CSS535: High-Performance Computing

Accelerated Image Restoration using CUDA

Kishan Nagendra
March 20, 2022
University of Washington, Bothell

Table of Contents

I. ABSTRACT

With advancements in technology, there is still a large amount of image data degraded in terms of quality. Using low-quality equipment to capture images and external environmental factors could result in a degraded image. Degradation can come in multiple forms like motion blur and noise. Application involving medical images, astronomical and forensic images would have to be restored for better detection, diagnosis, and treatment. The proposed project consists of accelerating the reconstruction of images by parallelization techniques using CUDA. The acceleration is demonstrated by showcasing the restoration speed in different hardware components(CPU and GPU) and highlighting performance. The image restoration algorithm involves optimizing the mathematical computations applied to the degraded image to result in a better resolution image. The process is done heuristically without using in-built libraries for most of the code to eliminate library dependencies. The algorithm is thus written from scratch to ensure the acceptable use of native language capabilities. In the second stage, the algorithm is optimized for each loop to enhance performance. The performance analysis report for the algorithm's runtime is shown in the results. The promising results indicate a gain in acceleration achieved in the performance with a change in hardware and algorithm design.

II. INTRODUCTION

In this paper, an accelerated image restoration strategy is proposed and demonstrated by providing a performance comparison of the implementation pipeline in different hardware architectures. Most of the time, an image is never an exact representation of the object under observation but is always corrupted by the system itself (Dougherty et al., 2001)**.** The advent of technology has inevitably created a demand for time-saving techniques in scientific computation. The need for real-time image processing has created a gap in the computation domain due to limitations bound by hardware and processors. This has led to a rise in the demand for processors with higher processing rates and compactness. Handheld cameras are one of the applications where compactness and high-speed processing are required. Traditionally, images are equipped with systems to only capture images. There is a growing demand for compact handheld devices to incorporate filter applications and other image processing units. One prominent application is the restoration of images which faces two fundamental issues. One is they include traditional structures in the algorithm, which is used for image processing. Second, these techniques are not adapted for modern hardware such as edge devices used to mitigate latency.

The increase in the number of processors has pushed towards making algorithms more modular and loosely coupled. This is because the reduction in the interdependence between components of algorithms gives scope for parallel approaches. The parallel techniques chosen rely on the intermediate computational dependencies, and it is important to ensure that the algorithms are not embarrassingly parallel. To achieve this, a parallel programming platform needs to be chosen to align with processing units that have the capability for parallel processing. CUDA and Nvidia GPUs are the preferred platform and hardware for image restoration. The goal is to achieve a considerable gain in the computation time for image restoration.

Image restoration (or) image enhancement is taking a noisy image and estimating the original image. Noise in images typically comes due to motion blur, noise, and camera misfocus. By reversing the process that blurs the image using a point spread function (PSF), the image information lost can be regained. The Wiener filtering technique is used since it is optimal for the mean square error and is a linear estimation of the original image. It minimizes the overall mean square error in inverse filtering and noise smoothing.

The advances in computer vision have proliferated the number of algorithms used for image processing. This project aims to compensate for the deficit in image restoration speed by overcoming the hurdles faced by traditional CPU utilization. These hurdles can be recognized as sequential methods used for processing the pixels in an image.

III. RELATED WORK

The approach proposed by Herng-Hua and Cheng-Yuan makes use of a collateral filter with parallel computation for image restoration of brain magnetic resonance (MR) images (Chang et al., 2019). Initial analysis showed that the GPU-based collateral filter method provided a high Peak signal-to-noise ratio (PSNR) upwards of 30dB. The processing speedup gain increased from 34 for a single image to 541 for 100 images.

The subjective image quality assessment method (Bo Hu et al., 2020) uses rank to rate the perceptual quality of restored images. The objective method uses a feature selection strategy that yields an appropriate image quality score. The algorithm incorporated for image restoration by Wen et al. indicates the use of fast Fourier transforms to deblur an image. The focus of this paper was to better restore images in visual quality and signal-to-noise ratio than the restoration techniques developed by combining fitting data term and a regularization term. On this note, Bahman et al's digital image restoration paper looks at it from the signal processing point of view. The algorithm proposed specifically explores a technique known as regularized least squares. The paper extends the application of this algorithm specifically to optics, astronomical and medical images.

The Expectation-maximization algorithm for image restoration alternated between fast Fourier transforms and discrete wavelet transforms with the diagonalization of the convolution operator. The overall computation is iterative, requiring O(NlogN) operations per iteration (Figueiredo & Nowak, 2003). Savakis et al indicate that a geometrical point spread function can be used in the place of a physically computed point spread function to achieve significant restoration when the signal to noise ratio is less than 30dB.

The reconstruction of images, ie. image restoration, use mathematical filters applied to the degraded image to result in a high-resolution image. In the traditional spatial domain filtering techniques used for image restoration, the pixels are processed by allocating the modules of the filter operation to a grid of pixels in a sequential manner (Maru et al., 2017). This, in turn, causes a spike in processing time and memory overhead due to repeated computation of the same variables. The hurdles faced by traditional CPU utilization can be overcome through parallelization techniques and GPUs, which is the objective of this project.

