HIGH PERFORMANCE PLATFORMS FOR BEAM PROJECTION AND ADAPTIVE IMAGING APPLICATIONS

by

Furkan Cayci

A dissertation submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Electrical & Computer Engineering

Summer 2016

© 2016 Furkan Cayci
All Rights Reserved

HIGH PERFORMANCE PLATFORMS FOR BEAM PROJECTION AND ADAPTIVE IMAGING APPLICATIONS

by

Furkan Cayci

Approved: _____

Kenneth E. Barner, Ph.D.
Chair of the Department of Electrical and Computer Engineering

Approved: _____

Babatunde A. Ogunnaike, Ph.D.
Dean of the College of Engineering

Approved: _____

Ann L. Ardis, Ph.D.
Senior Vice Provost for Graduate and Professional Education

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Fouad Kiamilev, Ph.D.
Professor in charge of dissertation

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Chase Cotton, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Charles Boncelet, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Willett Kempton, Ph.D.
Member of dissertation committee

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude for working with such an amazing, humble and supportive person, my advisor Fouad Kiamilev. He has shaped the way I perceive the world and given me new perspectives. His positive energy helped me get through the difficult challenges I have faced, and his constant support encouraged me to take up more challenges.

I would like to thank my committee members for dedicating their time and giving me feedback on this work. They have always supported me throughout this journey and I am grateful for this.

CVORG will always stay as a special place in my heart and I feel sad to say goodbye to these amazing people. I will miss you all.

I would like to thank my family for believing in me and supporting me. Their unfailing love have been a great source of energy for me even though they are far away. I will never forget the hardships that they have faced to provide me a better future. I am and always will be forever in their depth.

Finally I dedicate my work to my best friend and my wife Hatice Sinem. I am truly blessed to have you in my life and grateful for your friendship, support and love.

This project is funded by the US ARMY RDECOM under Contract No. W911NF-11-2-088 and W911QX-15-C-0041. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of US Army.

TABLE OF CONTENTS

LIST OF TABLES . . . . . **viii**
LIST OF FIGURES . . . . . **ix**
ABSTRACT . . . . . **xii**

Chapter

1 INTRODUCTION **1**

1.1 Introduction 1
1.2 Related Work in Imaging Through Turbulence Techniques 4
1.3 Related Work in Laser Beam Projection Applications 7
1.4 Thesis Overview and Outline 9

2 REAL-TIME IMAGE PROCESSING PLATFORM FOR ATMOSPHERIC TURBULENCE MITIGATION **12**

2.1 Introduction 12
2.2 Atmospheric Turbulence Effects In Imaging 15
2.3 Performance Needs 16

2.3.1 Convolution Operation 17

2.4 Platform Design 20

2.4.1 The hardware 21
2.4.2 The processing framework 22
2.4.3 Platform performance 26

2.4.3.1 Convolution operation runtime 26
2.4.3.2 Latency and throughput 27

2.4.3.3 Kernel launch setup time . . . 28

2.5 Lucky-Region Fusion Implementation . . . 29

2.5.1 Overview of the algorithm . . . 32
2.5.2 LRF implementations using existing tools . . . 33
2.5.3 Real-time implementation . . . 35

2.5.3.1 Modified algorithm . . . 36

2.6 Results . . . 37
2.7 Conclusions . . . 40

3 FIBER LASER PHASED-ARRAY CONTROLLER PLATFORM 43

3.1 Introduction . . . 43
3.2 Controller Platform Design . . . 46

3.2.1 Part I - Analysis and simulation framework . . . 46

3.2.1.1 Framework construction and layers . . . 47
3.2.1.2 Framework operation . . . 50

3.2.2 Part II - Hardware engine . . . 55

3.2.2.1 Processing back end . . . 55
3.2.2.2 Scatter interface . . . 57
3.2.2.3 Gather interface . . . 63
3.2.2.4 Final hardware engine . . . 64

3.3 Experiments And Results . . . 65

3.3.1 Stochastic Parallel Gradient Descent Method . . . 66
3.3.2 Simulations . . . 68

3.3.2.1 Monte Carlo parameter sweeps . . . 70
3.3.2.2 Transient analysis . . . 74

3.3.2.3 Convergence analysis 81

3.3.3 Hardware experiments 85

3.3.3.1 Hardware Engine performance with SPGD method . 85

3.3.4 Capacitive load test of the amplifiers 88
3.3.5 19-channel electrical loopback operation 90
3.3.6 7-channel optical loopback operation 92

3.4 Conclusions 96

4 SUMMARY AND FUTURE WORK 99

4.1 Summary 99
4.2 Future Work 100

BIBLIOGRAPHY 101

LIST OF TABLES

2.1 Frame rate speed 17

2.2 2D image convolution operation pseudo-code 19

2.3 Convolution timing using OpenMP on CPU 20

2.4 Convolution timing 26

2.5 Memory throughput 28

2.6 LRF implementation on Python 34

2.7 LRF implementation on OpenCV 34

2.8 Real-time LRF algorithm running on the Framework 38

LIST OF FIGURES

1.1 Adding two same frequency signals with various phase differences . 3

2.1 Atmospheric turbulence effects in imaging 13

2.2 Convolution operation 18

2.3 Hardware connection diagram 23

2.4 Framework layers 25

2.5 GPU kernel call overhead 29

2.6 Probability of getting a good short-exposure ‘lucky‘ image 31

2.7 LRF execution time comparison 39

2.8 Total speedup 40

2.9 LRF resuls on water tower 41

2.10 LRF results on lab setup 42

3.1 A typical fiber laser phased-array 45

3.2 Abstraction Layers 49

3.3 Transient analysis 51

3.4 Monte Carlo simulations 53

3.5 Algorithm development flow 54

3.6 Processing Back end connection diagram 58

3.7 Amplifier circuit model 60

3.8 Step response, rise and fall times of the circuit model 61
3.9 Eye diagrams 62
3.10 Final Hardware Engine connection diagram 64
3.11 Final Hardware Engine construction 65
3.12 SPGD black box optimization 67
3.13 SPGD algorithm operation 69
3.14 MC simulations - no distortion 71
3.15 MC simulations - sinusoidal phase noise 72
3.16 MC simulations - sinusoidal phase noise 73
3.17 SPGD transient run 1 - no phase-locking 75
3.18 SPGD transient run 2 - no phase-locking 76
3.19 MC simulations - step noise 77
3.20 MC simulations - sinusoidal phase noise 78
3.21 MC simulations - random phase noise 79
3.22 MC simulations - random phase noise 80
3.23 Convergence - no noise 81
3.24 Convergence - step noise 82
3.25 Convergence - sinusoidal noise 83
3.26 Convergence - random noise 84
3.27 Rise & Fall times of the the amplifiers 89
3.28 19-channel loopback module 90
3.29 SPGD operation on 19-channel electrical setup 91

3.30 7-channel optical loopback setup 92

3.31 7-channel experimental run 1 93

3.32 7-channel experimental run 2 94

3.33 7-channel experimental run 3 95

3.34 7-channel experimental run 4 96

3.35 Final Controller box 97

ABSTRACT

Mitigating atmospheric turbulence effects in long-range real-time applications such as imaging and laser beam projections requires efficient algorithms coupled with high performance platforms. Such long distance applications include remote power delivery, free-space optical communications, remote reconnaissance, surveillance, target identification, target tracking, and biometrics. Therefore, these algorithms need to be optimized and tuned for the specific application needs for dynamically changing environments. Current off-the-shelf systems are limited in performance, control channels, and flexibility. In this dissertation, to enhance this algorithm development and optimization process and to meet the performance needs of these applications, two separate platforms coupled with high-speed electronics and novel frameworks have been developed.

A high performance imaging platform is developed for designing and implementing atmospheric turbulence mitigation algorithms and testing them in real-time on-field experiments. A real-time implementation of the Lucky-region fusion (LRF) technique, a synthetic imaging technique in enhancing the quality of images distorted by atmospheric turbulence using the platform is demonstrated with high frame rates over 100 FPS.

An innovative, scalable and versatile platform is developed for closed-loop controlling of fiber laser phased-arrays. The controller platform consists of two complementary parts. The first part is an analysis and simulation software framework that is used for rapid algorithm development for blind optimization. The second part is a real-time hardware engine that is capable of controlling a large number of array elements and run the optimized algorithms at high speeds for field testing. The controller platform is then tested by utilizing a multi-channel optimization algorithm called

Stochastic Parallel Gradient Descent (SPGD) for phase-locking. Simulation and hardware based experimental results are presented, and significant controller speed up to 650 kHz multi-channel update rate is achieved.

Chapter 1

INTRODUCTION

1.1 Introduction

The area that surrounds Earth is called the atmosphere or the air and it consists of nitrogen, oxygen, argon, carbon-dioxide and many other gases at small percentages. The air is essential for sustaining life while absorbing most of the harmful ultraviolet lights that come from the sun, retaining the surface heat, and preventing excessive heat differences throughout the day. Together these gases form a medium that acts as a waveguide to absorb or emit radiation.

The propagation of light in a medium is expressed with refractive index. A light beam traveling through the Earth's atmosphere experiences random fluctuations in its intensity and phase due to the refractive index changes in the air. These refractive index changes are generally caused by eddies in the atmosphere that happen due to temperature and pressure variations between different locations and they pose a big challenge for many long distance applications that require light beams traveling through air.

An example to these applications would simply be long range imaging applications that can be used in astronomy, space observation and exploration, target tracking, drone flight, driverless cars and sports. In astronomy large telescopes are required to observe the stars, planets and galaxies. When ground-based telescopes are used, light still needs to travel through Earth's atmosphere. Thus, captured images often end up blurry and the features of the target are usually not identifiable. A number of telescopes, such as Hubble Space Telescope, are deployed to space to overcome this problem and bypass Earth's atmosphere. Although this approach bypasses the turbulence problem and provides significant contributions to space exploration studies, it is very costly to launch such a telescope to space and keep maintaining it in the orbit. Additionally, many other imaging applications require that the target be located on Earth or in the atmosphere which again makes these kind of space telescopes ineffectual. An alternative ground-based approach to improve image quality is to use *adaptive optics* [34] by utilizing special electronics and optics. In these studies, the incoming wavefront is measured and corrected by using deformable mirrors in real time to compensate for the dynamic changes in the turbulence. This approach is also expensive and requires both carefully constructed optics and specially designed electronics to compensate for the changes in the turbulence. Finally, pure software methods exist to mitigate the turbulence effects using image processing techniques, but these methods are often post-imaging processes, complex, platform specific and inefficient.

Another set of applications that need optical beam propagation through air are the ones that depend on projecting a laser beam to a far away target. Such applications may include free-space optical communications, remote power delivery to vehicles and

remote charging of air, ground and underwater sensors. These applications rely on the laser beam reaching the target with minimal distortions to achieve efficient power delivery. However, as the target moves further away, the efficiency drops significantly. This problem gave rise to alternative approaches by using an array of lasers to compensate for the inefficient power transfers also known as *beam combining* [16].

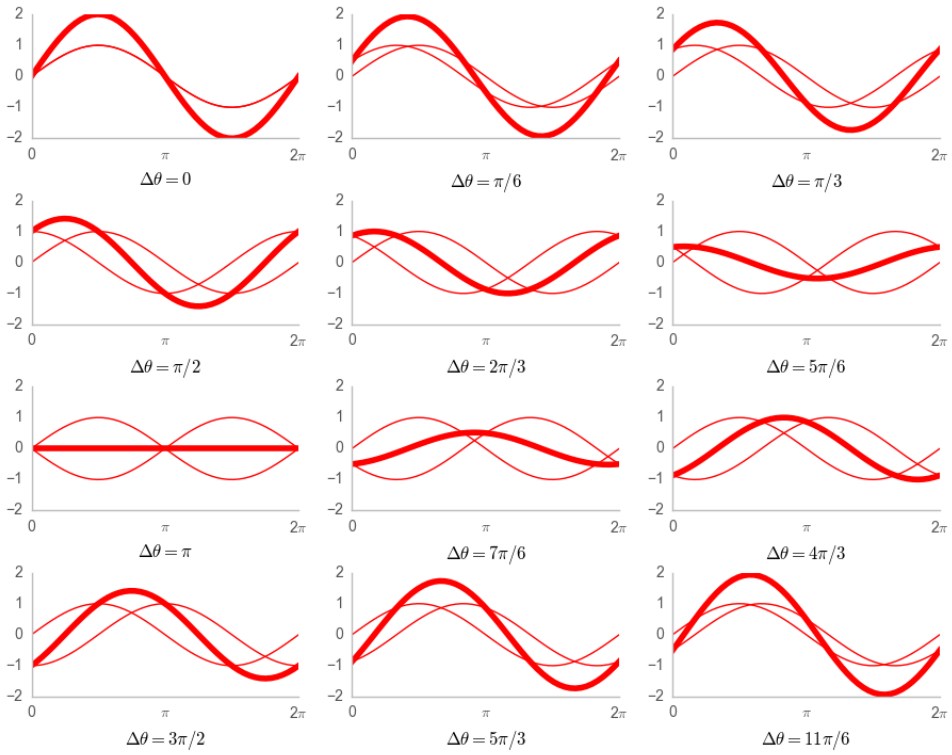


Figure 1.1: Adding two same frequency signals with various phase differences

One of the most promising beam combining techniques is called coherent beam

combining (CBC) [16] which depends on all the array elements being in the same phase to have a constructive addition at the target to increase the power. However, as the laser beams go through atmospheric turbulence, they generally experience random fluctuations in their intensity and phase values. This causes each beam to have a different phase at the target which ends up lowering beam intensity as shown in Figure 1.1.

Although coherent beam combining has been demonstrated using a small number of array elements, combining a large number of array elements requires a dedicated hardware system, careful architectural decisions, high speed electronics and robust algorithms.

1.2 Related Work in Imaging Through Turbulence Techniques

In literature, many techniques exist for mitigation turbulence effects. Some of these techniques depend on special adaptive optics and hardware systems to control these optical elements [3, 40]. These type of systems work by wavefront sensing using special wavefront detectors and controlling specially constructed, many-element mirrors called *deformable mirrors* to negate these sensed wavefront effects of the turbulence by generating an opposite waveform. This method has been highly used in astronomy applications where ground-based observation is involved.

The requirement of having these expensive optical elements such as deformable mirrors and wavefront sensors led researchers to look for alternative methods that mainly involve passive observation of the turbulence induced target using telescopes or

cameras and running software algorithms to mitigate the atmospheric turbulence effects. These methods are usually implemented as a post-processing solution to capture the still target and enhance the quality.

One of these methods used for turbulence mitigation is called *lucky imaging* [20]. The idea of lucky imaging depends on recording images at high frame rates with short-exposure times. Since the turbulence becomes mostly stationary in smaller time frames, capturing at high frame rates increases the chances of getting a good image.

There has been several applications of the lucky imaging approach using different tools or different frame selection methods. Law et al. [27] demonstrates their lucky imaging system, LuckyCam, attached to a 2.5 m telescope which is built for imaging stars. Their study shows that they are able to achieve near diffraction-limited images with a telescope in normal turbulence conditions. Although, their system provides good results against looking at stars, it is not practical to use in other applications due to the size and frame rate limitations. Oscoz et al. [33] built a faster imaging system to observe stars by using ground-based telescopes. They use an FPGA for recording and processing the images, and it is faster than the previous published systems. It's size again makes it not practical for various applications. Smith et al. [37] presents a case study in image quality variances in lucky imaging with different parameters such as wavelength, exposure time, telescope aperture and frame selection rate. By increasing the range of the parameters that are used in lucky imaging they are trying to help with future parameter optimizations. Another variance of lucky imaging is called Lucky-Region Fusion (LRF) approach presented by Aubailly et al. [4]. In this approach, the images are selected based on their sharp regions instead of as a whole.

These sharp regions are then fused into a synthetic image of the scene which yields a better quality latent image at the end. A more detailed explanation and experiments are presented in Chapter 2.

In addition to the lucky imaging methods, there are several different approaches to solve the atmospheric turbulence problem. Frakes et al. [19] uses an adaptive control grid interpolation approach by generating a motion vector field from the turbulence characteristics and feeding these into a turbulence compensation algorithm. It also aims to achieve real motion separation by applying a median filter and thresholding. Fishbain et al. [18] demonstrates turbulence compensation while keeping the moving objects intact. They start with generation of a reference frame, then they calculate the displacements of objects in relation to the reference frame. With this step, they manage to separate the moving objects from the turbulence and finally they apply turbulence compensation by minimizing the energy function in the optical flow. They sometimes omit using the optical flow due to computational intensity, and by doing so they state that their algorithm runs at 25 frames per second with 704 x 576 resolution. Lou et al. [5] uses a Sobolev gradient and Laplacian method to stabilize the video frames, then they apply a form of lucky-region method to fuse the images together to get a sharper image. Mao et al. [30] proposes a non rigid geometric distortions correction method by using optical flows to estimate the geometric distortions. Then they apply Bregman iteration for the optimization process. Although their results are comparable to the lucky-region fusion method, they state that fewer number of frames are needed for a sharp image. Micheli et al. [31] proposes a statistical dynamic model for the turbulence using a linear dynamic system, apply Kalman filtering to estimate the states and solve

the image recovery problem, and use nonlocal total variation deconvolution to sharpen the final image. Again this method takes a long time for a practical use case in real-time imaging scenarios.

1.3 Related Work in Laser Beam Projection Applications

In laser beam projection applications, usually a single laser with a big aperture is used. However, due to the power demands and temperature fluctuation problems alternative ways are sought such as using laser arrays by tiling smaller-size apertures together, and combining these multiple beams together at the target. Multiple techniques exist for beam combining. T. Y. Fan [16] characterizes these beam combining techniques into three broad classes according to the technique that the combination is performed: (I) Side-by-side beam combining by not actively controlling the phases of the beams that is used in conventional diode-laser arrays which produce lower beam quality (II) Wavelength or spectral beam combining (WBC) where the array elements run at different wavelengths and the beams get added together to generate a high power output. (III) Coherent beam combining (CBC) where the phases of each array element can be controlled individually to form a unified beam at the target by aligning all the phases together.

Considering different combination techniques as T. Y. Fan listed, there has been multiple coherent beam combining demonstrations in the literature. Some of these experiments only focus on keeping beam quality in a laboratory setup where the beams do not undergo any dynamically changing disturbances. Shay et al. [35] demonstrates phase locking of an optical array with 6 and 9 elements. Liu et al. [28]

demonstrates phase locking of a 3 element array using VLSI-based multi-dithering and stochastic parallel gradient descent techniques. These systems, while successful, utilizes a relatively undeveloped version of the algorithm and is not upgradable or scalable for supporting higher number of channels. Zhou et al. [50] presents the feasibility of coherent beam combining using numerical analysis, then they verify their results using two and three fiber amplifiers. Wang et al. [44] demonstrates coherent beam combining using a 9 channel (3 x 3) 1.14 kW Master Oscillator Power Amplifier (MOPA) array using a controller that is based on a Digital Signal Processor (DSP) core. The controller is capable of 50,000 iterations per second resulting in a 4.1 times improvement at the target beam quality. Yu et al. [49] characterizes commercial fiber amplifiers and then they exhibits 4 kW, 8 channel coherent beam combining with 78% efficiency. Goodno et al. [22] demonstrates phase locking of 5 coherent fiber tips using an FPGA at 100 kHz iteration speed. Huang et al. [22] shows an experiment with 6 channel phase locking and beam steering operation with a dither frequency of 1 kHz and beam steering up to 40 MHz speeds. Xiong et al. [45] presents an experimental study showing that they combined 2, 4, and 6 channel fiber arrays with separate tip-tilt controls using an FPGA based controller. Geng et al. [21] reports that they have phase locked 7 channel phased-array with additional tip tilt control for beam steering, and recently Weyrauch et al. [47] demonstrated a successful coherent beam combining of 21 laser beams over 7 km with reported SPGD rates of 140,000 iterations per second. As a side note, all of the above methods utilize the SPGD algorithm or variations such as a delayed version. A detailed explanation on SPGD algorithm is given in Chapter 3 along with the experimental implementation.

1.4 Thesis Overview and Outline

Mitigating atmospheric turbulence effects in real-time applications such as imaging and laser beam projections requires efficient algorithms coupled with high performance platforms. Furthermore, these algorithms need to be optimized and tuned for the specific application needs for dynamically changing environments. In this dissertation, to help with this algorithm development and optimization process, and to meet the performance needs of these applications, two separate platforms coupled with high speed electronics and novel frameworks have been developed.

In Chapter 2, the methodology for platform design for real-time imaging through atmospheric turbulence applications are presented. First, turbulence effects in imaging by examining real world captured turbulence data from actual ground-level turbulence, and artificially generated turbulence in the lab are explained. Based on this data, the performance needs are defined, the novel methodology and platform design are introduced. The presented platform has two parts: hardware and software. The hardware is built using mostly off-the-shelf parts that are capable of running parallel executions and includes additional modules to interface with different cameras. The software is built as a platform-independent framework that is used for developing image processing algorithms for turbulence mitigation and real-time testing of these algorithms by utilizing the hardware component for parallel executions. The chapter is concluded with a case study that shows the platform in action with the implementation of a popular turbulence mitigation algorithm called Lucky-Region Fusion (LRF). Finally performance metrics are presented and the results are discussed.

