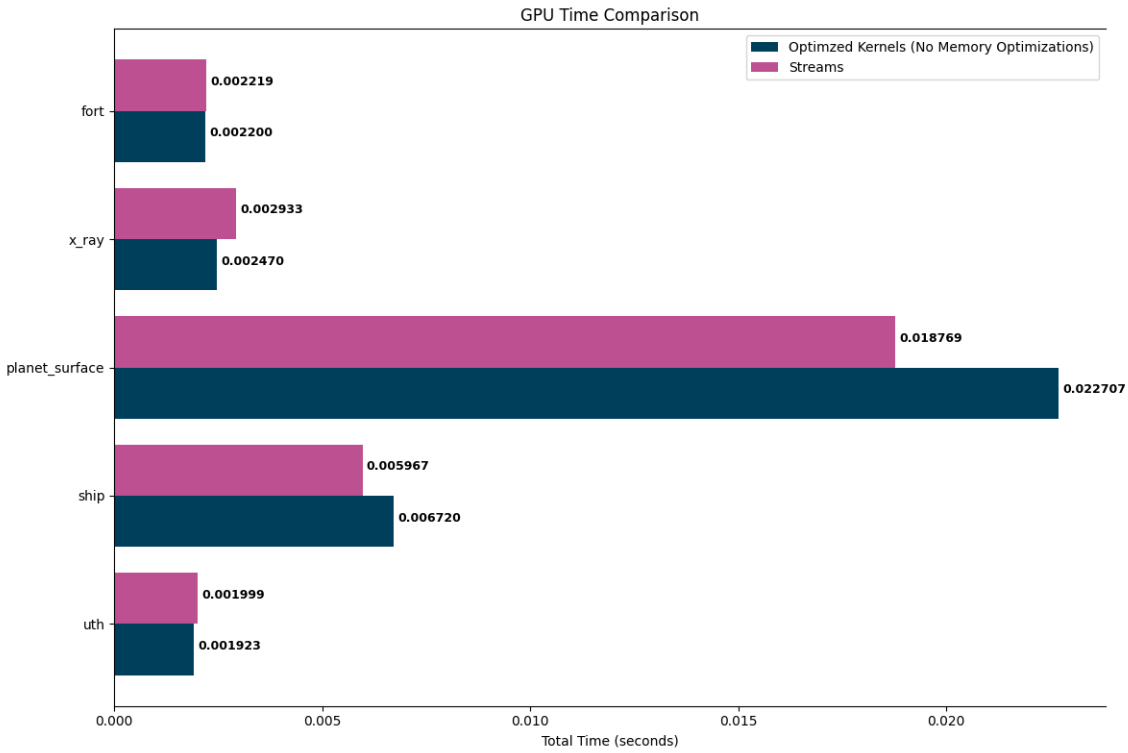

Figure 20: Performance comparison of Unified memory vs Stream implementation

than the CPU implementation. The execution times for this graph were measured for a large

A Device Query Outputs

In the following sections we provide the output of deviceQuery; NVIDIA's sampe utility that provides detailed information about the GPU(s) on a system. The GPUs that we used for the evaluation of all optimizations are shown in the following Listing.

A.0.1 Venus

```
1   ./deviceQuery Starting...
2
3  CUDA Device Query (Runtime API) version (CUDART static linking)
4
5 Detected 4 CUDA Capable device(s)
6
```