IV. METHODS

Hardware and Software Setup

The CPU and GPU implementation of this project was executed on the UWB Linux Lab Machine - csslab15p.uwb.edu. The OS installed is Debian GNU/Linux V10. CPU and GPU Specifications are listed below in Table 1.

CPU	Intel(R) Core(TM) i9-9900K
Memory(CPU)	128 GB
Cores	8 (16 Threads)
Clock Speed	Base = 3.60 GHz, Max = 5.00 GHz
Instruction Per Clock Cycle	FP32 = 32, FP64 = 16
Caches	16 MB Intel® Smart Cache L1 cache - 256 KB (code) / 256 KB (data) L2 cache - 2048KB L3 cache - 16384KB
GPU	NVIDIA GeForce RTX 3070
Architecture and Compute Capability	Ampere, 8.6
Cores (GPU)	5888
Clock Speed (GPU)	1500 MHz
Memory (GPU)	8GB
Bandwidth	448.0 GB/s
Theoretical Performance	FP 32: 20.31 TFLOPS FP 64: 317.4 GFLOPS

Table 1. CPU and GPU Specifications

Tools

For the sequential CPU image restoration implementation, C++ was utilized to write the methods for each of the manually implemented steps of the process. For the parallel GPU image restoration implementation, CUDA was utilized to write the kernels for each of the manually implemented steps of the process. The CUDA version utilized is V11.2.

For all methods and kernels not implemented manually, OpenCV was utilized. This includes GaussianBlur, DFT, IDFT, and Reshape built-in functions. OpenCV V3.2.0 was utilized as it is compatible with both CPU C++ and GPU CUDA software. Additionally, OpenCV data structures such as Mat and GPU Mat were utilized to store image pixel data, manipulate this data for the restoration process, and to move this data between the host and device. OpenCV Mat and GPUMat made image manipulation more streamlined and made access to image features such as size more efficient.

Code development, compilation, and testing were completed using the IDE Visual Studio Code V1.65.2 available on the Linux Machine. As this Linux Machine has a GPU with a newer compute capability the legacy NVIDIA CUDA profiler could no longer be used for this project. The NVIDIA Nsight profiler is an extension available through VSCode and was used to assess various metrics regarding the CUDA image restoration implementation. The Nsight profile command was utilized along with additional options so that we could gain insight into the performance and memory details of our implementation. The profiler outputted timings for the different kernels, timings for data movement between the host and device, memory utilization specifically for shared and global memory. The profiler allowed us to see where our biggest time sinks and bottlenecks were.

While the Nsight profiler did compute inclusive execution time for the implementation, in this application we decided to manually compute exclusive execution time using the C++ Chronos library for comparison to CPU timing, which also used the Chronos library for execution timing.

C++ Functions

The image restoration process implemented in the project is a three-stage process:

1. Create a point spread function (PSF) using summation value and normalization.
2. Create a Wiener filter based on the PSF function using various mathematical operations including DFT (Discrete Fourier Transform), Merge and Split of Filter.
3. Restore the blurred image using the wiener filter by means of several mathematical operations including DFT (Discrete Fourier Transform), IDFT (Inverse Discrete Fourier Transform), Mulspectrum, Merge and Split of image. Tables 2 and 3 highlights the main kernels and functions that were implemented in CUDA and C++ respectively.

Kernel	Return type	Description
getPSNR()	double	Calculate correctness of algorithm using MSE and PSNR. PSNR - Peak Signal to Noise Ratio MSE - Mean Squared Error
calcPSF()	Mat	Create a PSF function using summation value.
calcWnrFilter()	Mat	Create a wiener filter based on PSF Function
filter2DFreq()	Mat	Create the restored image using the wiener filter and input image.

Table 2. C++ Functions Implemented for image restoration

Sequential CPU Algorithm

The sequential algorithm of image restoration used the existing OpenCV implementation and manually handled the built-in functionalities. Each image underwent three stages of image restoration in a sequential manner by performing pixel-wise manipulations.

CUDA Kernels

Kernel	Return type	Description
calcPSF()	GpuMat	Calculate Summation in Point Spread Function (PSF) using AtomicAdd
psf_normalize()	GpuMat	Normalizes the Point Spread Function (PSF) using summation value
fft_shift()	GpuMat	Calculate FFT Shift of Point Spread Function
mergefilter()	GpuMat	Merge Image/Filter with zero values plane - Complex data type
splitfilter()	GpuMat	Split Image/Filter containing two planes to a single plane
pow_add_div_filter()	GpuMat	Adding and Division mathematical operations as part of restoration process
mulSpectrums()	GpuMat	Dot product of image and wiener filter
normalize_img()	GpuMat	Normalize final restored image - Fits into (0-255) range based on min and max

Table 3. CUDA Kernels Implemented for image restoration

GPU Algorithm - One thread per pixel

The CUDA GPU implementation handled each image sequentially. However, the CUDA kernels implemented were capable of handling one thread per pixel. For example, an image of size 256 * 256 consisted of 256 * 256 threads working parallelly. The size of grids and blocks were modified accordingly to fit all the pixels in the image. Since each thread was responsible for a single element, it was easy and possible to handle pixel-level mathematical computations including merging, split, summation, etc. In order to make sure no data is left out, __syncthreads() commands were used after a loop to ensure complete computation had been performed.