In Chapter 3, the focus is moved to atmospheric turbulence effects on a specific

subset of beam combining applications called coherent beam combining (CBC). This specific approach utilizes phased arrays to deliver a coherent beam to the target with increasing performance requirements depending on the number of array elements which is an important challenge for a controller platform. These challenges are targeted with a unique platform design by exploiting parallelization of algorithms using special hardware and emulation of it in software for speeding up the design process. First, our novel software framework that is used to develop algorithms for phase locking or beam steering in fiber laser arrays is introduced. The framework building blocks, simulation types, and hardware integration for an actual system emulation are explained. Then, an innovative hardware engine that can run these algorithms efficiently while keeping up with the increasing array element requirements is presented. The hardware engine is built around a processing subsystem that consists of three ICs which are used for specific tasks from monitoring the algorithm operation to driving the array elements. Also the careful planning and architectural design of the electronics that goes with the processing subsystem are discussed, The performance metrics are shown, and the integration of whole platform operation is explained. Next, a popular beam combining techniques called Stochastic Parallel Gradient Descent (SPGD) method is explained and implemented to test the platform operation on a 19-channel hardware loopback module that is developed to simulate phase locking operation. Finally, the platform operation on an 7-channel optical testbed for operation is demonstrated for performance verification.

Finally, in Chapter 4, an overview of the presented research and two unique

platform development studies are given, contributions to the field are listed, and different scenarios and applications where these platforms can be used are presented. The chapter and the thesis is concluded with the discussion of possible research areas and potential improvements for future work.

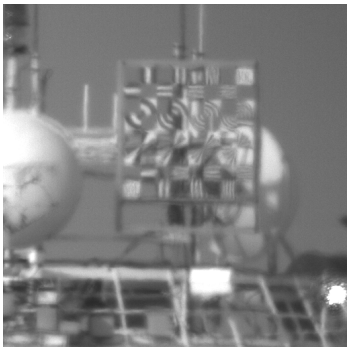
Chapter 2

REAL-TIME IMAGE PROCESSING PLATFORM FOR ATMOSPHERIC TURBULENCE MITIGATION

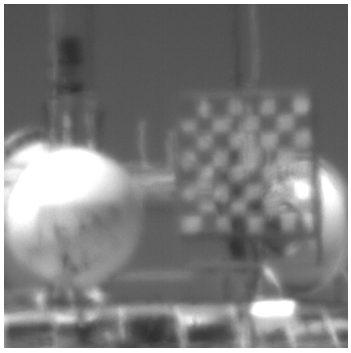
2.1 Introduction

Many modern day applications require obtaining high resolution images or video streams over long distances such as target tracking, drone observations, sports, astronomy and reconnaissance. However, long range target visualization is generally hindered by turbulence due to adverse atmospheric conditions such as humidity, temperature and wind shear. This issue leads to spatial and temporal distortions of the target image. Figure 2.1 gives an example of the atmospheric turbulence effects in long range imaging, and how changes in the atmospheric conditions affect the captured images.

In literature, there has been a lot of work published in atmospheric turbulence characterization and modeling [25, 26, 10, 1, 12, 39, 15] and mitigation techniques that are explained in Chapter 1. One of the problems in mitigation techniques is that most of the work in algorithm development is done in frameworks and tools that are built for image processing, and not intended for running these algorithms in real-time. For some cases, the algorithm's target might just be to generate a single image of the target and not a video sequence. For those use cases and applications, although not very efficient, these tools are usually sufficient and they satisfy the system requirements. However when the applications require a real-time visualization of the target, these tools fall



(a)



(b)

(c)

Figure 2.1: Atmospheric turbulence effects on a resolution chart attached to a water tower taken at different dates under various weather conditions at 4.3 km away from the camera lens. Image a conditions are not recorded. Image b is taken on a sunny day at 3 pm under 86 F degrees, 58% humidity. Image c is taken on a cloudy day at 12:15pm under 84 F degrees, 65% humidity.

short and underperform. The same problem arises when generating a turbulence mitigated video sequence via these tools. First, a sequence of frames or a short video of the turbulence degraded target is fed into an algorithm, then supplementary reference images are generated, and finally the output frames are recorded to be played back after the whole operation is complete. As the single frame generation approach, recording a video sequence to be played back later might be sufficient for a given application. However, it is not nearly good enough if an instant feedback is needed from the target.

Usually existing tools and frameworks are used for developing atmospheric turbulence mitigation techniques. Most of these tools and frameworks utilize only the main processor called Central Processing Unit (CPU) which is intended for general purpose computing operations and not for high performance in computationally intensive applications. For the computationally intensive applications such as graphics rendering, usually an alternative processor is used called a Graphics Processing Unit (GPU) that takes advantage of it's many cores designed for data parallelization. Most notable tools used for algorithm development are MatLab and Open Computer Vision (OpenCV) [7] which recently introduced GPU support for some of their filters. However, the speeds are not enough for performance critical applications where multiple filters need to be tied together. Therefore, a framework is created that can be used for algorithm development by using the built-in image filters and can run the implemented algorithms in real-time for long range imaging applications. To support the framework with development process and real-time execution, a platform is built that takes advantage of hardware acceleration and a frame grabber is added to the platform to connect a custom camera for different on-field experiments.

The rest of the chapter consists of the following highlights. Atmospheric turbulence effects in imaging applications is described and a simplified model is given to further enhance the understanding of these effects. Then, convolution operation, a common image processing filter, is explained, and is implemented on a CPU with parallelization. The parallelized CPU implementation performance results are given to support the need for hardware acceleration in real-time imaging applications. After that, the design methodology of the platform is presented by explaining the design considerations. Two parts are introduced: (I) The hardware to run the implemented algorithms and that is deployable for real-world scenarios. (II) The real-time image processing framework for rapid algorithm development and high-performance execution. The framework layers are explained and overall performance metrics are given and speedup based on the convolution operation is examined. Then, as a case study for testing the on-field performance of the platform, one of the popular turbulence mitigation algorithms called Lucky-Region Fusion (LRF) is explained and implemented with required real-time modifications. The implemented algorithm is tested with artificially generated and actual turbulence conditions and recordings. Finally, the performance metrics and results are presented.

2.2 Atmospheric Turbulence Effects In Imaging

As discussed in Chapter 1, atmospheric optical turbulence can take place due to variations in the refractive index in the path between the camera sensor and the target. Atmospheric characteristics such as temperature, humidity, wind, and pressure all factor into refractive index changes of the air. These changes in the refractive index

cause spatial and temporal variations of the propagation of the light passing through it. As a result, the incoming light to the ground-based imaging system, whether it is a telescope or a camera, results in a blurry and distorted image. An example to this can be the twinkling stars at night or the distant cars appearing distorted on the asphalt road in a hot and humid day. Developing a model for these issues with the consideration of high number of required parameters is a challenging task. Since all these parameters are rarely modeled as a linear problem, modeling the turbulence is a nonlinear problem.

Considering the two deformations: spatial and temporal variations, atmospheric turbulence effects in imaging on subsequent frames, at least to some degree, can be described using the following model:

$$f_k = D_k(K_k(f_{ideal})) + n_k \tag{2.1}$$

k represents the sequence number of the frames, f_{ideal} is the actual target image, K is the blurring mask, D is the geometric distortions, n is the additional random noise and f_k is the acquired distorted image of the target at a given frame. It can be seen that getting the actual target image, f_{ideal}, is an inverse problem. Several mitigation techniques are are available in literature based on this model that are given in Chapter 1.

2.3 Performance Needs

Image processing algorithms for turbulence mitigation often perform the same operations on a frame, such as edge detection or averaging, and repeat the same process

for subsequent frames. When real-time speeds are required, the displayed video needs to be more than at least 30 frames per second (FPS) for the video to appear smooth. Table 2.1 shows the maximum amount of time that is needed for a single frame to be processed and displayed before moving on to the next frame. For a 30 FPS video operation, all the necessary filters and algorithms need to be executed in 33.3 milliseconds or the application suffers from lagging or stuttering of the video.

Table 2.1: The maximum amount of time available to process and display the next frame at different frame rates

Frame rate (FPS)	Time (ms)
30	33.3
60	16.6
120	8.3

In order to determine the baseline of the processing needs for an algorithm and derive theoretical speeds that can be reached, one of the most commonly used image processing filtering methods, a convolution operation, is given as a case study.

2.3.1 Convolution Operation

Convolution operation is one of the most common filtering methods in image processing. A general 1-dimensional (1D) discrete convolution is defined as:

$$c(i) = (a * k)[i] = \sum_{n} a[i-n]\ k[n] \tag{2.2}$$

where a and k are the 1D discrete signals, and c is the convolution result.

In image processing where 2D is used, the convolution operation becomes a small matrix of numbers called a *kernel* or *mask* to be applied to each pixel in the image, and generate a weighted average of the neighboring pixels for a center pixel. The resulting image is the filtered image where filter type depends on the mask values and size. Considering these changes, the convolution operation for 2D space becomes:

$$C(i,j) = (a * k)[i,j] = \sum_{n} \sum_{m} a[i-n, j-m]\ k[n,m] \tag{2.3}$$

Figure 2.2 gives an example of a convolution operation.

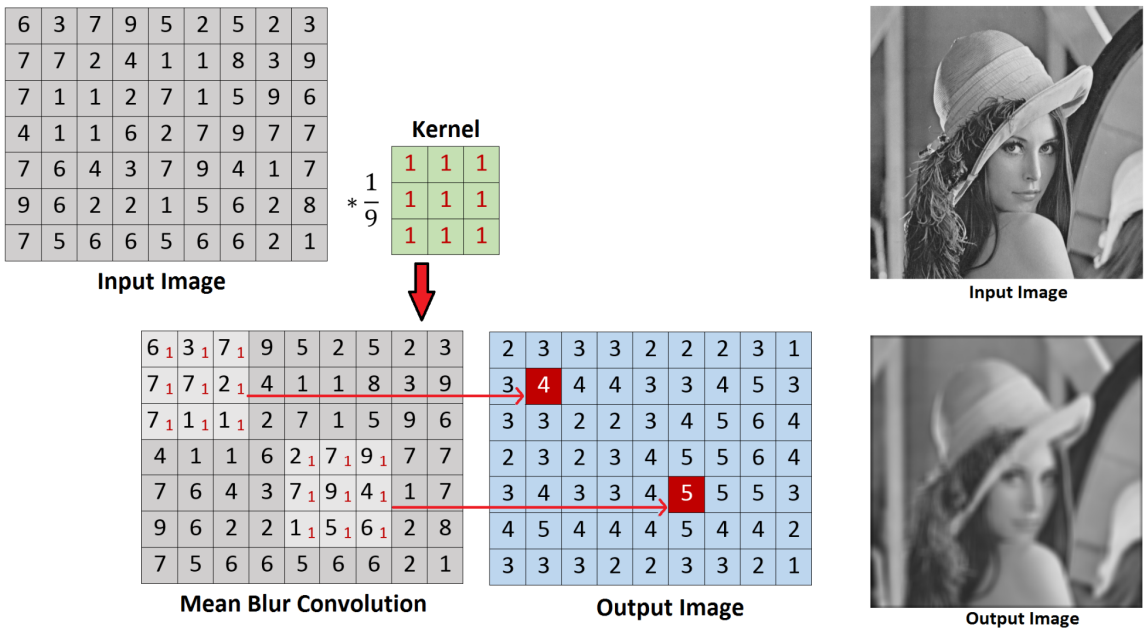


Figure 2.2: Convolution operation explanation and application on Lena with a blurring mask

In this example the input image is represented by a 9 x 9 matrix of pixel values

Table 2.2: 2D image convolution operation pseudo-code

```
Input : Image I (N x N)
Input :  Mask k (M x M)
Output: Image C (N x N)
for row = 0,...,N do:
  for col = 0,...,N do:
    sum <- 0
    for kr = -M/2,...,M/2 do:
      for kc = -M/2,...,M/2 do:
        sum <- sum +
          I[(row + kr) x N + (col + kc)] x k[kr x M + kc]
      end
    end
    C[row x N + col] <- sum
  end
end
```

to simplify the explanation and a 3 x 3 blurring kernel or mask is applied that averages each center pixel value based on its neighboring pixels. The resulting blurry effects of this type of mask can be seen on the image *Lena*.

Table 2.2 shows a pseudo-code for convolution operation implementation using a mask k with size M x M on a monochrome image I with size N x N.

Single-threaded CPU implementations of this code do not provide the necessary speed. Instead, this algorithm is implemented with the help of Open Multi-Processing (OpenMP) [13] which is used for parallelizing data-independent tasks on CPU. The Table 2.3 shows the convolution operation execution time with image sizes between 128 x 128 and 4096 x 4096 and mask sizes between 3 x 3 and 11 x 11 using OpenMP.

Table 2.3: OpenMP on CPU implementation of the convolution operation executed in 4 threads and averaged in 10 runs. The results are in milliseconds.

Image size (float)	Mask size 3x3	5x5	7x7	9x9	11x11
128 x 128	0.23	0.66	1.22	1.98	2.99
256 x 256	1.12	3.77	4.30	8.26	13.54
512 x 512	8.98	18.26	23.39	32.03	43.53
1024 x 1024	24.17	34.32	78.51	125.67	185.72
2048 x 2048	60.63	167.50	309.08	510.59	720.96
4096 x 4096	280.97	642.76	1138.27	1747.19	2441.03

The results from Table 2.3 show that convolution operation on a 1024 x 1024 image with 5 x 5 mask size already exceeds the theoretical limit to preserve 30 FPS operation. These results support that current CPUs are not enough to handle image processing applications especially with larger image sizes. Alternative approaches, such as hardware acceleration, are needed to achieve real-time speeds.

2.4 Platform Design

The developed real-time image processing platform for atmospheric turbulence mitigation has two parts: First, the high performance hardware that can handle real-time execution for image processing and has an interface for camera integration. And second, a software framework for image processing that is designed for easy algorithm development and real-time operation.

2.4.1 The hardware

It is noted above that the real-time applications require hardware acceleration for seamless operation. The possible candidates for the hardware acceleration are a Field Programmable Gate Array (FPGA), a Digital Signal Processor (DSP) or a graphics processing unit (GPU). Overall, a GPU is chosen to do hardware acceleration, because of its faster development time over an FPGA, and processing power and resources over a DSP. For a potential smaller platform needs, a DSP can be considered as a strong alternative for its small footprint. Other Application Specific Integrated Circuit (ASIC) methods are also viable and even though theoretically they can perform faster, they require even more time for development and implementation.

The platform hardware consists of three main parts: a CPU, a GPU, and a frame grabber for a high speed camera.

- The CPU that is used is an Intel i5 processor. The main jobs for the CPU are, aside from maintaining regular operating system jobs, generate the calls for low-level processing blocks, observe the operations by interacting with GPU, generate performance metrics, hold state machines for the algorithms and provide a user interface.

- The GPU is an NVIDIA GeForce GTX 760 GPU which is a commonly used and relatively cheap off-the-shelf product from NVIDIA that utilizes 1152 CUDA cores and 2GB of DDR5 RAM. It has enough memory to store the necessary number of frames for processing, and is a good performance/price tradeoff. The GPU is the main hardware acceleration unit that handles most of the processing

block operations through parallelizing the operations. One of the benefits of having enough memory is that, frames from each processing blocks can be stored in the GPU and the final image can be rendered directly on the monitor without the need to transfer it back to CPU memory, which decreases the latency.

- A frame grabber for connecting a high speed camera using Camera Link interface. It is added to the platform to provide any field deploy scenarios if needed. Current system has a Camera Link frame grabber with a Basler acA2000-340kc high speed camera that is capable of resolutions up to 2046 px x 1086 px and frame rates up to 340 FPS. However, other cameras and frame grabbers can be integrated with little wrapper coding.

A general connection diagram of the hardware is shown in Figure 2.3.

2.4.2 The processing framework

The processing framework designed to make algorithm development easy and consists of multiple abstraction layers to make it easier for integrating new modules and additional functionality. The framework can be divided into two separate parts in terms of their purpose. First, the back end of the framework uses OpenCL [38] to execute the filters on the GPU for hardware acceleration, and second, the front end of the framework that uses Python [41] and Python wrappers for C [24] and OpenCL for visualization, interfacing and state machines.

There are two choices for the back end programming language that runs on the GPU: CUDA programming platform [32] developed by NVIDIA that can only be used in NVIDIA GPUs, and OpenCL framework that is maintained by Khronos Group

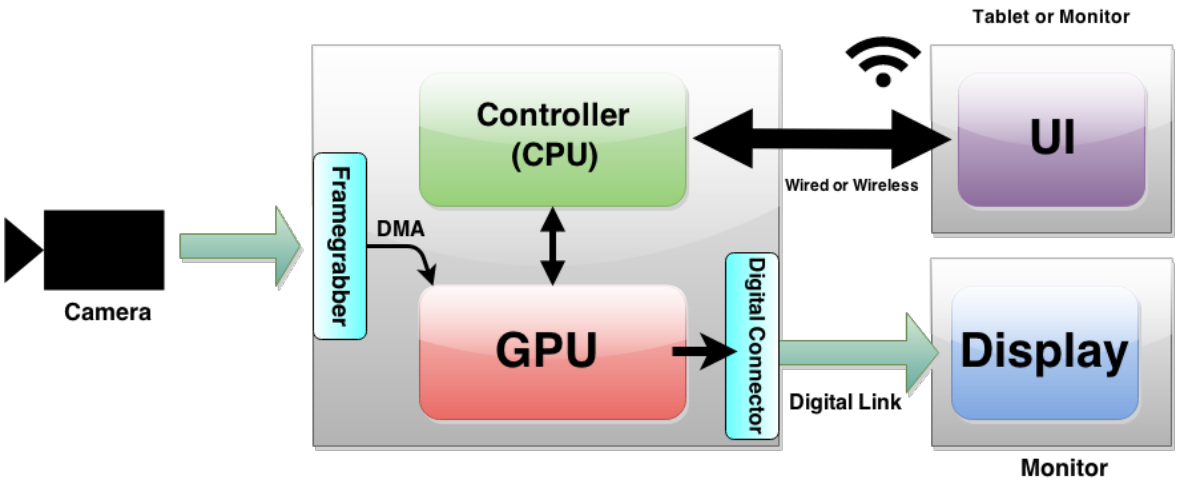


Figure 2.3: Hardware connection diagram

which can run on GPUs, FPGAs, DSPs and even CPUs. Overall, if the intended target is an NVIDIA GPU, the performance differences between two frameworks are negligible [17], and either one of them works. For this research project OpenCL is used, and the main reason for this is to make the processing framework not hardware dependent, and have future scalability options for possible FPGA integration or DSP core use.

For the front end, Python programming language is used for building a full application and organizing the blocks. One of the reasons for this is to make the development time shorter since it does not need compilation time and can be run instantly. It is an easy language to learn for beginners, and finally it is very easy to write function wrappers, whose main purpose is to call other subroutines or provide easy access to various libraries with almost no additional overhead, for different lower

level programming languages such as C/C++ and OpenCL.

Python is a high-level programming language, thus it is not ideal for performance critical applications and would take longer than necessary to execute implemented algorithms if used purely. However, since GPU is used to execute the performance critical processing blocks and applications, using Python as a wrapper for a lower-level functionality becomes an excellent fit for the framework. This approach also enables to hide away a lot of auxiliary declarations and function calls for GPU resulting algorithm development faster.

Aside from the front end and back end functionality, the software framework is divided into multiple application programming interfaces (APIs). An API describes a set of functions or routines that is used for executing a specific task, usually implemented to hide away the complexities or the procedures of the task that is being called by the programmer, and give out an easy integration with a third party software if needed. Figure 2.4 shows overall framework architecture design.

The framework consists of three main APIs:

1. Camera API is used to have a common API for using various cameras. This way it would be easy to replace the existing camera with a new one with a simple addition of a camera wrapper code that translates the camera software calls for the framework use.