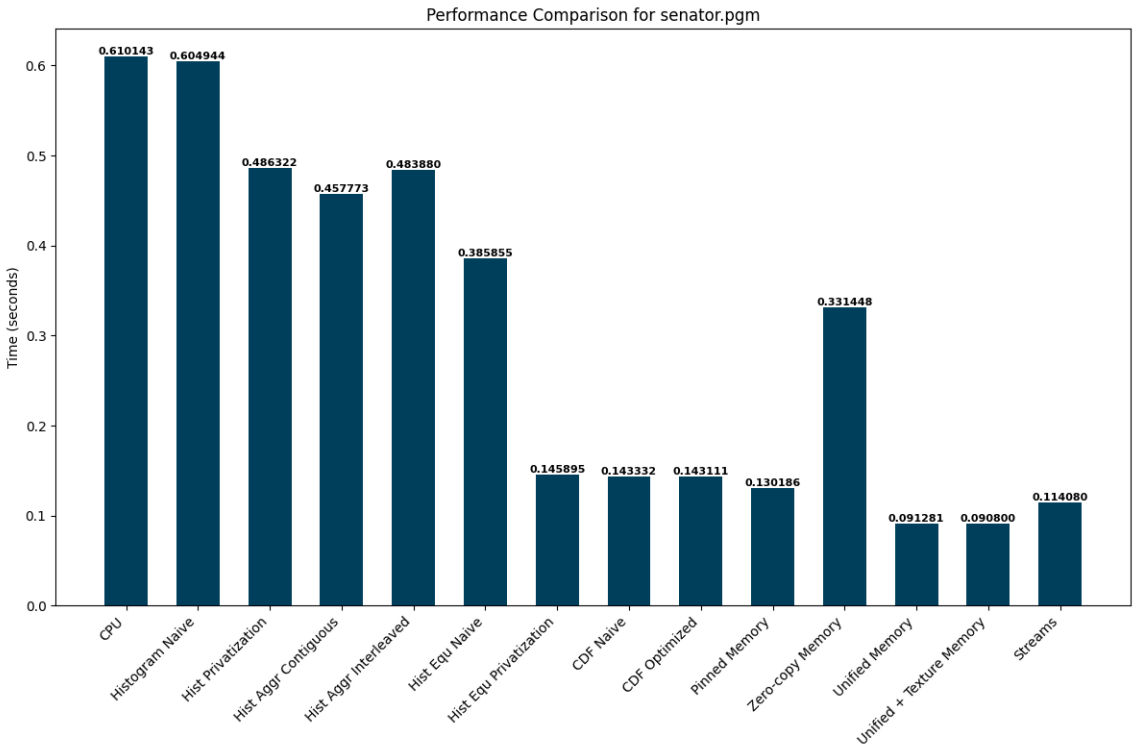

Figure 21: Average Execution Times of each implementation

```
Device 0: "Tesla K80"
  CUDA Driver Version / Runtime Version          11.4 / 11.5
  CUDA Capability Major/Minor version number:    3.7
  Total amount of global memory:                 11441 MBytes
     (11997020160 bytes)
  (13) Multiprocessors, (192) CUDA Cores/MP:     2496 CUDA Cores
  GPU Max Clock rate:                            824 MHz (0.82 GHz)
  Memory Clock rate:                             2505 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 1572864 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D
     =(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048
     layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384),
      2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
```

```
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535,
      65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy
     engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Compute Preemption:            No
  Supports Cooperative Kernel Launch:            No
  Supports MultiDevice Co-op Kernel Launch:      No
  Device PCI Domain ID / Bus ID / location ID:   0 / 6 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice()
        with device simultaneously) >

Device 1: "Tesla K80"
  CUDA Driver Version / Runtime Version          11.4 / 11.5
  CUDA Capability Major/Minor version number:    3.7
  Total amount of global memory:                 11441 MBytes
     (11997020160 bytes)
  (13) Multiprocessors, (192) CUDA Cores/MP:     2496 CUDA Cores
  GPU Max Clock rate:                            824 MHz (0.82 GHz)
  Memory Clock rate:                             2505 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 1572864 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D
     =(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048
     layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384),
      2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
```

```
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535,
      65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy
     engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Compute Preemption:            No
  Supports Cooperative Kernel Launch:            No
  Supports MultiDevice Co-op Kernel Launch:      No
  Device PCI Domain ID / Bus ID / location ID:   0 / 7 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice()
        with device simultaneously) >

Device 2: "Tesla K80"
  CUDA Driver Version / Runtime Version          11.4 / 11.5
  CUDA Capability Major/Minor version number:    3.7
  Total amount of global memory:                 11441 MBytes
     (11997020160 bytes)
  (13) Multiprocessors, (192) CUDA Cores/MP:     2496 CUDA Cores
  GPU Max Clock rate:                            824 MHz (0.82 GHz)
  Memory Clock rate:                             2505 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 1572864 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D
     =(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048
     layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384),
      2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
```

```
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535,
      65535)
  Maximum memory pitch:                         2147483647 bytes