Data

All images used in this project are .png files. Images that are in grayscale are utilized, pixel data is represented between 0 and 255 integers. The OpenCV Mat and GPU Mat representations of the images contain float type data as opposed to integer type data. Though the original image pixel data is represented as integers, a floating point representation of the pixels is needed to contain the necessary information and degree of precision for the various processing steps.

Images are square and are of dimensions equal to a factor of 2. Images were gathered from OpenCV resources, image databases, and other online resources. Images were stored in the same folder as the implementation code. The initial unaltered image, the blurred version, and the restored version were all stored after the image restoration process was completed.

Experiment Setup

Five image sizes were experimented with to assess the performance of the image restoration implementations. The sizes tested were: 8 pixels by 8 pixels, 32 pixels by 32 pixels, 64 pixels by 64 pixels, 128 pixels by 128 pixels, and 256 pixels by 256 pixels.

For the GPU CUDA implementation, various execution configurations were tested for performance. A thread was assigned per pixel. Square grid configurations and square block configurations were tested. Execution configurations were constrained by the specifications of the GPU compute capability. More specifically for example, the block dimensions were constrained by the limitations of threads per block. As only 1024 threads are supported by a block, the max 2D dimensions used for execution configurations was 32 threads by 32 threads per block.

Setup, Compilation, and Execution Instructions

To implement the image restoration algorithm for CPU and GPU the following steps were followed.

Setup:

1. Install CUDA toolkit v11.2. on a machine with NVIDIA GeForce RTX 3070 GA104 GPU.
2. Install OpenCV v3.2.0.
3. Install NVIDIA Nsight Profiler in Visual Studio Code v1.65.2
4. Transfer CPU and GPU implementation files to specified machines and store them in a specific directory.
5. Store the required input images in .png format to the machine.

Compilation:

1. For CPU and GPU implementations, specify the name of the image file on which the image restoration will be performed.
2. Specify the path where the blurred image would be stored after execution.
3. Specify the path where the restored image would be stored after execution.
4. For GPU implementation, specify the appropriate block size for the image size being tested.
5. To compile CPU implementation:

```
g++ program_name.cpp -o program -I/usr/local/include/opencv4 -lopencv_core
-lopencv_highgui -lopencv_imgcodecs -lopencv_imgproc  $(pkg-config opencv4 --libs)
```

6. To compile GPU implementation:

```
nvcc -std=c++11 program_name..cu -o program -I/usr/local/include/opencv4
-lopencv_core  -lopencv_highgui -lopencv_imgcodecs -lopencv_imgproc  $(pkg-config
opencv4 --libs)
```

Execution:

1. To execute CPU/GPU implementation:

```
./program
```

2. To execute GPU implementation with profiler:

```
nv-nsight-cu-cli ./program      (or)
nsys profile --stats=true --force-overwrite true --show-output true ./program
```

Gaussian Blur

A clear image is taken, and gaussian blur is introduced as a first stage to demonstrate the image restoration process. This is to ensure that we have the scope for baseline comparison to the original image, which would help estimate the difference in pixels in the form of absolute values. The image with noise undergoes different stages throughout the process of restoration. The processing pipeline and steps are discussed below.

Point Spread Function (PSF)

The PSF function indicates the 2D distribution of light signal for a spatial domain in the form of a dot. This dot can be spread across a blurred area in an image to vary the light intensity. When the point spread function is obtained in the form of impulse response in the optical domain, this is then converted to the frequency domain that can be used to produce changes to the frequency domain of the distorted image. One crucial step in deciding the point spread function is to gather information about the signal-to-noise ratio. This will help estimate the radius of the point spread function. This varies based on the wavelength of light at hand and viewing. For instance, shorter wavelengths of light in regions of the image result in a smaller value of the point spread function. For the current use case, the point spread function is produced with radius (R). The overall size of the kernel is the same as the size of the image. The image of the PSF having a smaller radius is shown in Figure 1.

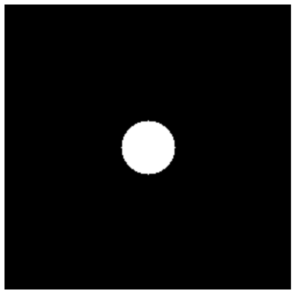


Figure 1. Circular Point Spread Function (PSF)

When implementing the code for PSF, atomicAdd operation of CUDA was used to sum the brighter pixels. To effectively handle summation operation, the variable maintaining the count was passed in a native Mat format for accommodation. After the summation, the results were normalized to get floating-point precision values for the white circular portion of the above-shown kernel.

Frequency Domain

To better manipulate pixels of the image in the discrete format, a continuous representation with the sine and cosine containing real and imaginary values is essential. This representation can be obtained by using a discrete Fourier transform to create a continuous frequency domain for image representation. When the kernel image is created, it is juxtaposed with the distorted image in the frequency domains to obtain a combined dot product. This essentially creates a single representation that has both frequency and phase information. Figure 2 shows the overlap process of the intermediate product linked to the kernel and the restored image.