2. Display API. By displaying the images directly from GPU memory, significant speedups can be achieved. The main reason for this is to eliminate the memory transfers from GPU to CPU when displaying the frames. In fact, this is one

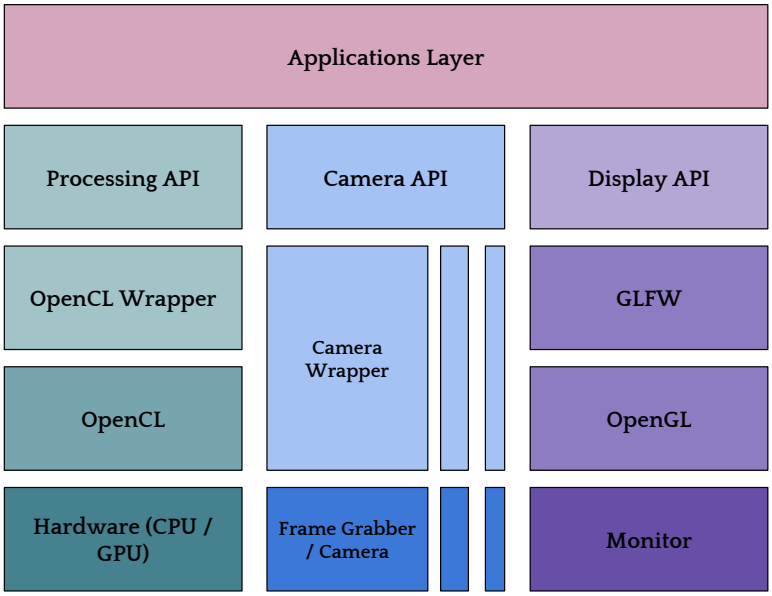


Figure 2.4: Framework layers

of the most important aspects of the framework for decreasing the latency and improving the performance. For this purpose, Open Graphics Library (OpenGL) [36] is integrated. OpenGL is an API used for rendering vector graphics by interacting with GPU that is commonly used in visual simulations, video games, 3D animations and computer-aided design (CAD). Although OpenGL has a broad range of features, it is currently only used to display images with the help of GLFW library to create windows and handle events.

3. Processing API is the interface to interact with the GPU filter blocks and algorithms. Each of the implemented filters or algorithms on the GPU has an

equivalent wrapper that can be called using this API. Some of these filters include Sobel edge, Sharpening, Smoothing, Gaussian, Laplacian, Prewitt, Scharr, High pass, Mean and Median.

2.4.3 Platform performance

2.4.3.1 Convolution operation runtime

In order to evaluate the platform for performance, a convolution operation run is done with varying mask and image sizes. Table 2.4 shows the convolution execution time of a Gaussian mask with 5 different sizes, between 3 x 3 and 11 x 11, applied to different resolution images. The resolutions that are chosen varies between 128 x 128 pixels and 4096 x 4096 pixels. Each pixel is a float4 type that can store 4-channels (RGBA) at 32-bits each. The values shown are the median value of 100 consecutive runs to eliminate any possible outliers.

Table 2.4: Execution time of a convolution operation for varying image and mask sizes. Each pixel is float4. The results shown are a median over 100 consecutive runs. The results are in milliseconds.

Image size (float4)	Mask size 3x3	5x5	7x7	9x9	11x11
128 x 128	0.02	0.03	0.04	0.05	0.08
256 x 256	0.06	0.09	0.15	0.19	0.28
512 x 512	0.24	0.34	0.55	0.71	1.05
1024 x 1024	0.99	1.39	2.24	2.89	4.33
2048 x 2048	3.96	5.54	9.15	11.52	17.12
4096 x 4096	16.14	22.69	36.79	46.50	62.77

It can be seen that for the image sizes below 1024 x 1024 resolution, the convolution operation takes at most 1 milliseconds with up to 11 x 11 mask size. At 1024 x 1024 resolution and above, especially with the higher mask sizes, It is important to note that the correlation between different pixel sizes is roughly 4X, so each time the pixel resolution is doubled, the time it takes to execute the relevant mask increase 4 times.

2.4.3.2 Latency and throughput

When trying to minimize the latency, it is essential to omit unnecessary memory buffer transfers between the GPU memory and external DDR memory. Table 2.5 shows the memory transfer sizes of different number of pixels between the host memory (i.e DDR) and the GPU memory. Each pixel is 4-channel (RGBA) and each channel is 32-bits. Thus making single pixel 128 bits or 16 Bytes. It can be seen that memory transfer actually depends on the data size and time increases linearly, Based on these values, total transfer speed or the throughput can be calculated to be around 6130 MBytes per second.

From these results it can be deduced that data transfer is the bottleneck of the application where it needs most of the time and it is an important factor in the the algorithm execution speed.

When a GPU function called a *kernel* is executed, the data needs to be transferred two times. First for transferring it from the host memory to GPU memory, and second for transferring it back to host memory for showing or displaying the results

Table 2.5: Memory transfer rates from host memory to GPU Memory with different image sizes. Each pixel is 16 Bytes and the transfer time is in milliseconds.

Image size (float4)	Total size (Bytes)	Time (ms)
128 x 128	256 kB	0.04
256 x 256	1 MB	0.16
512 x 512	4 MB	0.65
1024 x 1024	16 MB	2.61
2048 x 2048	64 MB	10.89
4096 x 4096	256 MB	43.02

which brings additional latency. In order to minimize this, a couple of different measures can be taken. One of these measures is to keep the data at the GPU all the time. This ensures the subsequent kernel calls receives the specific memory pointer to the GPU memory and starts execution right away. Another one of these measures is to directly display the data from GPU memory which eliminates the second data transfer step and lowers the latency. Open Graphics Library (OpenGL) or recently released Vulkan API can be used for rendering and displaying the GPU memory contents directly to the screen. Lastly, the memory can be transferred to the GPU using a direct memory access (DMA) from the camera to eliminate the extra latency with the host memory read/write operations.

2.4.3.3 Kernel launch setup time

One of the performance numbers that affect the execution time is the kernel launch setup time. Calling kernels in GPU is not a latency-free operation and it takes

some time for kernels in the queue to get ready. Therefore, when two kernels are called consecutively, an extra latency from this kernel launch overhead gets added to the execution time. Figure 2.5 shows a kernel call overhead timing between two filter calls over 1000 runs. On average, 60 microseconds is needed to launch the queued kernel on the GPU. Thus, the total number of GPU kernels need to be kept at minimum to eliminate the additional setup time penalties.

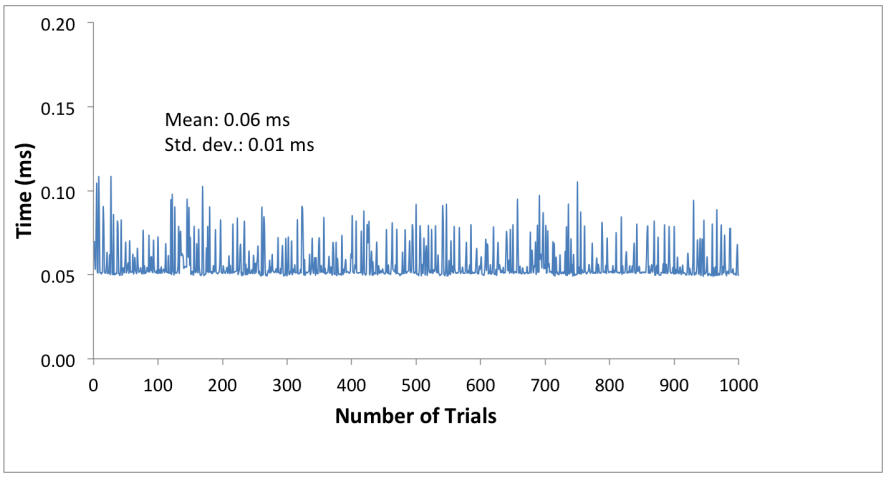


Figure 2.5: Distribution of kernel call overhead on the GPU over 1000 consecutive runs

2.5 Lucky-Region Fusion Implementation

In order to test the platform for real-time turbulence mitigation operation, an effecitve algorithm called Lucky-Region Fusion (LRF) method [4] is used as a case study. LRF is derived from a popular technique called Lucky Imaging [20] which is originally developed for observing the stars. Since the images that are recorded from the telescopes are degraded by the atmospheric turbulence, by using a high speed camera

and short exposure times, turbulence changes are minimized in the subsequent frames. The sharpest frames then averaged together to increase the signal-to-noise ratio (SNR) of the image. Fried [20] calculated the probability of getting a good short-exposure image through turbulence which is:

$$P \approx 5.6 \times e^{(-0.1557(D/r0)^2)}, \ for \ D/r0 > 3.5 \tag{2.4}$$

where D is the aperture diameter and $r0$ is the coherence length of the distorted wavefront.

Although the probability is high for an aperture diameter of 3.5 for a fixed $r0$, as the aperture diameter increases the probability of getting a good short-exposure image decreases as can be seen in the Figure 2.6. In fact for $D/r0$ around 6.5, the probability falls below 1%.

There are multiple reasons for choosing LRF method: LRF method by itself produces good results under mild turbulence conditions. It can compensate for the temporal and spatial dislocations given enough frame size and capture at high frame rates. More importantly, it performs very well with additional post and pre processing blocks to further enhance the image quality and add different functionality. This opens up a lot of possibilities for improvements and feature additions such as stabilization and object tracking and detection. Another reason is that it is computationally efficient to implement and run on hardware accelerators such as GPUs that takes advantage of many small processor to perform data-independent operations in parallel.

LRF tries to increase the odds of getting sharper frames by selecting sharp

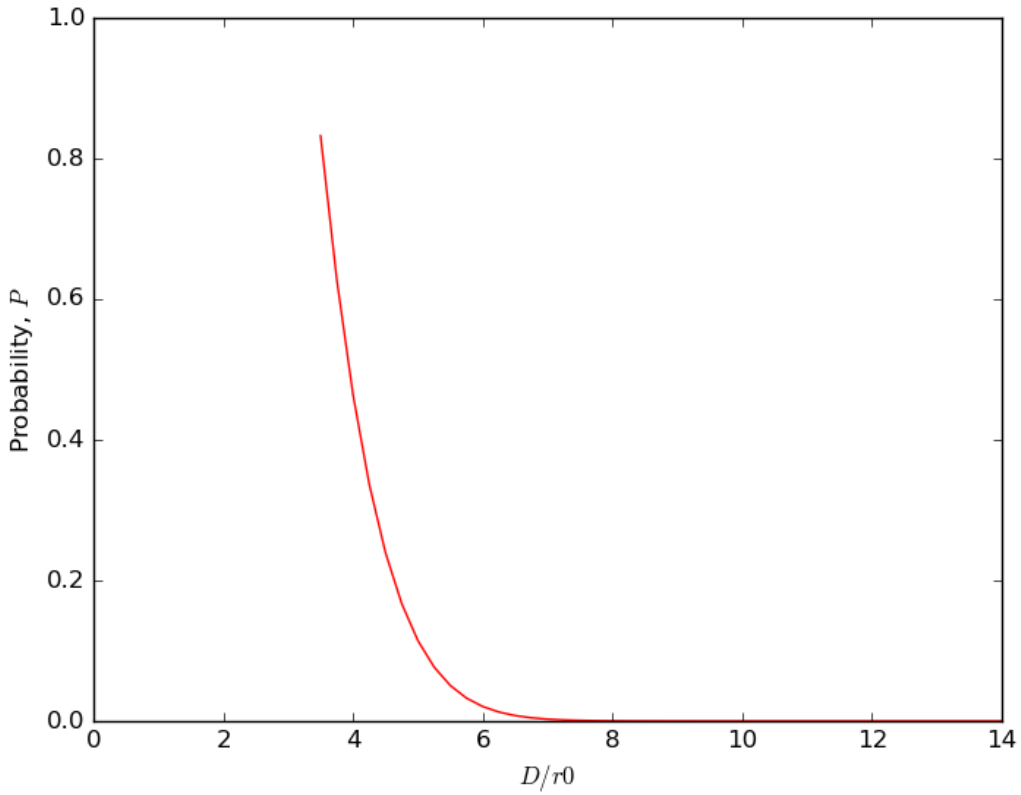


Figure 2.6: Probability of getting a good short-exposure 'lucky' image

regions of a frame instead of looking at it as a whole. One of the most important problems for lucky imaging is the image selection criteria. Depending on the selection criteria, a much sharper frames can be generated. In LRF method, the sharp regions of a frame are detected by using a metric called *image quality map*. Then the sharp regions are fused together to get the final latent image. Since the probability of getting a sharp region in a frame is higher than the whole frame being sharper, this approach

works better at getting a clearer latent images. The algorithm was developed and demonstrated on CPU as a post-processing method after recording the frames [4].

2.5.1 Overview of the algorithm

The base LRF algorithm consists of the following steps: First, an edge metric is generated from the input frame using the following relationship:

$$Q_n(r) = |\nabla I_n(r)| \tag{2.5}$$

where I is the input frame, n is the sequence number, r is the two dimensional location (x, y), and $Q_n(r)$ represents the *edge metric* which is the gradient of the input image frame $I_n(r)$.

Then, $Q_n(r)$ is used to generate the *quality map* of the frame using:

$$M_n(r) = (Q_n(r) * G(k, a)) = \sum \sum Q_n[r - k]\ G(k, a) \tag{2.6}$$

where $M_n(r)$ is the quality map of the frame and $G(k, a)$ is the Gaussian mask with mask radius a.

After getting the input frame quality map, the same steps are repeated to generate the quality map for the output frame:

$$Q_{out}(r) = |\nabla I_{out}(r)| \tag{2.7}$$

$$M_{out}(r) = (Q_{out}(r) * G(k, a)) = \sum \sum Q_{out}[r - k]\ G(k, a) \tag{2.8}$$

where $I_{out}(r)$ is the output frame, $Q_{out}(r)$ is the gradient of the output frame and $M_{out}(r)$ is the quality map of the output frame.

Finally these input and output frames gets fused together based on their quality map to form the the next output frame with the following equation:

$$I_{out} = \frac{I_{out}(r) \ \times \ M_{out}(r) + I_n(r) \ \times \ M_n(r)}{M_{out}(r) + M_n(r)} \tag{2.9}$$

Lastly, the initial setup of the output frame can be defined as:

$$I_{out}(r) = I_0(r) \tag{2.10}$$

where $I_0(r)$ is the first frame in the sequence.

As a final remark, it should be noted that different methods exist for both calculating the quality maps for the frames and fusing together the input and output frames using the generated quality maps. However, only one method is given in this section.

2.5.2 LRF implementations using existing tools

LRF algorithm is implemented using two separate methods OpenCV framework and scipy Python library. Both implementations are benchmarked using different size images. Table 2.6 shows the execution times for the Python implementation.

Once the image size hits 512 x 512 resolution, the total execution time exceeds 1 seconds, totaling at 1.6 seconds which is significantly poor performance for a real-time scenario. Increasing the image size to 2048 x 2048 resolution takes more than 20

Table 2.6: Python implementation of the LRF method using scipy libary. The results are averaged over 10 runs and shown in milliseconds. The image types are *float*.

Function type	Image size (float)				
	128x128	256x256	512x512	1024x1024	2048x2048
Sobel	0.82	3.02	17.89	66.11	298.05
Gauss	1.07	3.97	18.57	60.98	244.06
Fuse	75.38	351.00	1582.00	5238.66	20971.29
Total	77.28	358.00	1618.46	5365.76	21513.40

seconds to execute which is expected for an inefficient Python implementation.

OpenCV implementation of the same algorithm is shown in Table 2.7. It can be seen that the results are already improved by around 100-500x. Running the functions with 1024 x 1024 image size yields to around 21.8 ms execution time.

Table 2.7: OpenCV implementation of the LRF method. The results are averaged over 10 runs and shown in milliseconds. The image types are *float*.

Function type	Image size (float)				
	128x128	256x256	512x512	1024x1024	2048x2048
Sobel	0.07	0.23	1.10	5.53	22.99
Gauss	0.06	0.21	1.02	4.58	16.37
Fuse	0.04	0.22	1.53	11.68	31.56
Total functions	0.19	0.68	3.65	21.80	70.93
Misc. overhead	15.44	30.52	64.88	234.60	550.17
Total execution	15.64	31.21	68.54	256.41	621.11
Frame rate (FPS)	63.91	32.03	14.59	3.90	1.61

Total functions row show the total execution time of the functions themselves without any additional overheads. *Misc. overhead* row covers all the additional overheads for running the algorithm such as memory transfers and image display. When these times are added to the calculations, the total frame rates drop to around 63 FPS for 128 x 128 image size and 1.6 FPS for 2048 x 2048 image size.

A hardware-accelerated implementation of the LRF method is done and demonstrated by [29, 23] using an FPGA. The implementations utilized the parallelization of the filters, and reached up to 100 FPS. However, it was impractical for further development due to the excessive development and testing times for any feature additions. In order to make the development of the algorithm faster, an alternative hardware acceleration used based on Graphics Processing Units (GPUs).

2.5.3 Real-time implementation

For the real-time implementation of the algorithm explained above, first a 3 x 3 Sobel operator is used to get an edge metric, $Q_n(r)$ for each pixel in a frame which defines how sharp a single pixel is. Then a Gaussian mask with radius 3 is used to smooth out any random noise sources and generate a more reliable metric to value the pixels. The calculated values for each pixel is called *quality map*, $M_n(r)$, that effectively tells how good the pixel is relative to the others which becomes the final sharpness value for each pixel. After getting a quality map for each pixel, the same steps are repeated for the output frame, $I_{out}(r)$, Finally, a new pixel value gets calculated for each pixel based on its current and past values using the quality metrics as weights [8].

The described implementation is good for a fixed number of sequence frames. However, as the number of frames are increased, keeping an infinite number of previous frame information starts to become a problem by creating shadowing effects due to the stronger turbulence swings and any movement in the target scene. Hence, the algorithm needs to be expanded when real-time operation is required.

At least two additions are needed to address the shadowing problem. The first one is to limit the number of frames that are fused together, and second one is to remove the old frames from the fused stream. Although, by doing so solves the shadowing problem that might happen due to the spatial dislocations under some conditions, it does get rid of any sharp regions that are detected and thus the output frame always gets decided by the last few frames that are defined as the frame buffer.

2.5.3.1 Modified algorithm

Based on the proposed changes for real-time implementation, the algorithm is modified to calculate the fusing operation not between two frames, but between all the frames in the frame buffer. Also, another step is added to the algorithm to remove or defuse the oldest frame from the output result.

With these changes, the implementation of the algorithm is be done by introducing two new terms:

$$\begin{aligned} M_{total}(r) = I_{n-1}(r) \times M_{n-1}(r) \ + \ I_{n-2}(r) \times M_{n-2}(r) \\ + \ \ldots \ + \ I_{n-k}(r) \times M_{n-k}(r) \end{aligned} \tag{2.11}$$

$$M_{sum}(r) = M_{n-1}(r) \ + \ M_{n-2} \ + \ \ldots \ + \ M_{n-k}(r) \tag{2.12}$$

where $M_{total}(r)$ is the sum of all $I(r) \times M(r)$ values in the sequence, and $M_{sum}(r)$ is the sum of all $M(r)$ values in the sequence.

$$I_{out}(r) = \frac{M_{total}(r) - M_{n-k} \times I_{n-k}(r) + M_n \times I_n(r)}{M_{sum}(r)} \tag{2.13}$$

2.6 Results

The GPU implementation of the modified LRF algorithm using the developed framework is also done utilizing three separate kernels. *Sobel* kernel is implemented to get the edge metric of the image, *Gauss* kernel is implemented to get the quality map and *FusDef* kernel is developed to fuse the latest frame into the output and take out the oldest frame from the output.

Table 2.8 shows the execution time of each of these kernels on images with varying sizes in milliseconds. The *Total kernel time* represents the execution time of the kernels individually, and do not include any additional overheads. Execution time of three kernels starts around 95 microseconds for a small image size and peaks around 14.5 ms when the image resolution is 2048 x 2048 pixels. *Kernel setup times* adds 3 setup times totaling to 0.18 ms. Since displaying is done using OpenGL and do not require any memory transfer, it can be pipelined with the operation and can be neglected. Additionally, initial memory transfer from the camera to the GPU memory can be done automatically using DMA and also can be executed separately from the operation.

Table 2.8: GPU implementation of the modified real-time LRF implementation averaged over 100 runs shown in milliseconds.

Kernel Type	Image size (float4)				
	128x128	256x256	512x512	1024x1024	2048x2048
Sobel	0.03	0.11	0.41	1.48	5.89
Gauss	0.02	0.07	0.25	0.89	3.54
Fuse & Defuse	0.02	0.08	0.32	1.29	5.12
Total kernels	0.09	0.27	0.99	3.67	14.55
Kernel setup times	0.18	0.18	0.18	0.18	0.18
Total execution	0.17	0.35	1.17	3.85	15.00
Frame rate (FPS)	5712	2794	852.6	259.31	66.66

Figure 2.7 gives a comparison of these performance numbers with the previously given OpenCV implementation from Table 2.7.

From these two comparisions, it can be seen that the presented framework implementation has an execution speedup of around 5x. By adding the additional overheads from memory transfers and displaying, a total of 40 to 90x speedup is achieved that is shown in Figure 2.8.