  Texture alignment:                            512 bytes
  Concurrent copy and kernel execution:         Yes with 2 copy
     engine(s)
  Run time limit on kernels:                    No
  Integrated GPU sharing Host Memory:           No
  Support host page-locked memory mapping:      Yes
  Alignment requirement for Surfaces:           Yes
  Device has ECC support:                       Enabled
  Device supports Unified Addressing (UVA):     Yes
  Device supports Compute Preemption:           No
  Supports Cooperative Kernel Launch:           No
  Supports MultiDevice Co-op Kernel Launch:     No
  Device PCI Domain ID / Bus ID / location ID:  0 / 132 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice()
        with device simultaneously) >

Device 3: "Tesla K80"
  CUDA Driver Version / Runtime Version         11.4 / 11.5
  CUDA Capability Major/Minor version number:   3.7
  Total amount of global memory:                11441 MBytes
     (11997020160 bytes)
  (13) Multiprocessors, (192) CUDA Cores/MP:    2496 CUDA Cores
  GPU Max Clock rate:                           824 MHz (0.82 GHz)
  Memory Clock rate:                            2505 Mhz
  Memory Bus Width:                             384-bit
  L2 Cache Size:                                1572864 bytes
  Maximum Texture Dimension Size (x,y,z)        1D=(65536), 2D
     =(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers 1D=(16384), 2048
     layers
  Maximum Layered 2D Texture Size, (num) layers 2D=(16384, 16384),
      2048 layers
  Total amount of constant memory:              65536 bytes
  Total amount of shared memory per block:      49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                    32
  Maximum number of threads per multiprocessor: 2048
  Maximum number of threads per block:          1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
```

```
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535,
      65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy
     engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Compute Preemption:            No
  Supports Cooperative Kernel Launch:            No
  Supports MultiDevice Co-op Kernel Launch:      No
  Device PCI Domain ID / Bus ID / location ID:   0 / 133 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice()
        with device simultaneously) >
> Peer access from Tesla K80 (GPU0) -> Tesla K80 (GPU1) : Yes
> Peer access from Tesla K80 (GPU0) -> Tesla K80 (GPU2) : No
> Peer access from Tesla K80 (GPU0) -> Tesla K80 (GPU3) : No
> Peer access from Tesla K80 (GPU1) -> Tesla K80 (GPU0) : Yes
> Peer access from Tesla K80 (GPU1) -> Tesla K80 (GPU2) : No
> Peer access from Tesla K80 (GPU1) -> Tesla K80 (GPU3) : No
> Peer access from Tesla K80 (GPU2) -> Tesla K80 (GPU0) : No
> Peer access from Tesla K80 (GPU2) -> Tesla K80 (GPU1) : No
> Peer access from Tesla K80 (GPU2) -> Tesla K80 (GPU3) : Yes
> Peer access from Tesla K80 (GPU3) -> Tesla K80 (GPU0) : No
> Peer access from Tesla K80 (GPU3) -> Tesla K80 (GPU1) : No
> Peer access from Tesla K80 (GPU3) -> Tesla K80 (GPU2) : Yes

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 11.4, CUDA
    Runtime Version = 11.5, NumDevs = 4, Device0 = Tesla K80,
   Device1 = Tesla K80, Device2 = Tesla K80, Device3 = Tesla K80
Result = PASS
```

Listing 1: Output of deviceQuery on Venus