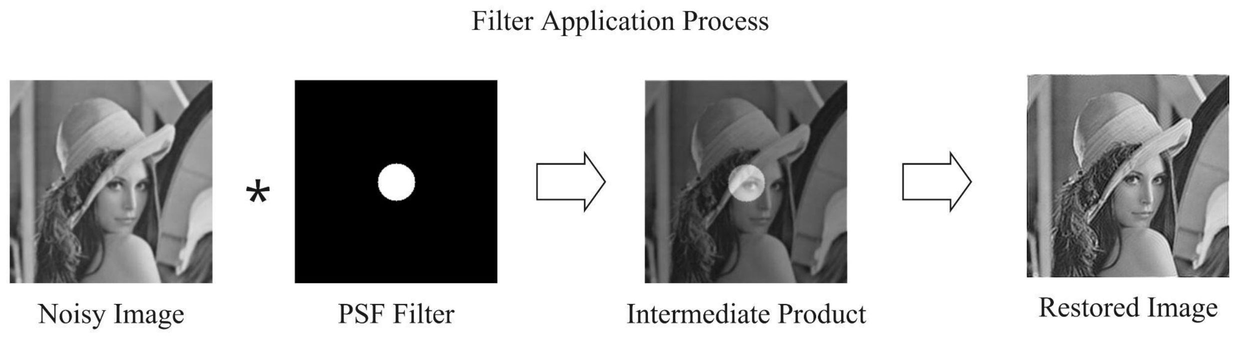


Figure 2. Process Pipeline for blending the Kernel with the blurry/noisy Image

This contiguous representation in the frequency domain-containing real and imaginary parts is converted back to the discrete pixel-wise representation as in Figure 3.

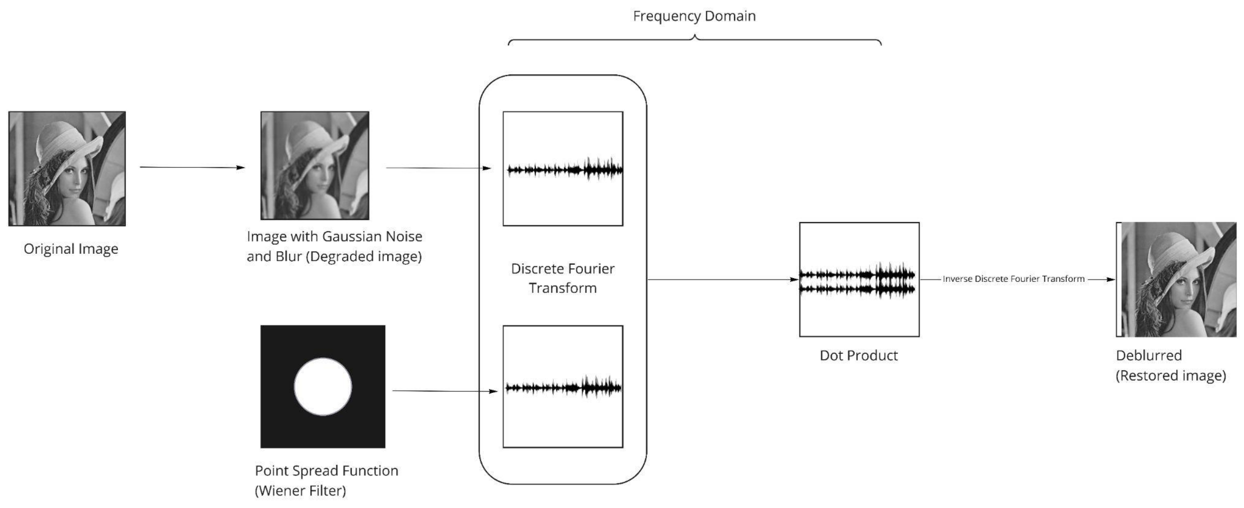

Frequency Domain
Original Image
Image with Gaussian Noise and Blur (Degraded image)
Discrete Fourier Transform
Point Spread Function (Wiener Filter)
Dot Product
Inverse Discrete Fourier Transform
Deblurred (Restored image)

Figure 3. Process Pipeline showcasing restoration with accommodated frequency domain

Performance Measurements

As this image manipulation application deals with pixel data in integer format, the Floating Point Operations Per Second (FLOP/s) calculation was not appropriate to assess implementation performance. Instead, performance was reported in execution time calculated in seconds. Execution time for the CPU implementation includes timing for all image processing methods, both manually implemented and from OpenCV. Execution time for the GPU implementation has only timing for the image processing manually implemented kernels and OpenCV kernels. The GPU implementation timing did not include data movement from host to device and back as this would not be comparable to the CPU, so only exclusive execution timing for the GPU is compared to the CPU.