From these results, it can be seen that running the algorithm in image size 1024 x 1024 on a GPU takes around 3.8 milliseconds to execute which theoretically can run at 259 FPS. This number includes all the necessary setup and execution times, however it does not include the memory transfer time. Memory transfer time is usually the bottleneck of these applications if not managed properly. By carefully timing the memory transfers from camera to the GPU using direct memory access APIs, the

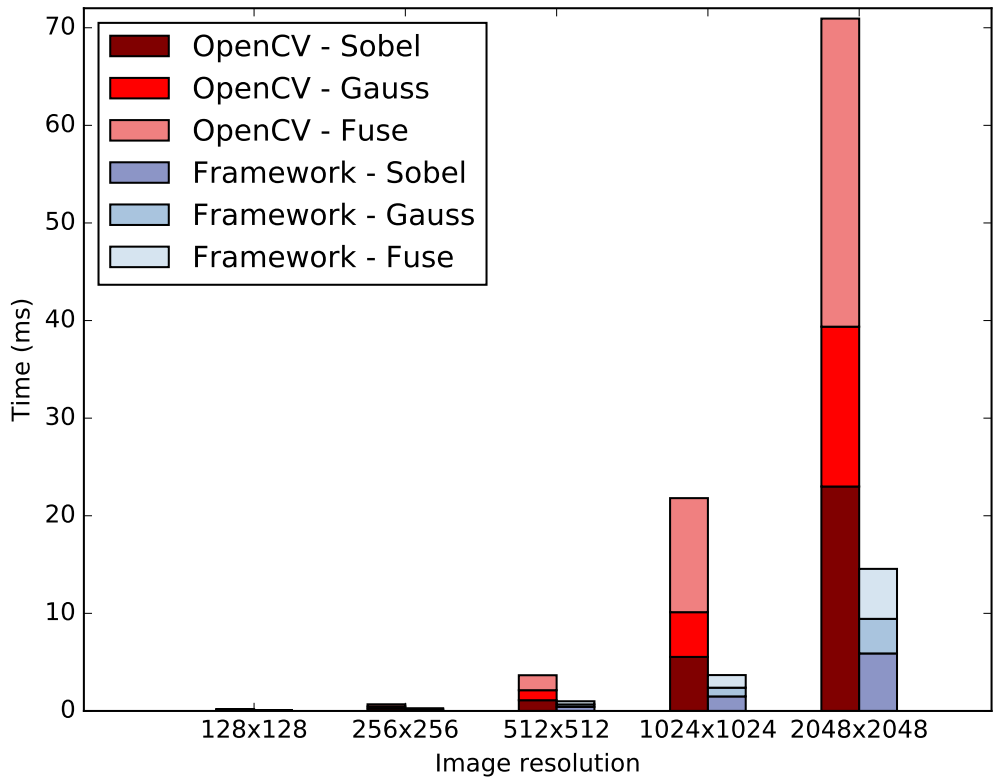


Figure 2.7: Comparison of execution times using OpenCV implementation vs. the presented framework implementation

previously stated theoretical speeds can be preserved. If a DMA option is not available, by adding the relevant memory transfer time of 2.6 milliseconds from Table 2.5 to the execution time of the algorithm, the total frame rate becomes 154 FPS for a 1024 x 1024 resolution 4-channel image.

Figures 2.9 and 2.10 show the final execution of the algorithm on both a turbulence degraded water tower data set and the artificially generated resolution chart

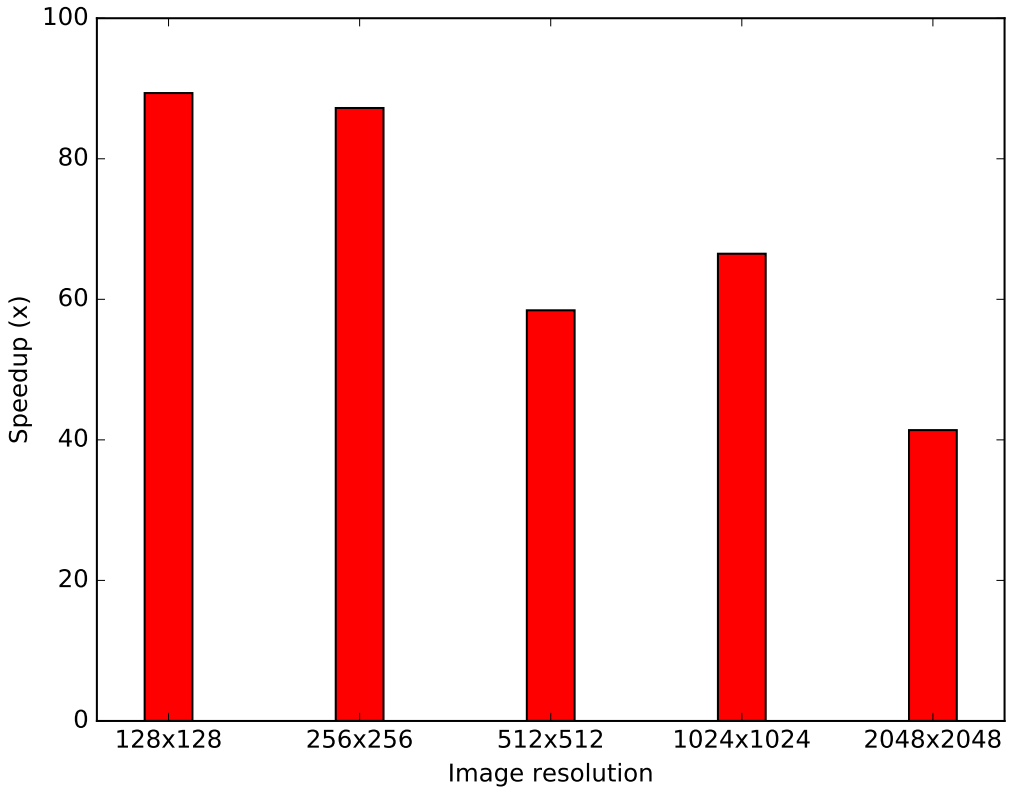


Figure 2.8: Total speedup of the real-time implementation of the LRF algorithm with the presented framework based on the OpenCV implementation

data set in the lab.

2.7 Conclusions

In conclusion, a real-time long range imaging platform that can be used for on-field experiments and applications by taking advantage of the hardware acceleration is built. The hardware presented utilizes a popular and relatively cheap GPU to exploit

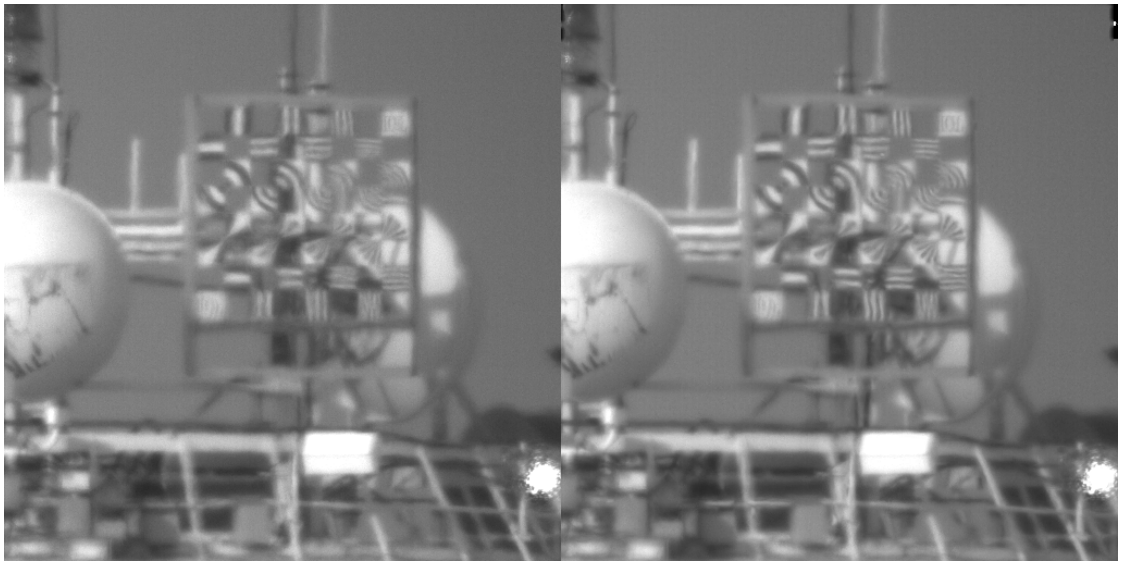


Figure 2.9: Water tower data set over 2.3 km distance under real turbulence. Left - unprocessed video. Right - LRF using Gaussian blur mask radius 3 and standard deviation 1.5

data parallelism, and a camera interface to integrate application specific cameras for image acquisition. In order to support the real-time operation on the GPU, a software framework is built that is easy to use and that has built-in filters for algorithm development for long range imaging applications.

The platform performance is presented using the convolution operation with different mask and image sizes. It is shown that the platform can perform 5 x 5 convolution operation on a 4-channel 32-bit 1024 x 1024 resolution image at around 1.4 milliseconds. The equivalent parallelized CPU implementation takes 34.22 milliseconds which is a 24 times speedup. When the mask size is increased, the performance speedup increases over 40 times between two implementations. In addition to the pure performance of the filter execution times, the framework and platform are implemented

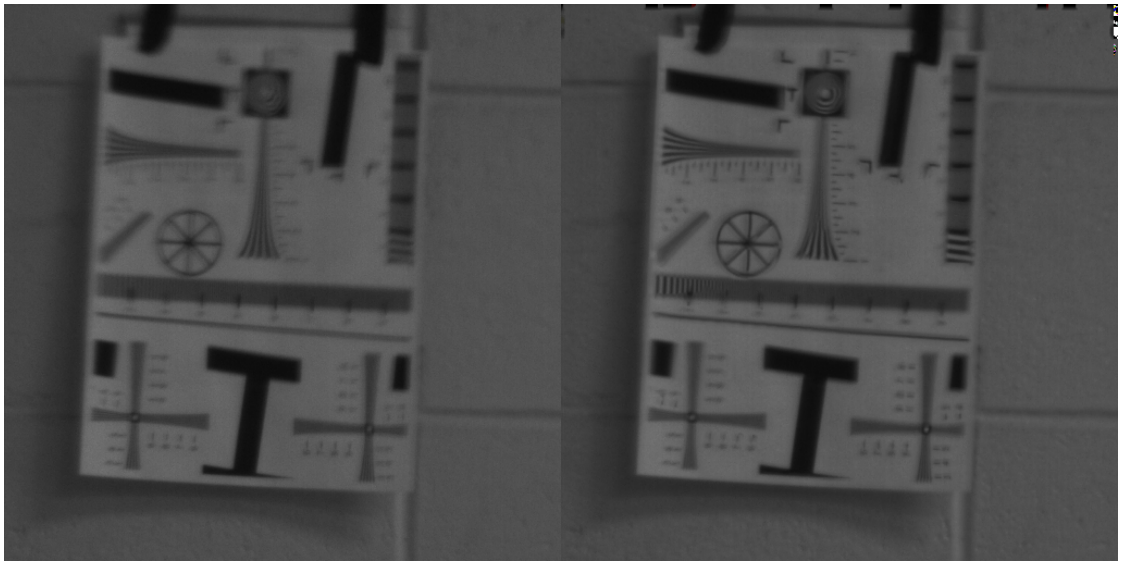


Figure 2.10: Artificially generated turbulence using a heater in the lab. Left - unprocessed video. Right - LRF using Gaussian blur mask radius 3 and standard deviation 1.5

to minimize the additional overheads for processing and visualizing at high data rates. In the presented comparisons, the presented framework implementation processes 4 times more data than the CPU version based on their data types which is not included in the performance speedup calculation.

Finally, an effective turbulence mitigation algorithm called Lucky-Region Fusion method that depends on processing high frame rates is examined, and a modified LRF implementation for real-time implementation is presented. The final implementation of the algorithm consisting of three separate steps take a total of 3.67 milliseconds with 4-channel 1024 x 1024 resolution on the GPU without any additional overhead. Including the one-way memory transfer time and kernel setup times, the LRF implementation can be run at 154 FPS.

Chapter 3

FIBER LASER PHASED-ARRAY CONTROLLER PLATFORM

3.1 Introduction

In laser beam projection applications, usually a single laser with a big aperture is used for transmission of a laser beam. When a laser beam propagates through the atmosphere, it undergoes various deformations such as scintillation, beam wandering and scattering due to the turbulent conditions [2]. These deformations cause the beam quality and efficiency to decrease. Increasing power requirements along with the lower efficiency of monolithic aperture telescopes under turbulent conditions pushed researchers into other beam projection techniques such as tiling smaller-size apertures together, and combining these beams at the target to generate a higher power, and more efficient beam.

Beam combining can be done using multiple techniques such as wavelength or spectral beam combining (WBC) and coherent beam combining (CBC) [16]. In this research project, the focus is only for coherent beam combining in phased-arrays where the phases of each array element are controlled individually to form a unified beam at the target by phase alignment.

Coherent beam combining is usually done with a fiber laser phased-array that is built with a fiber-coupled laser for generating the laser beam, multiple fiber splitters

to distribute the laser beam into multiple channels, phase shifter to control the phases, tip-tilt control for beam steering, of the fiber lasers and fiber amplifiers to increase the power output. Along with the mentioned optical system, a high-speed controller is needed to compensate for atmospheric turbulence effects. Controlling multiple phase shifters or tip-tilts requires the capability to compensate for atmospheric turbulence effects and requires optimized algorithms that run on efficient processing systems which are coupled with fast electronics [46]. A typical full system diagram is shown in the Figure 3.1, where FCL is the fiber-coupled laser source, FBS is the fiber beam splitter, FPS is the fiber phase shifter, FA is the fiber amplifier, FC is the fiber collimators and CL is the collimating lenses. The laser beams leaving the lenses go through the atmospheric turbulence and reach the target unorganized due to various distortions such as scintillation and phase-shifts. The beam intensity at the target spot gets detected with a far field metric detector (FFMD) and gets inserted into the controller where it controls the phases of the fiber channels.

Often in multi-laser control operations, either in phase-locking or beam combining, an off the shelf computer is used along with other electronics for interfacing the optical system. Although the electronics might be adequate to drive the components in the optical system, the processing power is usually not sufficient to handle fast operations. This problem gets worse when the number of control channels increases. In some cases, special hardware components are used to handle the processing tasks, but again the implementation is not efficient, and it is not scalable due to architectural design problems. In order to overcome these challenges and address the problems, we have build a modular and scalable hardware engine capable of fast vector operations

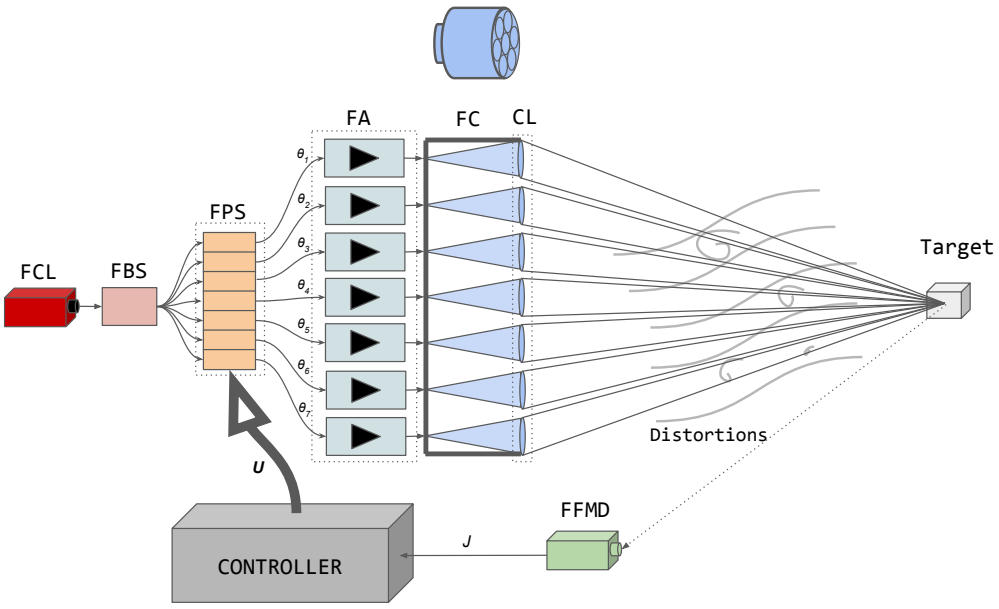


Figure 3.1: A typical 7-channel fiber laser phased-array elements and operation

and a large number of I/O drive capabilities.

The rest of the chapter aims to provide the methodology for high performance phased-array controller platform design for high power coherent beam combining. The chapter introduces the developed controller platform that consists of two complementary parts: (I) A novel analysis and simulation framework is implemented which is used for developing and optimizing algorithms for phase-locking and beam combining. The framework parts are described and simulation tools are explained. (II) An innovative hardware engine is built for driving multiple phased-array elements by utilizing the powerful processing back end and custom electronics for fast operations. The design process and decisions are explained and fast electronics operations are presented along

with the theoretical models.

The experiments start with the implementation of a popular beam-combining algorithm called Stochastic Parallel Gradient Descent (SPGD) method. The simulation results showing the algorithm performance are presented utilizing the framework. Then, the system performance in an open-loop and a closed-loop are calculated and verified. Finally, the experimental setup with 19 channel electrical loopback and a 7-channel optical loopback operation is presented.

3.2 Controller Platform Design

In order to achieve high-performance controller platform operation for coherent beam combining through atmospheric turbulence using high power fiber laser phased-array systems, two novel and complementary parts are designed and developed. The first part is an analysis and simulation framework that is built for developing and implementing algorithms for blind optimization problems such as multi-channel beam combining operation. The second part is the hardware engine used to perform fast calculations and drive large number of phase shifters and fiber positioners with minimal latency. Although the analysis and simulation framework of the first part can be used separately by emulating a controller and the phased array, it can also be combined with the second part by transferring the developed algorithms to the hardware engine for running experiments [11].

3.2.1 Part I - Analysis and simulation framework

In general, designing and developing new algorithms and optimizing parameters for custom hardware systems is a time-consuming task because it requires several

successive steps. First, the algorithm needs to be compiled for the target architecture. Then the compiled binary or the firmware needs to be transferred to the target hardware. Finally, the application needs to be executed. In addition to these steps, experimental runs may require (I) collection and storage of data from multiple sensors and (II) getting performance benchmarks of the hardware runs. Some of these experiments might even need to be executed over several days to obtain results of enough sample size to be safely generalizable. When all these steps get repeated for each iteration of the algorithm, depending on the tunable parameter size, the development time grows exponentially.

Based on this principle, an analysis and simulation framework is developed to simulate and predict how well the hardware would perform the implemented algorithm with given parameters under various conditions such as turbulence. Although the framework is implemented for phase-locking operation for phased-arrays, it can be used as a general blind optimization tool for many closed-loop applications that require multi-channel controls [9].

3.2.1.1 Framework construction and layers

The analysis and simulation framework consists of three abstraction layers: Applications and Analysis Layer, Algorithm Layer and the Target Layer.

The top layer, the Applications and Analysis Layer, is used for data visualization, data parsing, and implementing the overall iteration scheme for the target

application. It handles all the function calls to the lower levels for the performance-critical algorithms, and the hardware emulation for simulating the phased-array behavior if necessary. It is also used for visualization of the performance metrics from the hardware and state information for the phase shifters in experiments that are run in hardware.

The Algorithms Layer is the middle layer where the performance-critical algorithms are developed and implemented for beam combining. The implemented beam combining algorithms include two fixed parameters for reading the feedback metric, J, from the external feedback mechanism (such as a photo-detector), and the control values, $\mathbf{U}$. The control values for each array element in the phased-array are used for either phase controlling or beam steering. Keeping these two common parameters, new algorithms can be developed and new parameters can be added for various constants.

The bottom layer is called the Target or Hardware Layer. It consists of two application targets. The first one is the hardware engine for on-field experiments, and the second target is the software emulation of a phased-array and controller for speeding up the development process and optimization.

These abstraction layers are shown in Figure 3.2.

The Algorithms Layer of the framework is implemented using C programming language for several reasons. One of these reasons is to have a lower level access to the processor and better control over memory operations. Lower level access means the functions and operations are translated to the machine language directly and efficiently without going through any other abstraction layers. This lower level access and better control over memory is important since the implemented algorithms depend on fast

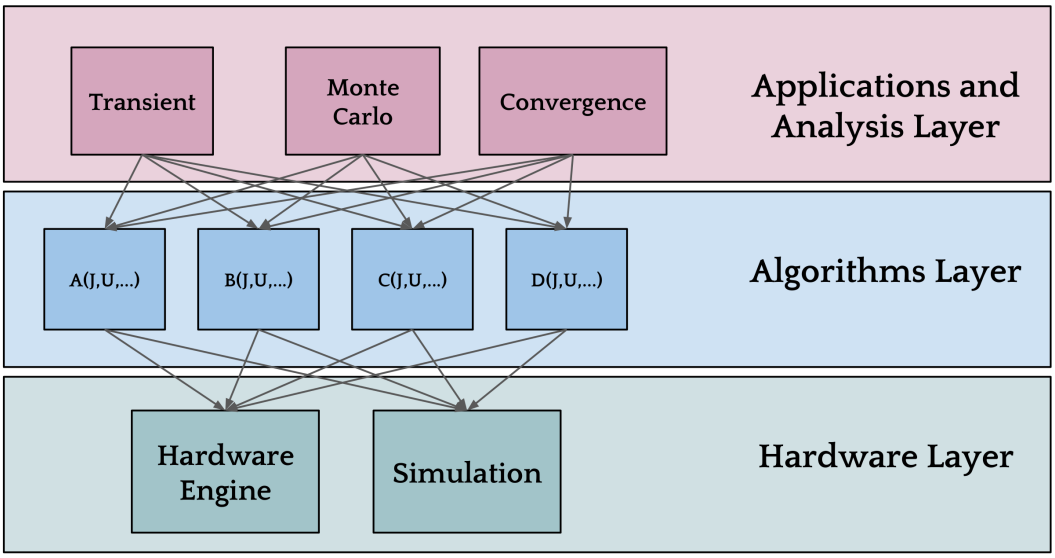


Figure 3.2: Analysis and Simulation Framework abstraction layers

execution times to be successful at compensating for laser beam distortions through turbulent atmosphere.

Another reason for using C language is to have the capability to compile the implemented algorithm for different platforms and architectures, which is called cross-compilation. This is especially important since porting the implemented algorithm to a different architecture would be a waste of effort and defeats the purpose of rapid algorithm development process.

The Applications and Analysis Layer of the framework is intended for user interaction for data visualization and data parsing where performance is not necessarily

a requirement. In addition, this layer handles all the calls to the Algorithms Layer and the hardware emulation functions for a simplified user-friendly experience. Therefore, Python programming language is chosen to perform these operations. Python has the capability to call algorithms and functions that are implemented in another programming language, like C, using an interface library such as C Foreign Function Interface (CFFI), Boost or ctypes. Python is also very effective at handling visualization and parsing of data with the help of its massive package database. Moreover, Python is one of the most popular programming languages, and easy to learn for beginners. However, it should be noted that Python is not essential since these goals can be achieved with any higher level programming language where there are libraries and tools for easy data parsing, visualization and function.

3.2.1.2 Framework operation

Algorithms that are run on dynamic conditions might require multiple parameters or coefficients for conducting proper operation. Based on these coefficients, it is important to examine the algorithm's behavior and control channel states through time with (or without) random effects. Therefore, the framework includes a visualization tool for transient analysis for observing the control channel states and the algorithm's effects on the feedback value. Figure 3.3 shows an example of this tool's operation.

The x-axis of the Figure 3.3 represents the number of iterations of the implemented algorithm. The reason for this axis to be tied to iterations instead of time is to separate the algorithm performance from the hardware performance. Once included, the hardware performance numbers can define the iteration speed which in turn

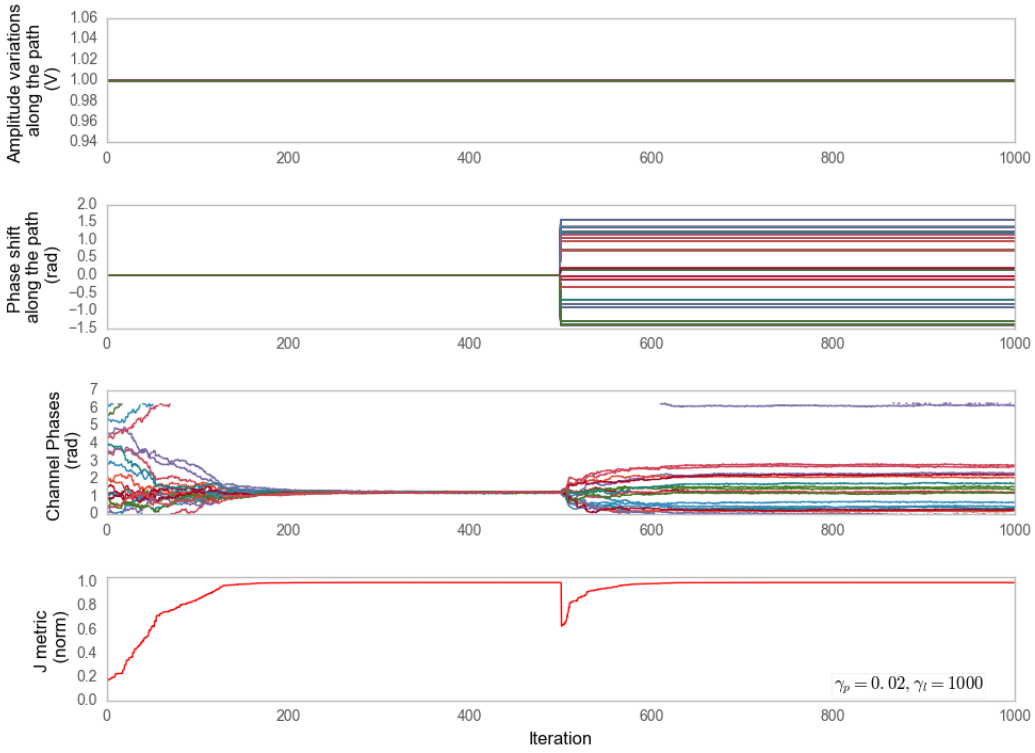


Figure 3.3: Transient analysis of the array elements, the outside noise value applied along the path to the laser beams, and the feedback value readings of the beam intensity based on the target phase information of the laser beams.

gives an assessment of how fast the algorithm performs. The top two columns of the Figure 3.3 represent the amplitude and phase distortions added to each laser beam's propagation path. These distortions can be either a sudden phase change to see the response of the algorithm or sinusoidal and random phase changes for observing the algorithm operation under dynamically changing conditions or scintillation effects on the amplitudes. As an example, at iteration number = 500, a random phase change

to each element is introduced along the beam path to see the algorithm response or convergence speed. The third column of the plot shows the actual phase values of the laser beams at the transmitter. These values are what is being controlled with the phase shifters to achieve higher beam intensity at the target. The final column of this plot shows the normalized feedback value representing the beam quality at the target. Having a beam quality 1 means the system has the maximum achievable beam intensity at value at 100% and the most desirable output. An additional feature here is the *convergence speed* which is defined as the rate at which the output achieves stability.

Additionally, it is important to estimate and optimize the algorithm parameters (or coefficients) in applications such as phase-locking of multiple fiber laser beams where the outside conditions are unknown and dynamically change. An effective way for achieving coefficient estimation and optimization is to run Monte Carlo simulations [14] by generating many runs with changing parameter values and observing the outcome of each run which can be classified as blind optimization. Since a phase-locking operation through unknown conditions is essentially a blind optimization problem with multiple parameters involved, the Monte Carlo method is implemented in the framework and is one of the data visualization techniques as shown in Figure 3.4.

The x- and y-axis of Figure 3.4 represent the two variable parameters that are being tested in the algorithm, and the heat-map shows a score of the statistical convergence of the algorithm and beam quality preservation for the given hardware and outside conditions. Having a higher number represents a better statistical convergence and beam quality.

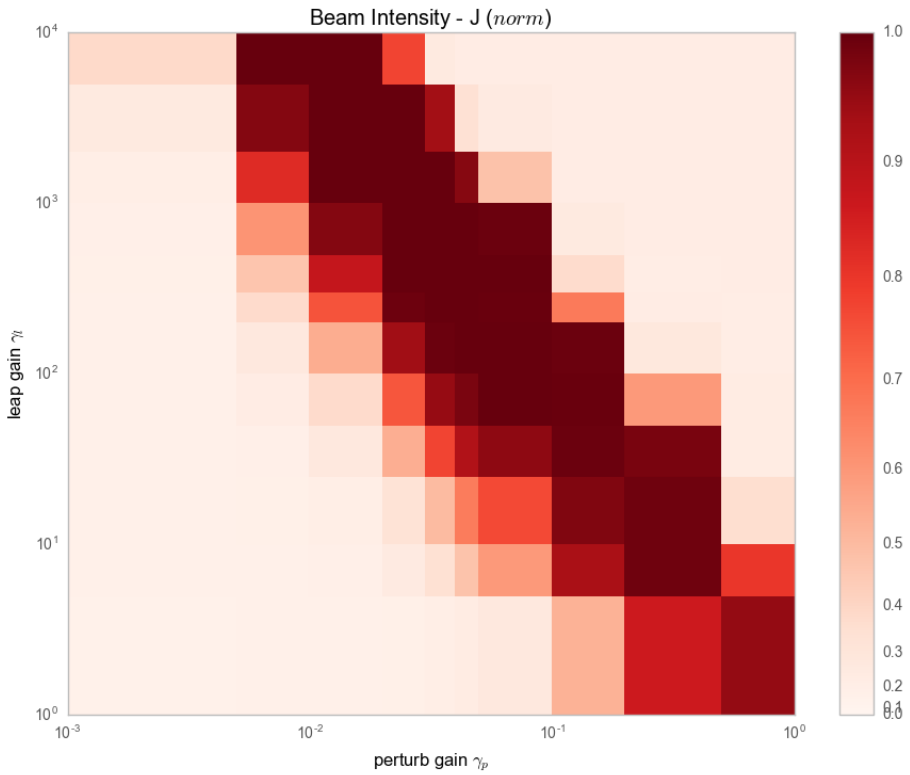


Figure 3.4: Monte Carlo simulations run for two different parameters

Overall, the analysis and simulation framework is an important aspect for algorithm development and parameter optimization in beam combining applications that use multiple control channels and depend on a beam intensity feedback value. The final property of the framework is that the implemented algorithms with their optimized parameters can be exported directly to the hardware engine for on-field experiments.

A final algorithm development flow is given in Figure 3.5 to help with development and optimization process.

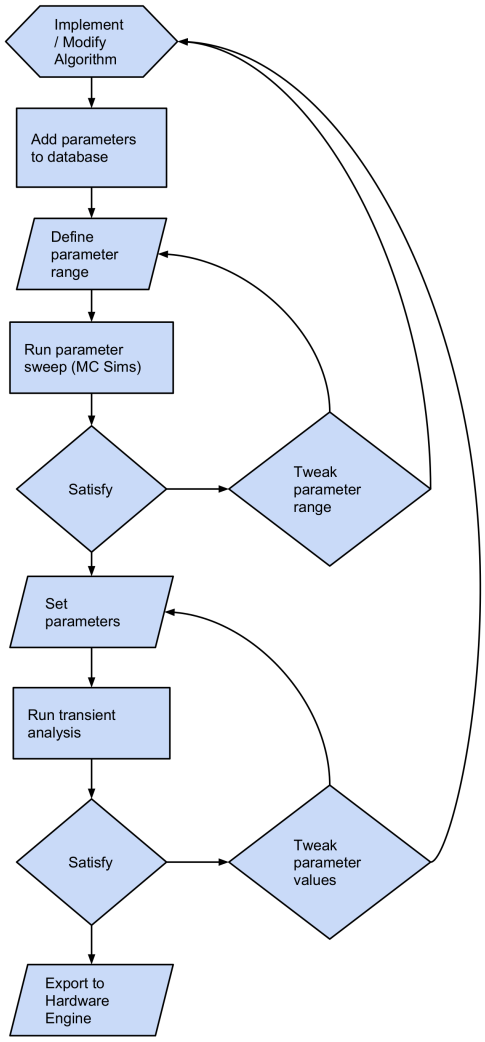


Figure 3.5: Algorithm development flow using the Analysis and Simulation Framework

3.2.2 Part II - Hardware engine

The hardware engine for operating the phased-arrays is designed to be modular and capable of standalone operation, and can drive many array elements for beam combining operation. Modular design is intended for future scalability of the system when the addition of new control channels are required for the phased-array. Moreover, the hardware engine can be used for other possible performance critical applications where controlling many channels at high speeds is needed (such as Adaptive Optics [34]). Standalone operation is also added for automatic beam combining operations to make the hardware engine independent of any other system. However, an optional user interface is built to also get feedback from the execution and performance metrics as well as direct control over the operation.

The hardware engine consists of multiple modules for modularity, but can be grouped together into three main categories: (I) The Processing Back End to provide the processing power of the controller to execute and generate new values for the phased-array elements and dispatch these results with minimal latency. (II) The Scatter Interface to take the values for array elements from the processing back end and deliver them to their respective targets. (III) The Gather Interface to receive that electrical feedback value about the beam intensity and pass it to the Processing Back end.

3.2.2.1 Processing back end

The Processing back end of the hardware engine holds the processing power for the controller and is built around three major integrated circuits (ICs):

1. An ARM (Advanced RISC Machine) Central Processing Unit (CPU) whose purpose is to monitor and control the operation flow, provide a user interface for observing system status and performance, show real-time plots and statistics, provide connectivity to the system and update firmware of other ICs.

2. A C674x floating-point Digital Signal Processor (DSP) core that is used for running algorithms for phase-locking or beam steering. It is based on Very Long Instruction Word (VLIW) architecture and can execute up to 8 instruction per cycle. The floating-point support improves accuracy over fixed-point and eliminates any possible quantization errors.

3. A Field Programmable Gate Array (FPGA) is used as a configurable latency-minimizing fanout interface between the narrow high data rate bus of the DSP and the large number of parallel lower data rate links used to drive the digital to analog converters (DACs) for each laser's positioning and phase control.

The ARM and DSP cores are part of TI OMAP-L138 dual-core SoC, and the FPGA is a Spartan 6 device from Xilinx.

ARM and DSP cores use a shared memory space to pass control information such as algorithm parameters, iteration speed and target distance between each other. These parameters are predefined before the execution is started. The user interface also has access to this shared memory space in order to change these control parameters in real-time and affect the operation of the system.

DSP and FPGA cores are connected together using TI's universal peripheral protocol (uPP) running at 75 MHz clock speed with the capability of sending 16-bits

per clock cycle. This channel is used to transfer the calculated phase values, **U** vectors, to the FPGA and receive the beam intensity feedback metric, J, from the FPGA.

Additionally, FPGA has some registers that are writable from DSP and ARM cores for defining phased-array and application properties such as the array element size and target distance. The connections between these three ICs are shown in the following Figure 3.6.

3.2.2.2 Scatter interface

Another important part of the hardware engine is the Scatter Interface. The main purpose of this part is to drive the control elements of the fiber laser phased-array based on the values it receives from the processing back end. It consists of three separate modules for scalability purposes.

Submodule 1 - The first module that is connected directly to the processing back end is a backplane that can hold 10 custom amplifier daughter modules. The module has the necessary level shifters that are used to translate the voltage levels to the required voltages based on the input signal specifications of the other components. In addition to holding 10 custom amplifier daughter modules, it is designed to be daisy chained together to increase the control channel capacity of the platform.

Submodule 2 - The second module is the custom amplifier daughter modules mentioned above. The purpose of these modules is to both translate the digital signals that are transmitted from the processing back end to the relevant analog values with the help of digital to analog converters (DACs), and amplify these analog voltages to the required voltage levels for controlling phase shifters. Each of these daughter

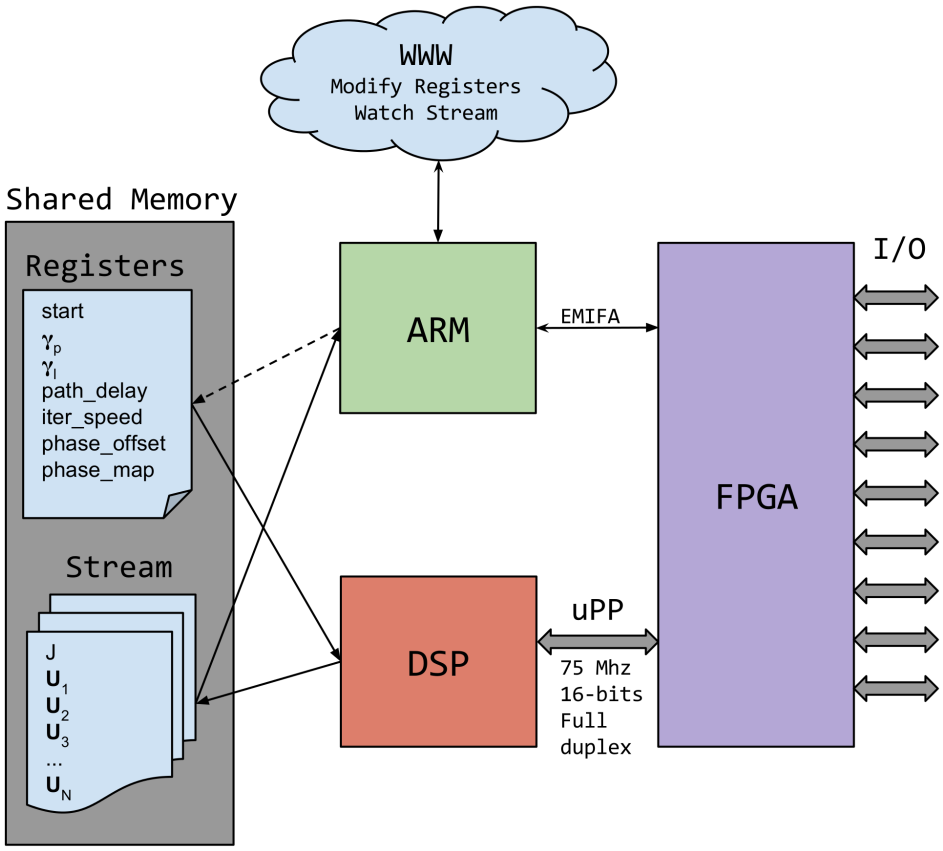


Figure 3.6: Processing Back end in the hardware engine connection diagram

modules has a 16-bit dual-channel DAC and two amplifiers attached to it. Therefore, each module is capable of driving two channels. Additionally, the digital lines are protected from any possible high voltage jumps from the amplifier supply voltages with opto-isolators.

The critical issue with the second module's operation is the load capacitance compensation. Driving a phase shifter requires a cable to be routed from the amplifier module to the phase shifter which introduces a load capacitance at the output of the amplifier. Therefore, additional circuitry is required to compensate for that capacitance so that the amplifiers can perform at high speeds without any ringing effects.

Building a circuit model requires the load capacitance that the amplifiers would drive. A fixed-length coaxial cable capacitance can be calculated using the following equation:

$$C = 7.3644 \times \frac{E_r}{log10(\frac{D_d}{D_i})}\ pF/ft \tag{3.1}$$

where E_r is the dielectric constant, D_d is the diameter of the dielectric and D_i is the diameter of the inner conductor. Based on this equation the capacitance of a RG316 coax cable with Polytetrafluoroethylene (PTFE) dielectric can be calculated as:

$$C = 7.3644 \times \frac{2.1}{log10(\frac{0.06}{0.02})} = 32.41\ pF/ft \tag{3.2}$$

From this result, it can be seen that each feet of RG316 coax cable introduces 32.41 pF capacitance. Assuming average cable length to be around 4 ft between the amplifier modules and the phase shifters, the amplifiers will be loaded with around 130 pF capacitance. Based on this load capacitance a circuit model is built as shown in Figure 3.7.

Circuit model is tuned and tested with a step response to a 0 - 10V voltage

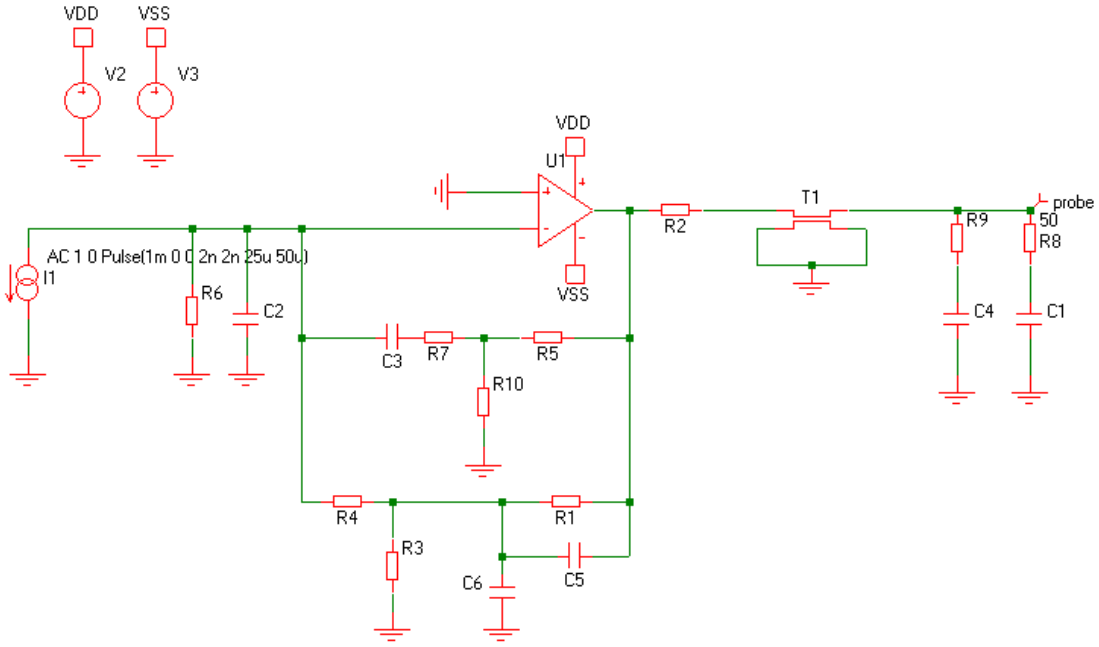


Figure 3.7: Amplifier circuit model for driving the phase shifters with additional coax cable.

swing with 4 different cable lengths shown Figure 3.8 a. Figure 3.8 b shows the fall time of the step response from (a) which is around 60 nanoseconds for all the tested cable lengths, 3.8 c shows the rise time of the step response from (a) which is around 60 nanoseconds for all the tested cable lengths.