Accuracy Measurements

In order to check the correctness of the implemented algorithm, the Peak Signal to Noise Ratio (PSNR), and Mean Squared Error (MSE) difference between input and restored images are calculated. PSNR is “the ratio between the maximum possible power of a signal and the power of corrupting noise that affects the fidelity of its representation” (*Peak signal-to-noise ratio* 2021). PSNR is widely used in image processing and image restoration to assess the quality of images. MSE is generally used as a measure of the quality of an estimator, for this application this refers to the filter parameters. The PSNR and MSE differences are taken for both the CPU and GPU implementation to ensure that the values are the same and that residuals are minimal. The intent is to ensure that the precision of correctness is not compromised for faster GPU implementation.

V. RESULTS

Performance Analysis

A performance comparison between CPU and GPU is indicated for different combinations of image size, grid size, and block size. The execution time of the algorithms is computed with floating-point precision and units of seconds.

For an image size of 8x8, the GPU performance is best when the combination of grid size is small. This is indicative of the prevented latency by conserving the grid pattern for smaller size and a higher allocation of blocks per grid. Table 4 indicates the performance for different combinations of grid and block sizes.

Image Size	**Grid Size**	**Block Size**	**CPU Manual Performance (secs)**	**GPU Performance (secs)**
8X8	8x8	1x1	0.000079	0.000078
8X8	2x2	4x4	0.000087	0.000076
8X8	4x4	2x2	0.000079	0.000077

Table 4. Grid and Block combinations for an 8x8 Grayscale Image

For an image size of 32x32, it is indicative that the performance time variation is increasing with the growth of the image size. Additionally, there is a substantial difference between the GPU and CPU performance by a magnitude of 10. The following table 5 indicates the performance for different combinations of grid and block sizes.

Image Size	**Grid Dim**	**Block Dim**	**CPU Manual Performance (secs)**	**GPU Performance (secs)**
32X32	4x4	8x8	0.000204	0.000090
32X32	8x8	4x4	0.000202	0.000088
32X32	2x2	16x16	0.000203	0.000089

Table 5. Grid and Block combinations with performance time for a 32x32 Greyscale Image

Image Dim	**Grid Dim**	**Block Dim**	**CPU Manual Performance (secs)**	**GPU Performance (secs)**
64X64	4x4	16x16	0.000622	0.000135
64X64	8x8	8x8	0.000607	0.000131
64X64	16x16	4x4	0.000609	0.000130

Table 6. Grid and Block combinations with performance time for a 64x64 Grayscale Image

Image Dim	**Grid Dim**	**Block Dim**	**CPU Manual Performance (secs)**	**GPU Performance (secs)**
128X128	4x4	32x32	0.001777	0.000258
128X128	8x8	16x16	0.001195	0.000247
128X128	32x32	4x4	0.001195	0.000257

Table 7. Grid and Block combinations with performance time for a 128x128 Grayscale Image

The matrix size of 256x256 has a substantial change. This is indicative that with the growing size of the input vectors, the algorithms showcase the considerable difference in the implementation of CPU to GPU, respectively. In the case of CPU, the manual execution time is a magnitude of 100 less than the time consumed by the GPU. Another factor to notice is better performance allocation for lower dimensions with larger block sizes. This pattern is the same even in the case of smaller images. This observation shows that the latency is least when the grid is split with the least combination and the largest block to accommodate an image.

Configuration	**Image Dim**	**Grid Dim**	**Block Dim**	**Performance (secs)**		
				CPU OpenCV	**CPU Manual**	**GPU**
1	256X256	8x8	32x32	0.002163	0.006569	0.000819
2	256X256	16x16	16x16	0.002163	0.006644	0.000821
3	256X256	32x32	8x8	0.002163	0.006622	0.000854

Table 8. Grid and Block combinations with performance time for a 256x256 Grayscale Image

Table 9 shows the Peak Signal to Noise Ratio (PSNR), and Mean Squared Error (MSE) difference between the original and restored images. It is evident from the values obtained that the correctness for CPU and GPU implementation is the same. This is indicative that there is no loss in the accuracy of the image with accelerated GPU implementation.

Image Dimension	**CPU MSE**	**CPU PSNR**	**GPU MSE**	**GPU PSNR**
8x8 (lena_8.png)	3179.125000	13.1077	3179.125000	13.107728
32x32 (lena_32.png)	493.272461	21.1999	493.272461	21.199935
64x64 (lena_64.png)	300.813232	23.3478	300.813232	23.347834
256X256 (bear_256.png)	81.768997	29.0049	81.768997	29.004917

256X256 (flower_256.png	163.291367	26.001171	163.291367	26.0012

Table 9. Correctness measure for CPU and GPU execution

The images in figure 4.0 are shown in sequences at different stages of the algorithm to output the overall image processing incorporated for restoration.