Submodule 3 - The third module in the scatter interface is the termination module which is attached to the end of the daisy chained backplane module. The main purpose of this third module is to terminate all the digital signals.

High speed digital signals usually generated from the processing back end travel through the other modules. In order to preserve their signal quality, high speed digital

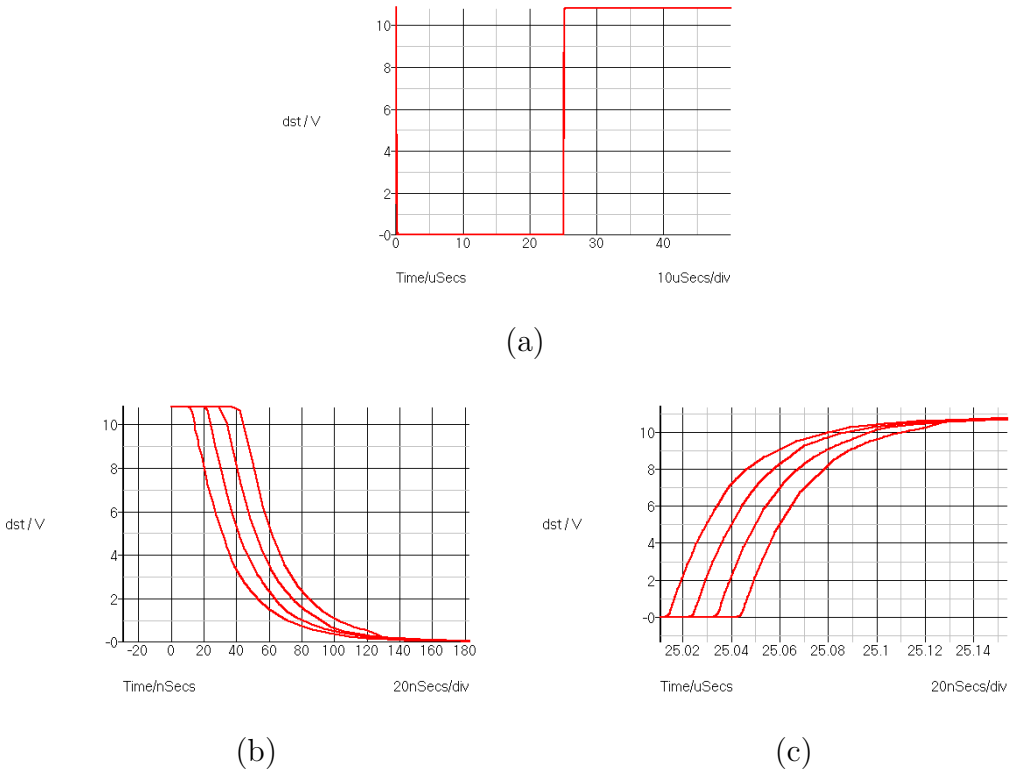
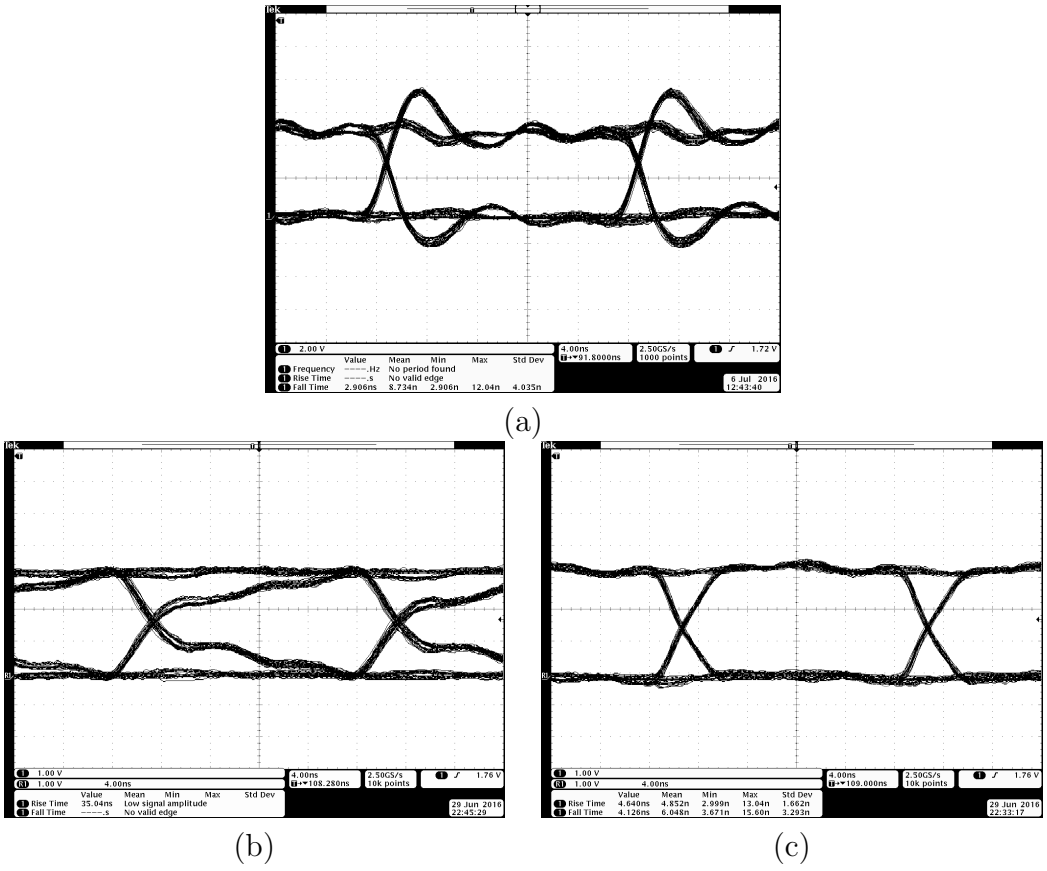


Figure 3.8: (a) 0 - 10V step response , (b) zoomed in fall time and (c) zoomed in fall time of the circuit model.

signals usually require some kind of impedance matching. Neglecting this fact can yield too many forms of distortion problems in the digital signals. These problems include (I) having a ringing at the transition edges which can cause false edge triggers, (II) overshooting or undershooting the signal levels at the transitions which may cause the receiver to not detect the edge, (III) damage the receiver if it exceeds the absolute limits of a sensitive receiver, or increase signal jitter. These problems are usually reduced by using proper terminations at those lines.

A termination of a digital signal refers to impedance matching of that signal

at the receiver or transmitter side to reduce distortions by minimizing any possible reflections. Available termination techniques include series termination by increasing the output impedance of the transmitter to match the characteristic impedance of the printed circuit board trace, parallel termination by adding an impedance matching resistor to ground at the receiver side to prevent reflections, and AC termination where a resistor-capacitor duo is attached to the end of the signal to ground. A detailed explanation can be found in Bogatin [6].



(a)

(b)

(c)

Figure 3.9: (a) no termination (b) poor termination (c) good termination for a single digital line going between 0 - 1 transitions.

In the controller, the digital signals that drive the DACs can go up to 50 Mhz speed and thus a termination module is necessary to preserve the signal quality for these received signals. Figure 3.9 presents eye diagrams which is used to see digital line characteristics such as the rise and fall times, signal quality, signal jitter, and maximum operating conditions by showing the random edge transitions of multiple iterations overlaid on top of each other in a digital line.

Figure 3.9 a shows the ringing and overshoot effects on the digital signal that does not have any termination. There digital line transitions have over and under shoot that introduces ringing effects which can both damage the subsequent circuit elements and create false transition effects. Figure 3.9 b shows an improved version of the line where a series and an AC termination is used. However, due to not properly matching the impedance, the signal quality suffers from slow rise and fall times that decreases the circuit operation frequency. Finally, Figure 3.9 c shows the same series and AC termination of the transmission line with proper impedance matching. The rise and fall time of the signal are around 4 ns and the overall signal quality is very good with slight fluctuations that are negligible.

3.2.2.3 Gather interface

The final part of the hardware engine is the Gather Interface which is used for receiving the incoming beam intensity metric, J, from a Far Field Metric Detector (FFMD) such as a photo-detector. Since this metric is an analog value, it needs to be digitized before sending to the FPGA, thus the Gather Interface consists of a 14-bit dual channel Analog to Digital Converter (ADC).

The special design considerations for the Scatter Interface digital lines are applicable here since the ADC can be clocked up to 50 MHz.

3.2.2.4 Final hardware engine

The final connection diagram of the Hardware Engine can be seen in Figure 3.11.

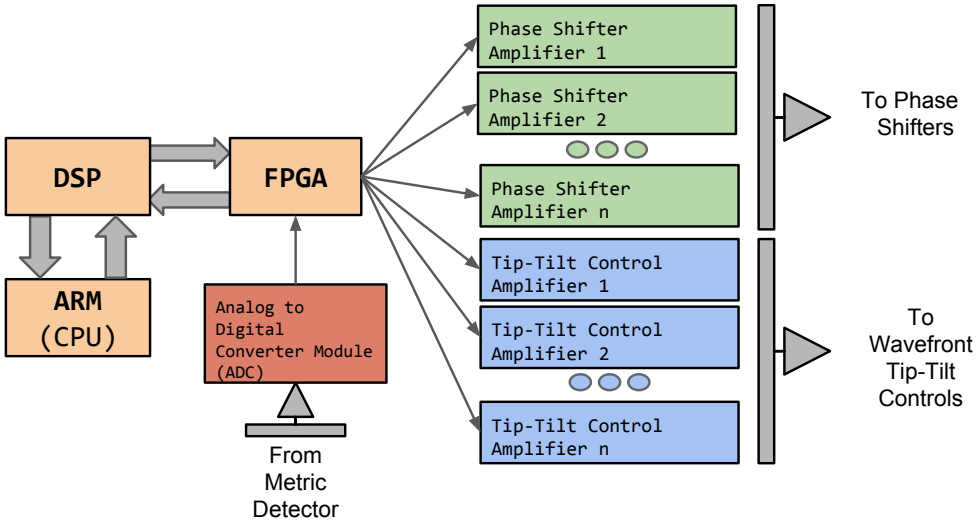


Figure 3.10: Final Hardware Engine connection diagram with the three modules: Processing Back end, Scatter Interface and Gather Interface shown in dotted lines.

Figure 3.11 shows the constructed Hardware Engine and the modules.

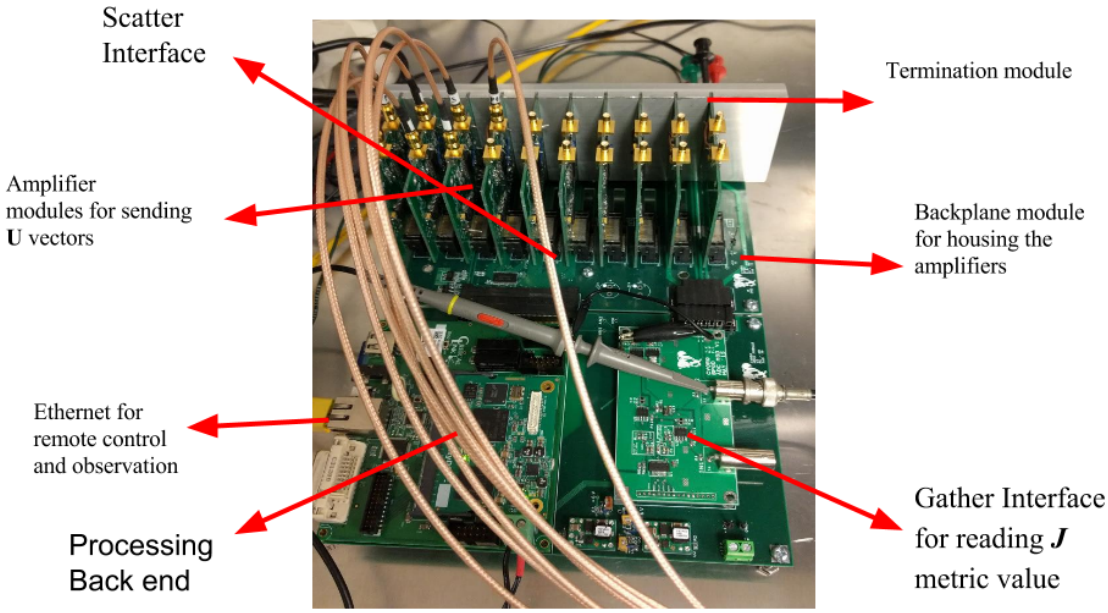


Figure 3.11: Final Hardware Engine construction with the three modules: Processing Back end, Scatter Interface and Gather Interface shown in dotted lines.

3.3 Experiments And Results

In the previous section, the methodology for high performance controller design for operating fiber laser phased-arrays through turbulence is presented, and the engineered system is shown. In this section, the simulation and experimental results are given using a phase-locking algorithm under various conditions to assess the platform operation. For the phase-locking algorithm, in literature there are two beam combining algorithms for multi-channel control operation. Multi-dithering [28] and Stochastic Parallel Gradient Descent (SPGD) method [43]. SPGD method is chosen

for implementation because of its simplicity and effectiveness under unknown turbulent conditions.

3.3.1 Stochastic Parallel Gradient Descent Method

Stochastic Parallel Gradient Descent (SPGD) method is a type of gradient descent algorithm which is a first-order optimization algorithm that is used to find the local minimum or maximum of a given function. The robust nature of the algorithm makes it an effective option for real world problems with poorly-understood, noisy, and time-varying channel impairments with multi-channel applications such as adaptive optics and beam combining. In literature, it has been demonstrated by Vorontsov et al. [42, 43] for wavefront corrections using large number of control channels in adaptive optics, and Liu et al. [43] for phase-locking of coherent fiber laser beams.

The algorithm works by applying small random perturbations to all of the control channels, $\mathbf{U}$, in order to maximize the system performance metric value, J, that is read back from a feedback mechanism such as a photo-detector for coherent beam combining applications. Figure 3.12 aims to visualize this process by representing the real-world system and unknown conditions as a black box.

In each iteration of the algorithm, the next values of the control channels, $\mathbf{U_{k+1}}$, are calculated based on the following Equation;

$$\mathbf{U}_{k+1} = \mathbf{U}_k - \gamma_l \; \delta J \; \delta \mathbf{U} \tag{3.3}$$

where k is the iteration number, $\mathbf{U} = \{u_1, u_2, \ldots, u_N\}$ is the vector output for each channel, γ_l is the leap gain coefficient, δJ is the variation in system performance

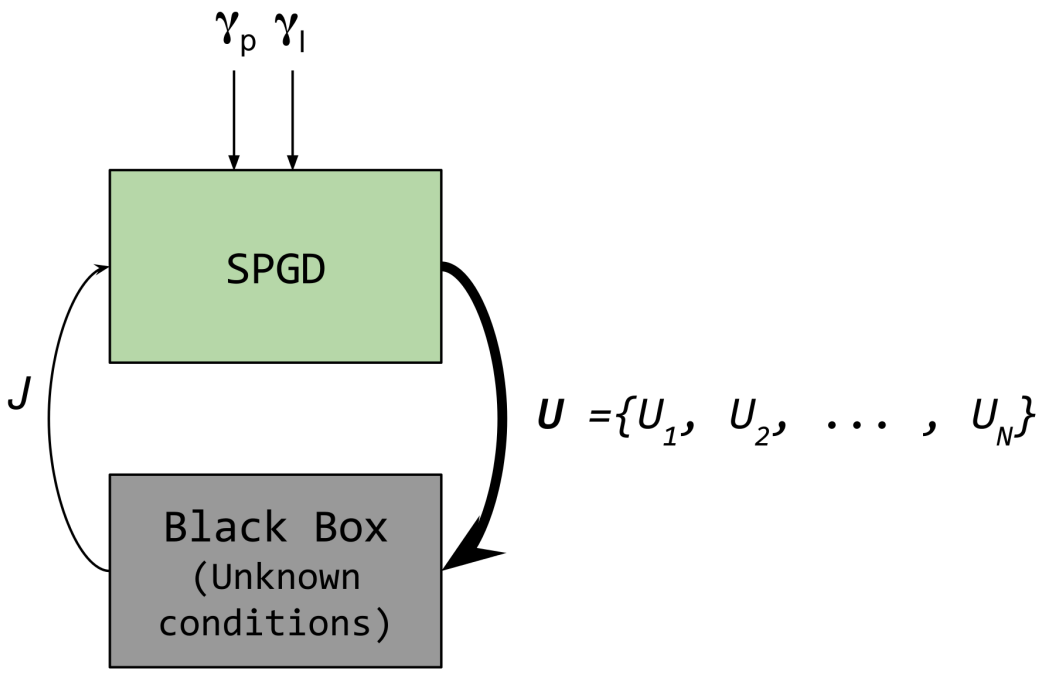


Figure 3.12: SPGD algorithm maximizing J output of the Black box unknown conditions by sending $\mathbf{U}$ vector values with the help of the control parameters, γ_p and γ_l.

metric, and $\delta\mathbf{U} = \{\delta u_1, \delta u_2, \ldots, \delta u_N\}$ is the random small perturbations which can be defined as;

$$\delta\mathbf{U} = \gamma_p \mathbf{X} \tag{3.4}$$

$$\mathbf{U} = \mathbf{U} + \delta\mathbf{U} \tag{3.5}$$

where γ_p is the perturbation gain coefficient and

$$\mathbf{X} = \{x_1, x_2, \ldots, x_N\} \sim U([-0.5, 0.5])$$

SPGD method is implemented and used in the rest of the experiments conducted below with 19 control channels, $\mathbf{U}$, and varying algorithm parameters: perturbation gain γ_p and leap gain γ_l values.

Each iteration of the algorithm consists of two steps. The first step is to compute the random perturbations for the channels according to Equation 3.4 and the second step is to generate the compensation values based on Equation 3.3. Figure 3.13 shows this operation.

3.3.2 Simulations

In this section, different simulations with various conditions are conducted in the analysis and simulation framework to examine the SPGD method operation and optimize the parameters γ_p and γ_l. Two important characteristics are important while looking at the operation: First one is the beam quality at the target being close to the theoretical limit which is represented by the feedback metric, J, and preserving the quality throughout the operation. The second one is the rate at which the algorithm converges for higher beam quality which is essential for correcting more severe turbulence conditions.

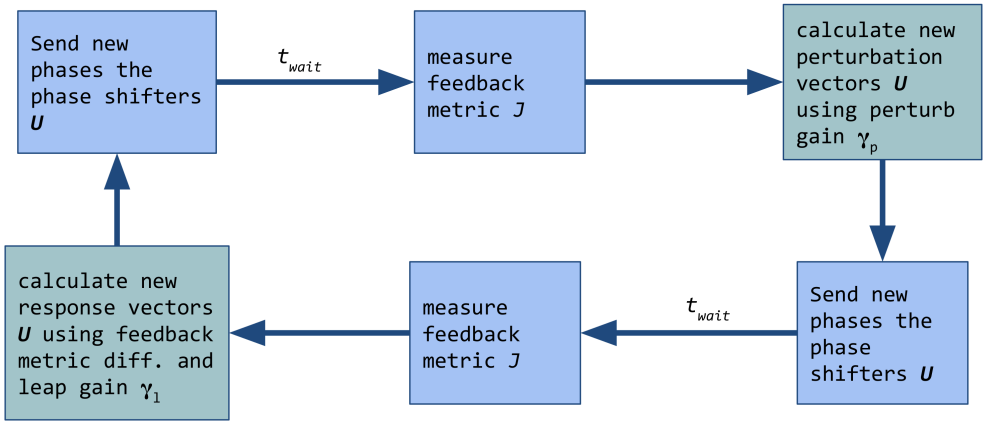


Figure 3.13: SPGD algorithm operation, each loop is defined as one full iteration, or two steps

The first set of simulations are the Monte Carlo (MC) simulation results for optimizing the parameters under different distortion conditions.

The second set of simulations are the results that shows the algorithm operation with each iteration called the *transient analysis*. Algorithm behavior is tested with different distortion conditions, and the plots are given.

The final set of simulations show the convergence speed with given parameters, and display how different parameters affect the beam quality and convergence speed.

3.3.2.1 Monte Carlo parameter sweeps

The first simulations for parameter tuning is the Monte Carlo (MC) simulations which are useful for assessing the system behavior based on given parameters. Each of these parameters are iterated through a predefined range of values to generate a large matrix of inputs for the algorithm. The results of these simulation runs then get displayed as a heat-map. For these simulations, the parameters in question are γ_p shown in x-axis and γ_l shown in y-axis for the implemented algorithm and the resulting heat-map shows the beam quality at the target normalized between 1 and 0.

Figure 3.14 shows the 19-channel phase-locking operation with ideal conditions, that is no disturbance is added to the laser beams.