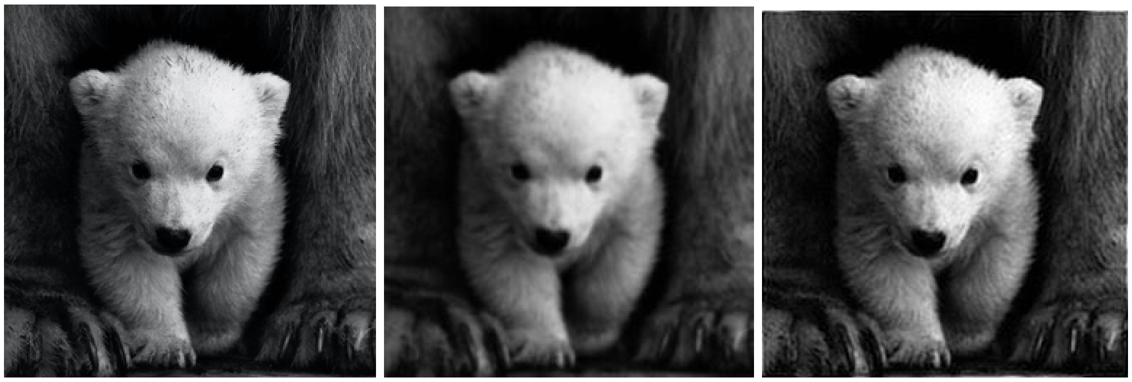
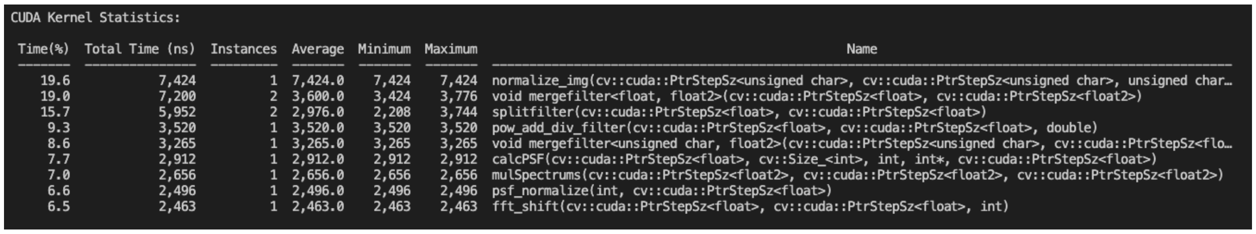


Figure 4. Original Image(left), Image with Gaussian blur(center), Restored Image(right)

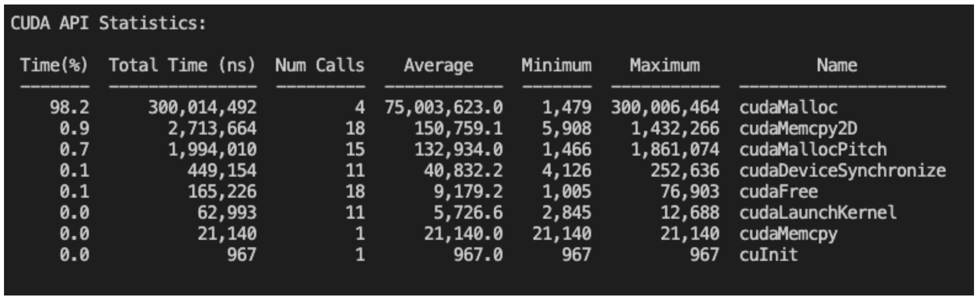
Profiler Results

NVIDIA profiling tools were used to understand and optimize the performance of your CUDA application. Figure 5 shows the execution times taken for every CUDA kernel implemented and the time taken by each CUDA API. The kernels such as normalize, merge and split filter took up around 45% of the time due to its repeated number of instances in image restoration. Also, it is interesting to note that around 98% of the total time was spent on cudaMalloc process.



```
CUDA Kernel Statistics:

 Time(%)  Total Time (ns)  Instances  Average  Minimum  Maximum                                                   Name
 -------  ---------------  ---------  -------  -------  -------  ----------------------------------------------------------------------------------------------------
    19.6            7,424          1  7,424.0    7,424    7,424  normalize_img(cv::cuda::PtrStepSz<unsigned char>, cv::cuda::PtrStepSz<unsigned char>, unsigned char…
    19.0            7,200          2  3,600.0    3,424    3,776  void mergefilter<float, float2>(cv::cuda::PtrStepSz<float>, cv::cuda::PtrStepSz<float2>)
    15.7            5,952          2  2,976.0    2,208    3,744  splitfilter(cv::cuda::PtrStepSz<float>, cv::cuda::PtrStepSz<float>)
     9.3            3,520          1  3,520.0    3,520    3,520  pow_add_div_filter(cv::cuda::PtrStepSz<float>, cv::cuda::PtrStepSz<float>, double)
     8.6            3,265          1  3,265.0    3,265    3,265  void mergefilter<unsigned char, float2>(cv::cuda::PtrStepSz<unsigned char>, cv::cuda::PtrStepSz<flo…
     7.7            2,912          1  2,912.0    2,912    2,912  calcPSF(cv::cuda::PtrStepSz<float>, cv::Size_<int>, int, int*, cv::cuda::PtrStepSz<float>)
     7.0            2,656          1  2,656.0    2,656    2,656  mulSpectrums(cv::cuda::PtrStepSz<float2>, cv::cuda::PtrStepSz<float2>, cv::cuda::PtrStepSz<float2>)
     6.6            2,496          1  2,496.0    2,496    2,496  psf_normalize(int, cv::cuda::PtrStepSz<float>)
     6.5            2,463          1  2,463.0    2,463    2,463  fft_shift(cv::cuda::PtrStepSz<float>, cv::cuda::PtrStepSz<float>, int)
```