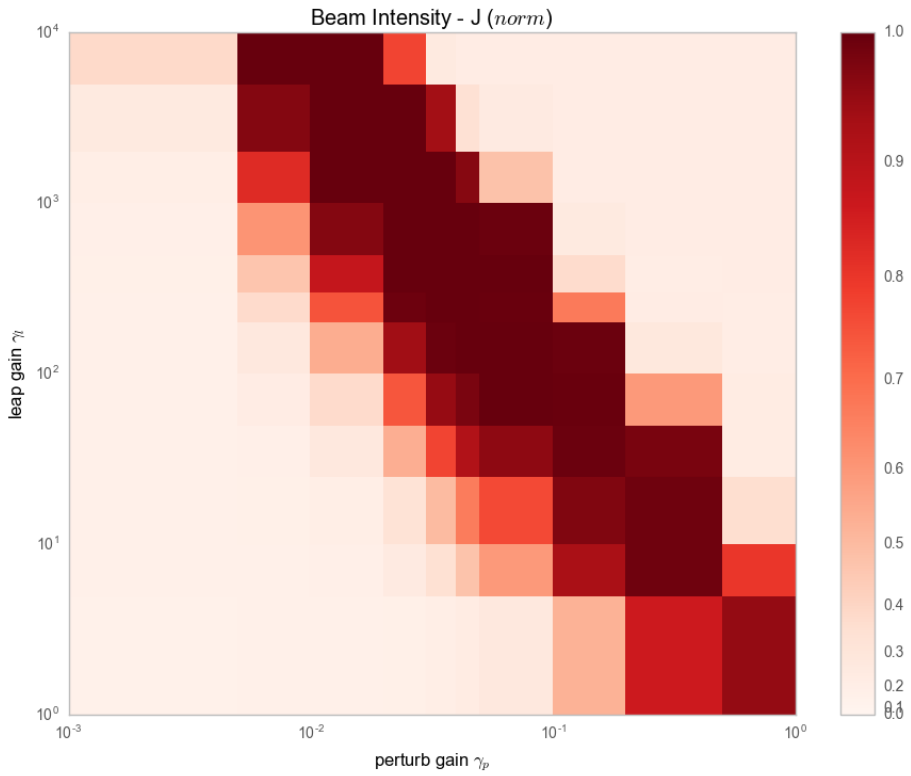


Figure 3.14: Monte Carlo simulation run for maximizing beam intensity of a 19-channel operation with varying perturb and leap gain parameters with no distortions added to the laser beams

It can be seen that there is a combination of parameter values that forms a line which generate a good quality beam. A possible combination of these values for $[\gamma_p, \gamma_l]$ pair can be $[0.1, 0.25], [0.05, 100], [0.02, 1000]$ which should generate phase-locking at small number of iterations and achieve high intensity beam at the target.

The second simulation run presented in Figure 3.15 shows the algorithm operation under a sinusoidal phase noise added to one of the channels for emulating dynamically changing turbulence conditions. The sinusoidal phase noise completes its period in 100 iterations and is between $[\pi/4, \ -\pi/4]$. It can be seen that the line is now thinner with overall lower beam intensity values, and the possible combination of the values that generate a good beam are much less than the previous simulation run.

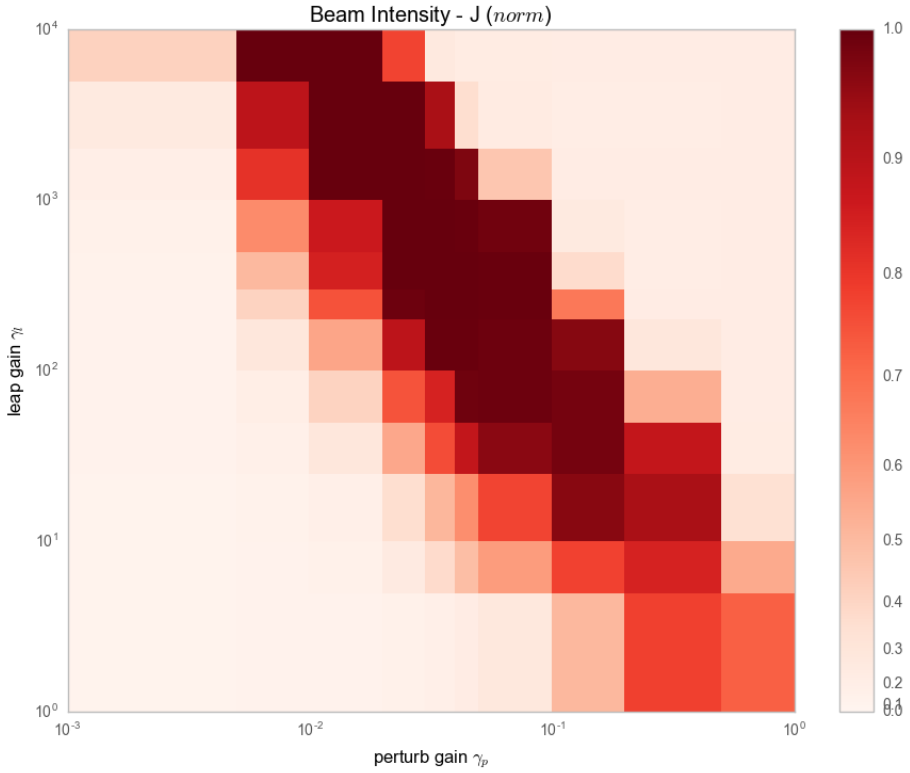


Figure 3.15: Monte Carlo simulation run for maximizing beam intensity of a 19-channel operation with varying perturb and leap gain parameters over a sinusoidal noise added to one channel

The final MC simulation run presented in Figure 3.16 shows the algorithm operation under a dynamically changing random phase noise added to each channel. The phase noise values added in each iteration, and changing between $[\pi/4, \ -\pi/4]$. It shows that the beam quality does not rise above around 80% with this type of noise.

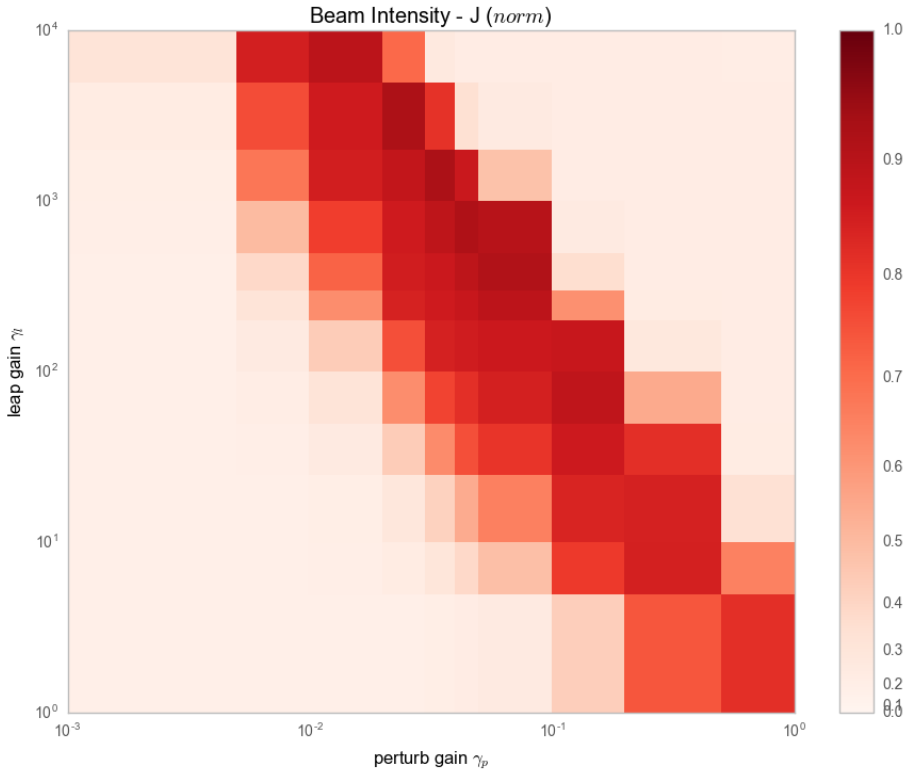


Figure 3.16: Monte Carlo simulation run for maximizing beam intensity of a 19-channel operation with varying perturb and leap gain parameters over a dynamically changing random noise added to each channel in each iteration.

3.3.2.2 Transient analysis

The transient response of the algorithm is tested with different noise conditions mentioned above to see how the operation of the algorithm with each iteration.

It is important to note that the y-axis of these plots represents the algorithm iterations and not the time values. The reason for this is to keep the algorithm speed dependent on the hardware engine iteration time. Therefore, having a faster hardware engine decreases the time between each iteration of the algorithm and makes it converge faster. For example, if the hardware engine is capable of running 100,000 iterations per second, the time between each iteration becomes 10 microseconds, thus 1000 iteration takes 10 milliseconds to run on the hardware engine. A 100 iteration sinusoidal noise applied to a channel becomes equivalent of a 1 kHz phase change rate. As an example to verify MC simulation results and comprehend how it translates to the actual beam quality, a parameter space is chosen from the first MC simulation where the beam intensity values are below 30%. Figures 3.17 and 3.18 show the algorithm operation with the chosen extreme parameters over 1000 iterations.

In Figure 3.17, the beam quality does not increase rapidly to converge to the theoretical maximum for 19-channels under ideal conditions over 1000 iterations which is not ideal for phase-locking operation since the on-field experiments would include dynamic distortions along the path. In the second Figure 3.18, the parameter leap gain, γ_l is too large for a stable operation and makes the phases jump around as can be seen in the third column of the plot. These two operating conditions verifies the selected parameters are not a desired option for phase-locking operation.

The next set of simulations are run with the parameter space selected according

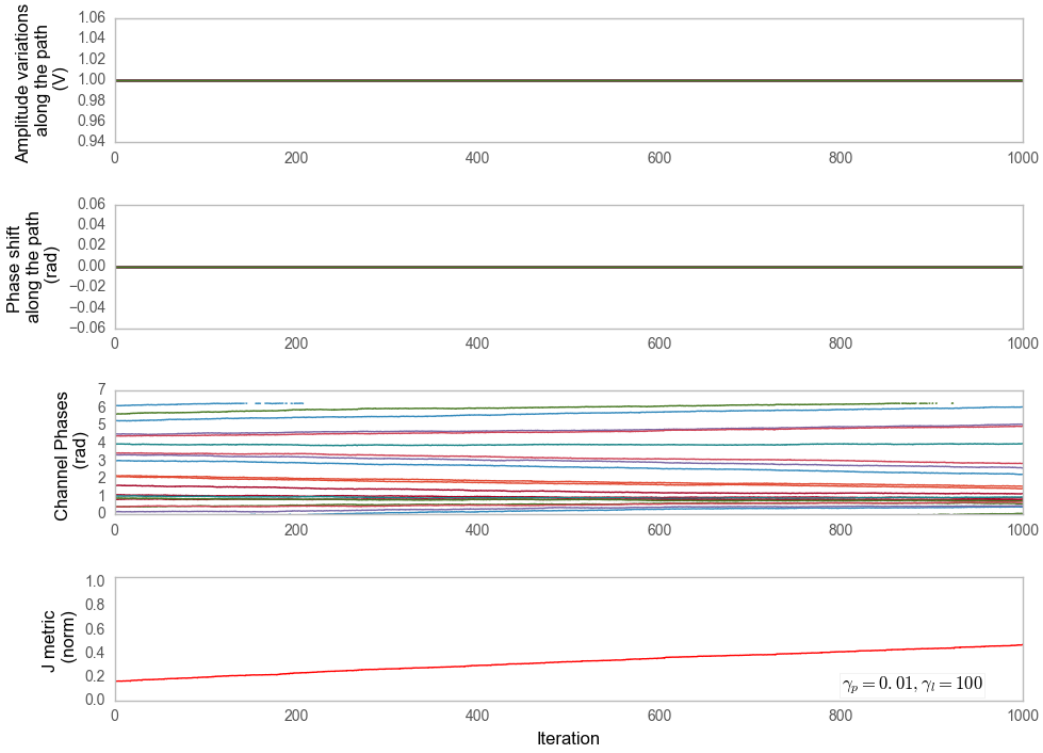


Figure 3.17: 1000 iteration SPGD operation using 19-channels with $\gamma_p = 0.01$ and $\gamma_l = 100$

to the MC simulation suggested areas. Figure 3.19 shows a 19-channel operation with $\gamma_p = 0.02$ and $\gamma_l = 1000$.

In the first part of Figure 3.19, the laser beams are assumed to travel under ideal conditions. It can be seen that at iteration 0 laser beam phases start at different values and the algorithm tries to maximize the feedback value by perturbing these phase values, and achieves a maximum around iteration 170 by having the same phase

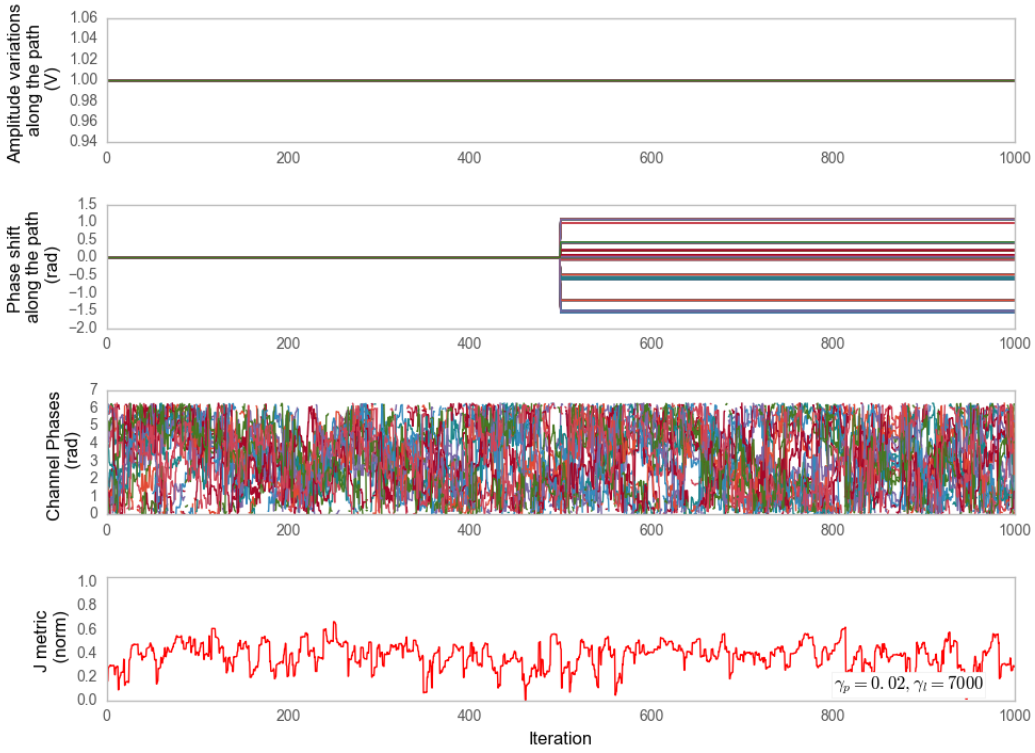


Figure 3.18: 1000 iteration SPGD operation using 19-channels with $\gamma_p = 0.02$ and $\gamma_l = 7000$

values for all the beams. When iteration is at 500, a uniform random phase distortions are applied along the path. These distortions cause the beam quality to drop from the maximum and the algorithm tries to compensate for the changes and finds the optimal value for each phase value to generate back the highest beam intensity in around 40 iterations.

Figure 3.20 shows a 19-channel operation with $\gamma_p = 0.02$ and $\gamma_l = 3000$. In

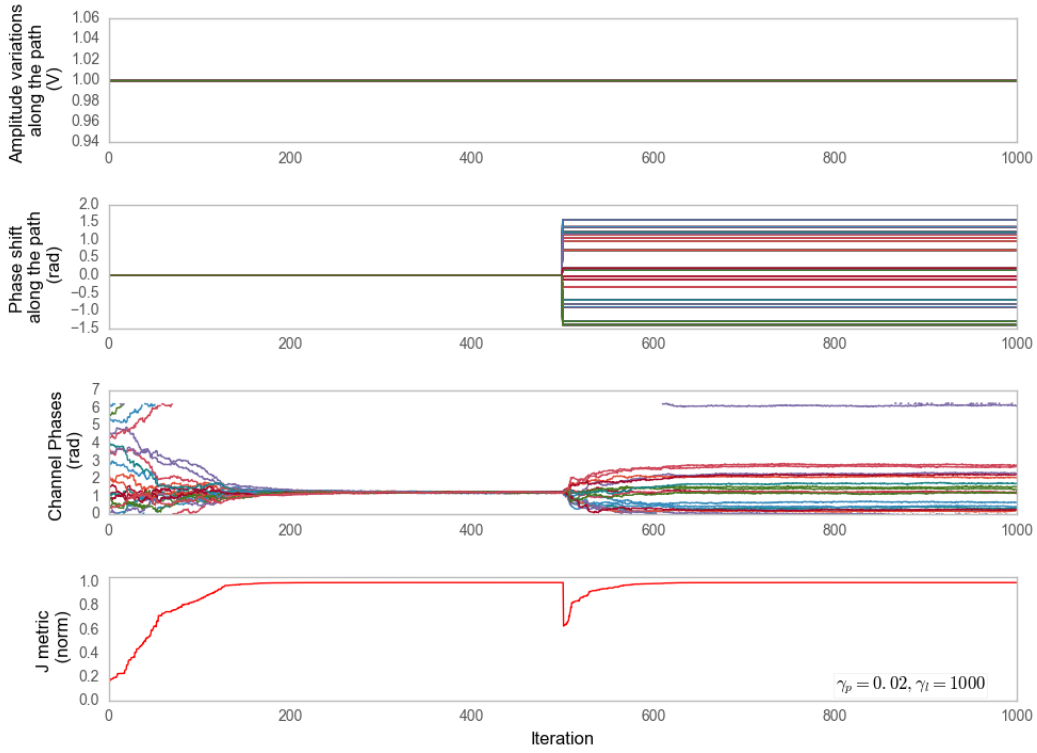


Figure 3.19: 1000 iteration SPGD operation using 19-channels with $\gamma_p = 0.02$ and $\gamma_l = 1000$ with random step noise added at $k = 500$

this simulation run, a sinusoidal phase noise as a propagation noise is inserted to all of the laser beams to represent a dynamic change over time. The sinusoidal phase noise completes its period in 100 iterations and is between $[\pi/4, \ -\pi/4]$. As can be seen from the last column of the graph, although the beam quality is not steady and keeps varying. The convergence speed of this run is around 40 iterations from 20% to 80% and the average beam quality after converging is 80% with a variance of 0.4%

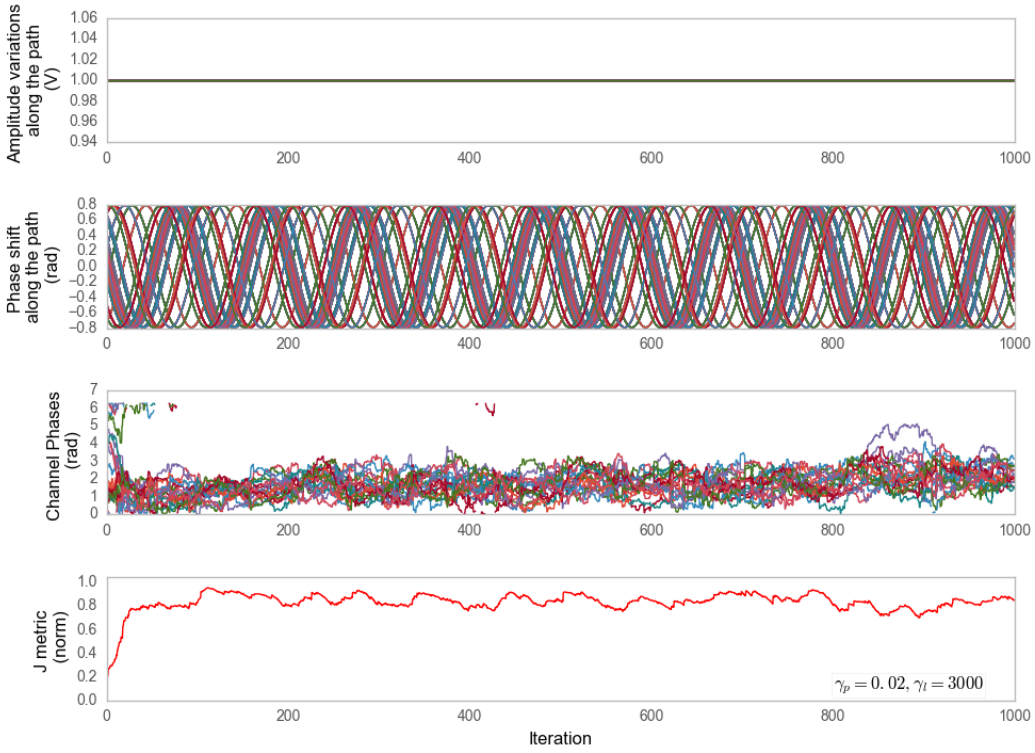


Figure 3.20: 1000 iteration SPGD operation using 19-channels with $\gamma_p = 0.02$ and $\gamma_l = 3000$ under sinusoidal phase noise

In the transient analysis shown in Figure 3.21, random noise values are added to the beam propagation path in each iteration to examine the beam quality. The random phase noise values are between $[\pi/4, \ -\pi/4]$. The convergence achieved within the first 70 iterations from 20% to 80% and the average beam quality after converging is 76% with a variance of 0.55%.