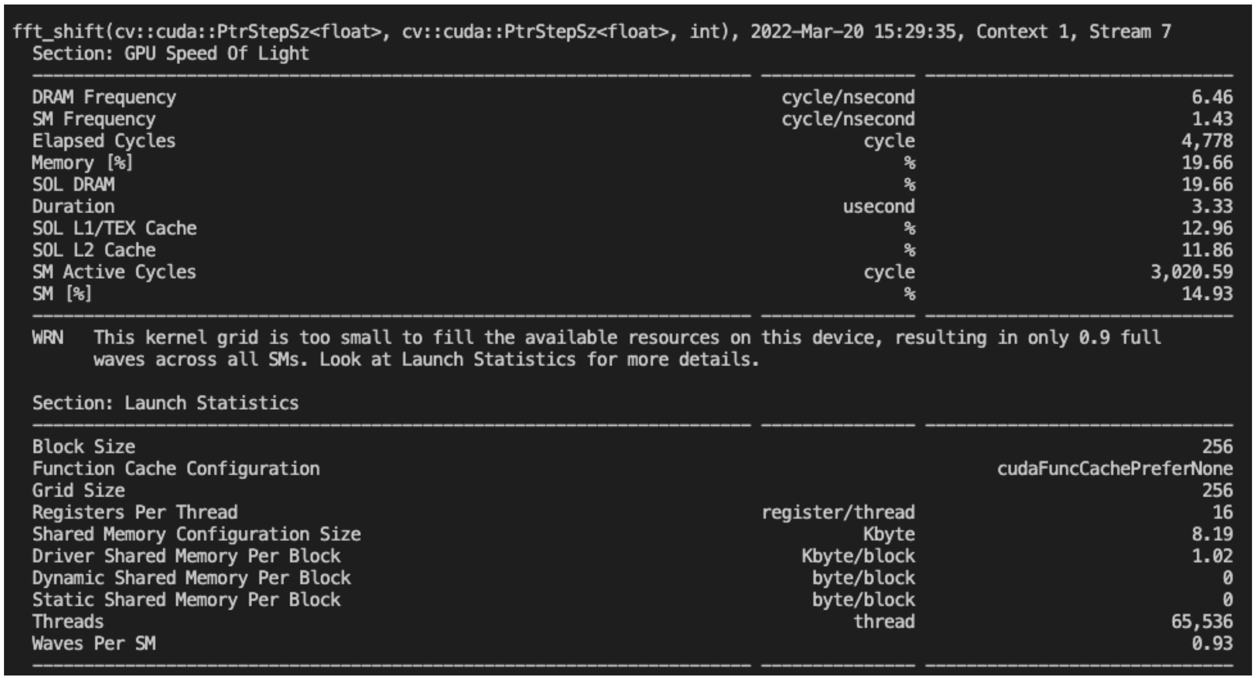


```
CUDA API Statistics:

 Time(%)  Total Time (ns)  Num Calls    Average     Minimum    Maximum           Name
 -------  ---------------  ---------  ------------  -------  -----------  ---------------------
    98.2      300,014,492          4  75,003,623.0    1,479  300,006,464  cudaMalloc
     0.9        2,713,664         18     150,759.1    5,908    1,432,266  cudaMemcpy2D
     0.7        1,994,010         15     132,934.0    1,466    1,861,074  cudaMallocPitch
     0.1          449,154         11      40,832.2    4,126      252,636  cudaDeviceSynchronize
     0.1          165,226         18       9,179.2    1,005       76,903  cudaFree
     0.0           62,993         11       5,726.6    2,845       12,688  cudaLaunchKernel
     0.0           21,140          1      21,140.0   21,140       21,140  cudaMemcpy
     0.0              967          1         967.0      967          967  cuInit
```

Figure 5. CUDA Kernel and API statistics showing execution times taken

Figure 6 shows the CUDA memory usage for a specific kernel implementation - fft_shift. Similarly memory usage of individual kernels can be profiled to explore and understand the usage of registers, cache, and shared memory used.



```
fft_shift(cv::cuda::PtrStepSz<float>, cv::cuda::PtrStepSz<float>, int), 2022-Mar-20 15:29:35, Context 1, Stream 7
  Section: GPU Speed Of Light
  ---------------------------------------------------------------------- --------------- ------------------------------
  DRAM Frequency                                                           cycle/nsecond                           6.46
  SM Frequency                                                             cycle/nsecond                           1.43
  Elapsed Cycles                                                                   cycle                          4,778
  Memory [%]                                                                           %                          19.66
  SOL DRAM                                                                             %                          19.66
  Duration                                                                       usecond                           3.33
  SOL L1/TEX Cache                                                                     %                          12.96
  SOL L2 Cache                                                                         %                          11.86
  SM Active Cycles                                                                 cycle                       3,020.59
  SM [%]                                                                               %                          14.93
  ---------------------------------------------------------------------- --------------- ------------------------------
  WRN   This kernel grid is too small to fill the available resources on this device, resulting in only 0.9 full
        waves across all SMs. Look at Launch Statistics for more details.

  Section: Launch Statistics
  ---------------------------------------------------------------------- --------------- ------------------------------
  Block Size                                                                                                        256
  Function Cache Configuration                                                                   cudaFuncCachePreferNone
  Grid Size                                                                                                         256
  Registers Per Thread                                                   register/thread                             16
  Shared Memory Configuration Size                                                 Kbyte                           8.19
  Driver Shared Memory Per Block                                             Kbyte/block                           1.02
  Dynamic Shared Memory Per Block                                             byte/block                              0
  Static Shared Memory Per Block                                              byte/block                              0
  Threads                                                                         thread                         65,536
  Waves Per SM                                                                                                     0.93
  ---------------------------------------------------------------------- --------------- ------------------------------
```

Figure 6. CUDA Memory Usage for a kernel implemented in CUDA - fft_shift

VI. CONCLUSION

Results of experimentation show that the CUDA implementation of the image restoration algorithm outperforms the C++ implementation as the problem size grows i.e., for larger images. Results also show that this acceleration does not come at the cost of accuracy. The MSE and PSNR calculations are the same, with minimal residuals, for both implementations.

These results were only possible through the team's knowledge and experience from this high-performance computing course. The course clarified our fundamentals about CUDA and optimization strategies that can be done with manipulating the executive configuration. The scope of memory and its usage in terms of global and local to better optimize occupancy was an outcome of exercises and assignments discussed in the class.

VII. FUTURE WORK

To further optimize this algorithm, a tiling and loop unroll approach can be incorporated inside each kernel. The proposed design and algorithm can be extended to real-time image processing applications incorporated on edge devices. In addition, as the restoration acceleration is achieved, there is a scope for applying the same to video-based platforms where the video streams are dealt with frames.

VIII. REFERENCES

Villa, O., Lustig, D., Yan, Z., Bolotin, E., Fu, Y., Chatterjee, N., ... & Nellans, D. (2021, February). Need for Speed: Experiences Building a Trustworthy System-Level GPU Simulator. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)* (pp. 868-880). IEEE.

Dougherty, G., & Kawaf, Z. (2001). The point spread function revisited: image restoration using 2-D deconvolution. In Radiography (Vol. 7, Issue 4, pp. 255–262). *Elsevier BV*. https://doi.org/10.1053/radi.2001.0341

Monika Maru, M. C. Parikh, Image Restoration Techniques: A Survey, *International Journal of Computer Applications* (0975 – 8887) Volume 160 – No 6, February 2017

Chang, HH., Li, CY. An automatic restoration framework based on GPU-accelerated collateral filtering in brain MR images, *BMC Med Imaging* 19, 8 (2019)

Bo Hu, Leida Li, Jinjian Wu, Jiansheng Qian, Subjective and objective quality assessment for image restoration: A critical survey, *Signal Processing: Image Communication*, Volume 85, 2020, 115839, ISSN 0923-5965,

OpenCV Out-of-focus Deblur Filter. Retrieved March 4, 2022, from https://docs.opencv.org/3.4/de/d3c/tutorial_out_of_focus_deblur_filter.html

Marwa, E., Heshan, L., and Wu-chun, F. Performance Characterization and Optimization of Atomic Operations on AMD GPUs, Department of Computer Science, Virginia Tech, 2011

Marwan-abdellah, cufftShift CUDA-based implementation for linear 1D, 2D and 3D FFT-Shift funtions. Retrieved March 18, 2022, from https://github.com/marwan-abdellah/cufftShift

Opencv image / video similarity measurement, PSNR (peak signal to noise ratio) and SSIM, video / image conversion. Retrieved March 18, 2022, https://chowdera.com/2021/02/20210206131113204S.html

Wen, Y. W., Ng, M. K., & Ching, W. K. (2008). Iterative algorithms based on decoupling of deblurring and denoising for image restoration. *SIAM Journal on Scientific Computing*, *30*(5), 2655-2674.

Banham, M. R., & Katsaggelos, A. K. (1997). Digital image restoration. *IEEE signal processing magazine*, *14*(2), 24-41.

Figueiredo, M. A., & Nowak, R. D. (2003). An EM algorithm for wavelet-based image restoration. *IEEE Transactions on Image Processing*, *12*(8), 906-916.

Savakis, A. E., & Trussell, H. J. (1993). On the accuracy of PSF representation in image restoration. *IEEE transactions on image processing*, *2*(2), 252-259.

Wikimedia Foundation. (2022, March 16). *Flops*. Wikipedia. Retrieved March 18, 2022, from https://en.wikipedia.org/wiki/FLOPS

Wikimedia Foundation. (2021, December 26). *Peak signal-to-noise ratio*. Wikipedia. Retrieved March 19, 2022, from https://en.wikipedia.org/wiki/Peak_signal-to-noise_ratio