In the transient analysis shown in Figure 3.22, random phase noise conditions

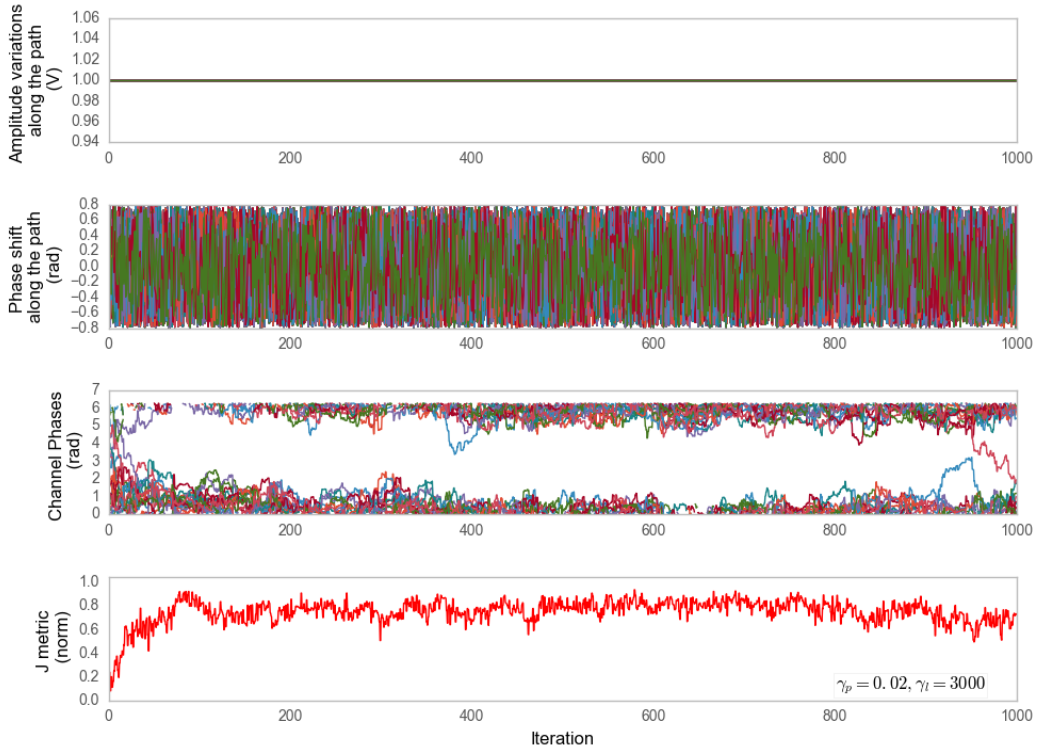


Figure 3.21: 1000 iteration SPGD operation using 19-channels with $\gamma_p = 0.02$ and $\gamma_l = 3000$ under random phase noise

along the propagation path are increased to values values are between $[\pi/2, \ -\pi/2]$ in order to represent more severe turbulence conditions. In this simulation run, it can be seen that the convergence is never achieved and the average beam quality stays at 38% average with a 1.7% variance.

At the end of these transient simulation runs, it can be stated that the transient analysis tool is very useful for visualizing and simulating the phase-locking operation

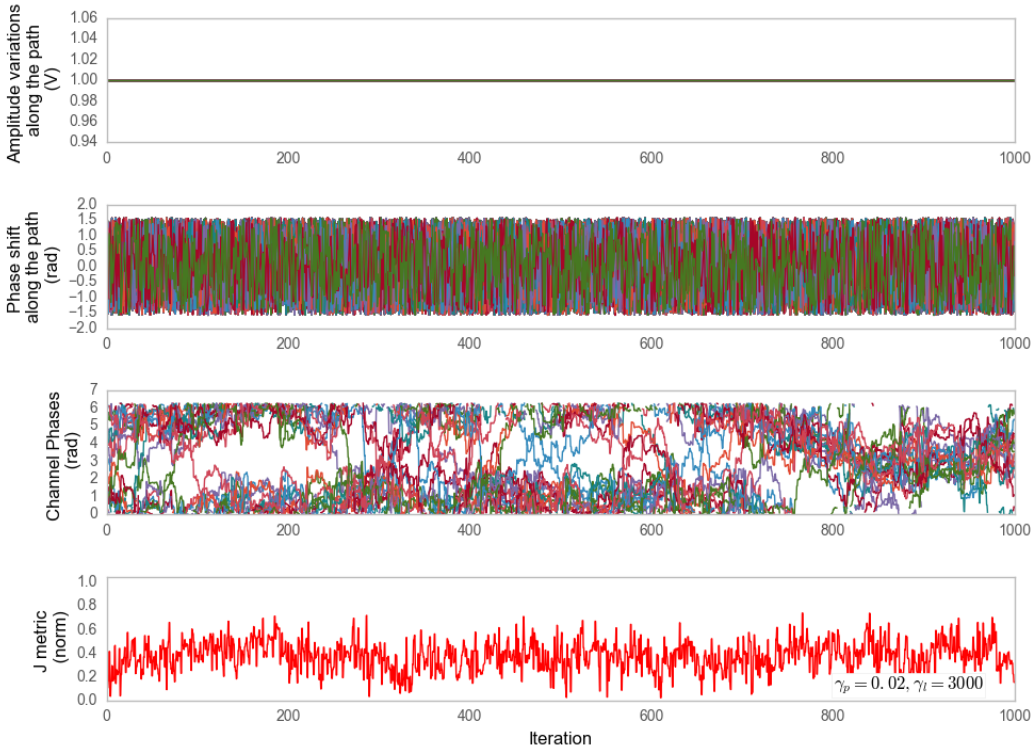


Figure 3.22: 1000 iteration SPGD operation using 19-channels with $\gamma_p = 0.02$ and $\gamma_l = 3000$ under random phase noise

in each iteration with varying distortion conditions. Additionally, high beam quality through dynamically changing turbulent conditions in the beam propagation path can be achieved unless the turbulence effects are very severe.

3.3.2.3 Convergence analysis

In the transient analysis above, the coefficients γ_p and γ_l have a significant impact on the convergence rate and general beam quality. In order to visualize the convergence rate differences based on these coefficients following set of convergence analysis are conducted with varying distortions.

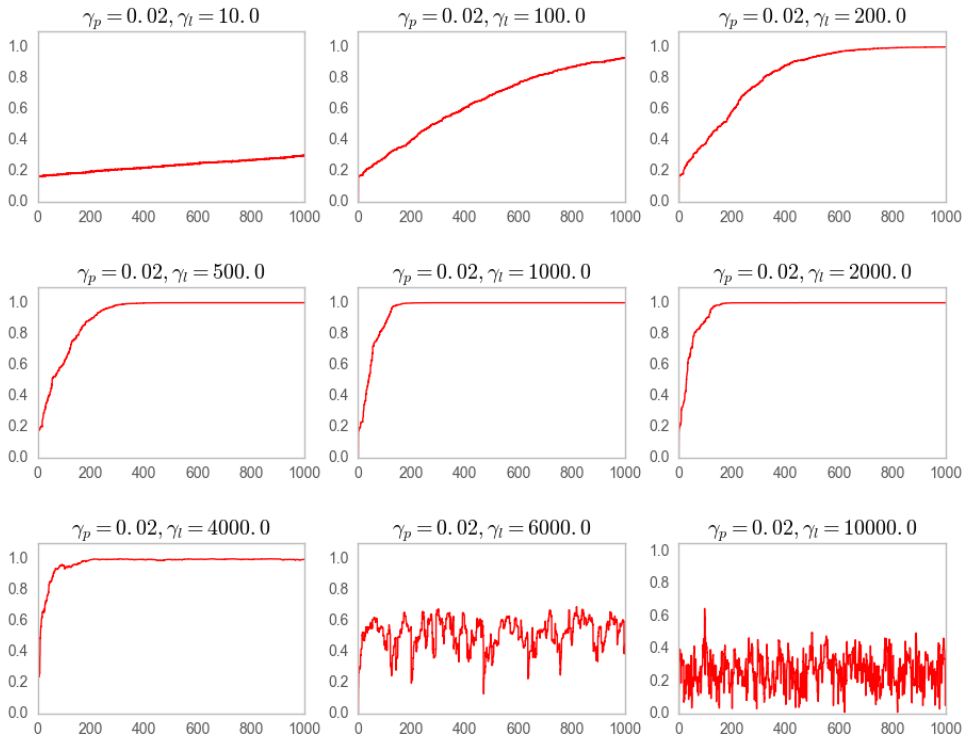


Figure 3.23: 1000 iteration SPGD operation using 19-channels with fixed $\gamma_p = 0.02$ and different γ_l values in ideal conditions with no noise.

Figure 3.23 shows the convergence rate of different parameter values under ideal

conditions with no noise added to the propagation path over 1000 iterations. The y-axis shows the beam intensity and x-axis represents the number of iterations. The first two graphs have a slow convergence due to γ_l value being too small whereas the last two graphs fail to phase-lock by losing the stability due to higher values of γ_l The middle graphs where $\gamma_l$value is above around 1000 up to 2000 show great promise with phase-locking under 200 iterations.

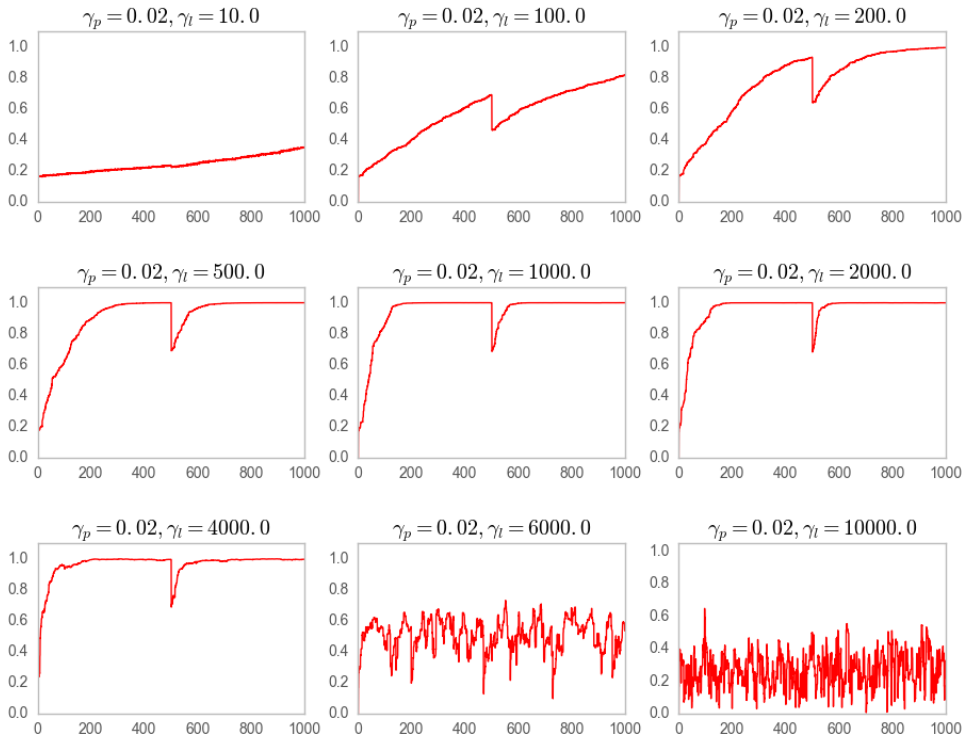


Figure 3.24: 1000 iteration SPGD operation using 19-channels with fixed $\gamma_p = 0.02$ and different γ_l values with step noise input.

Figure 3.24 shows the convergence rate of different parameter values with a step noise at mid point at the propagation path over 1000 iterations. The optimal conditions for running the algorithm appears to be around γ_l values between 1000 to 4000 since these values give the fastest convergence rate.

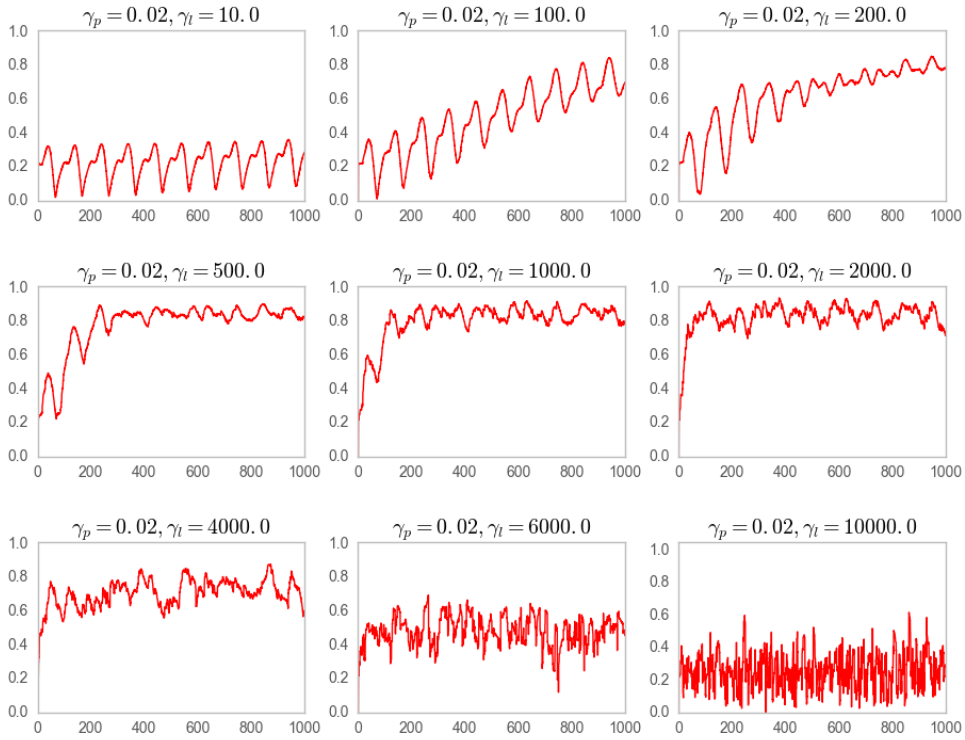


Figure 3.25: 1000 iteration SPGD operation using 19-channels with fixed $\gamma_p = 0.02$ and different γ_l values with sinusoidal noise added to all channels.

Figure 3.25 shows the convergence rate of different parameter values with sinusoidal noise $[\pi/4, \ -\pi/4]$ added to all the channels over 1000 iterations. It can be seen that γ_l value of around 2000 gives the best results amongst the others.

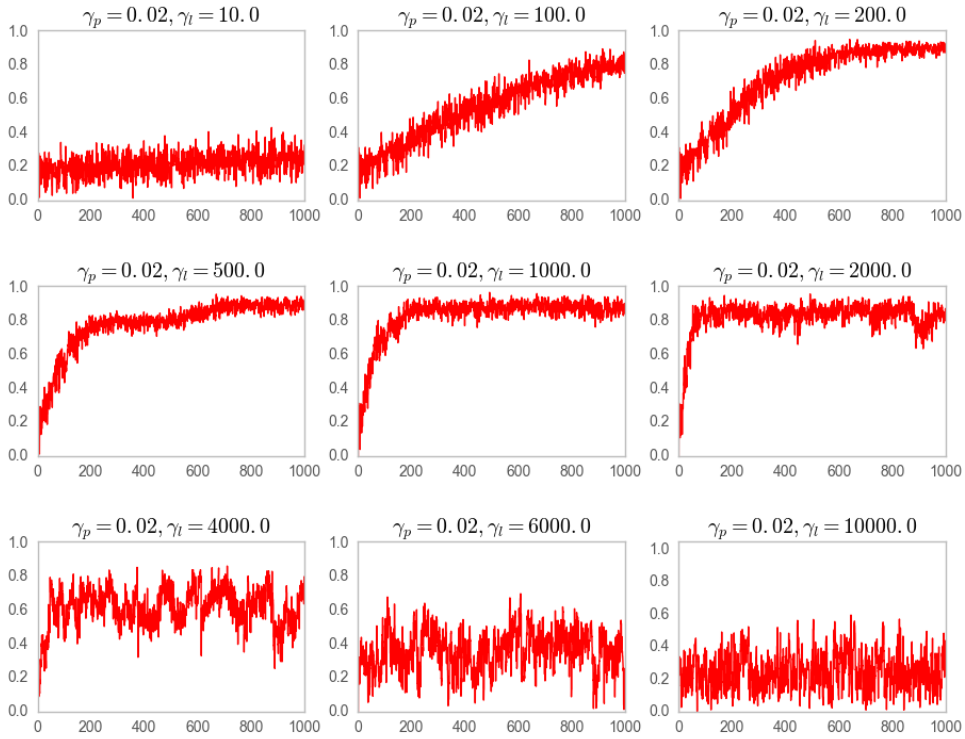


Figure 3.26: 1000 iteration SPGD operation using 19-channels with fixed $\gamma_p = 0.02$ and different γ_l values with random noise added to all channels.

Figure 3.26 shows the convergence rate of different parameter values with random noise values added to the beam propagation path. The middle graphs generate the best results where the γ_l values are between 500 to 2000.

3.3.3 Hardware experiments

Analysis and Simulation Framework shows the algorithm operation in detail and the algorithm behavior under different distortions through the laser beam propagation path. The presented simulations help with parameter optimization and finding optimal conditions. The implemented algorithm along with the optimized parameters can then be exported to the Hardware Engine for experimental operation under the actual hardware setup.

In order to test the Hardware Engine performance along with the implemented SPGD algorithm iteration speed, first a loopback module is built. Then, a capacitive load test is run to examine the amplifier output full range rise and fall times for driving the coaxial cables and phase shifters. Finally, the controller is attached to a 7-channel optical loopback setup consisting of a low-power laser source, 8-channel phase shifters and a photo-detector.

3.3.3.1 Hardware Engine performance with SPGD method

As mentioned earlier, the Hardware Engine relies on the floating-point DSP core to perform the performance-critical phase-locking operation. The DSP is clocked at 300 MHz, and has eight functional units each capable of executing one instruction per clock cycle.

One step of the SPGD algorithm for calculating the $\mathbf{U}$ vector output based on the J performance metric value is shown below:

1) Read J value from a register
2) Calculate $\delta\mathbf{U}$

3) Generate new $\delta\mathbf{U}$ values

4) Multiply γ, δJ, and $\delta\mathbf{U}$ and subtract from the previous $\mathbf{U}$

5) Write to $\mathbf{U}$ registers

Based on these steps, the total clock cycles needed to execute these operations is less than 450 cycles which brings the theoretical algorithm execution up to 680 kHz.

The calculated $\mathbf{U}$ vector values for each channel in the phased-array get transferred to the FPGA for loading multiple DACs that reside in the Scatter Interface. The total delay of this operation consists of two main delays: (I) DSP to FPGA transfer delay t_{DSPtoF} and (II) FPGA to DAC transfer delay t_{FtoDAC}.

The DSP is connected to the FPGA using the uPP bus that is 16-bit wide and runs at 75 MHz clock speed. Total DSP to FPGA transfer delay, t_{DSPtoF}, for the new control channel values of the phased-array can be calculated using the following formula:

$$t_{DSPtoF} = \frac{Total\ number\ of\ bits}{Number\ of\ bits\ per\ clock\ cycle} \times 1\ clock\ period \tag{3.6}$$

Substituting the values gives:

$$\begin{aligned} t_{DSPtoF} &= \frac{N \times 16\ bits}{16\ bits\ per\ clock\ cycle} \times \frac{1}{75\ MHz} \\ &= 0.0133 \times N\ \mu s \end{aligned} \tag{3.7}$$

where N is the number of control channels.

Upon receiving the U vectors from DSP, the FPGA places them into shift registers ready to transfer to DACs.

The FPGA is connected to the DACs using Serial Peripheral Interface (SPI) that consists of a single data line that runs at 37.5 MHz for achieving global synchronization. The dual-channel DACs require a total of 36-bits to operate (16-bits per DAC and control signals in between). The total FPGA to DAC transfer delay, t_{FtoDAC}, can be calculated as:

$$\begin{aligned} t_{FtoDAC} &= 36 \times 1 \; clock \; period = 36 \times \frac{1}{37.5 \; MHz} \\ &= 0.96 \; \mu s \end{aligned} \tag{3.8}$$

The total from DSP to DACs delay $t_{DSPtoDAC}$ then becomes:

$$t_{DSPtoDAC} = t_{DSPtoF} + t_{FtoDAC} = 0.96 + 0.0133 \times N \; \mu s \tag{3.9}$$

Likewise, the return delay using 14-bit J metric value from ADC to DSP delay $t_{ADCtoDSP}$ can be calculated as:

$$\begin{aligned} t_{ADCtoDSP} &= t_{ADCtoF} + t_{FtoDSP} \\ &= 14 \times \frac{1}{37.5 \; MHz} + 1 \times \frac{1}{75 \; MHz} \\ &= 0.386 \; \mu s \end{aligned} \tag{3.10}$$

For a 19-channel operation ($for\ N = 19$), the total delay t_{delay} is:

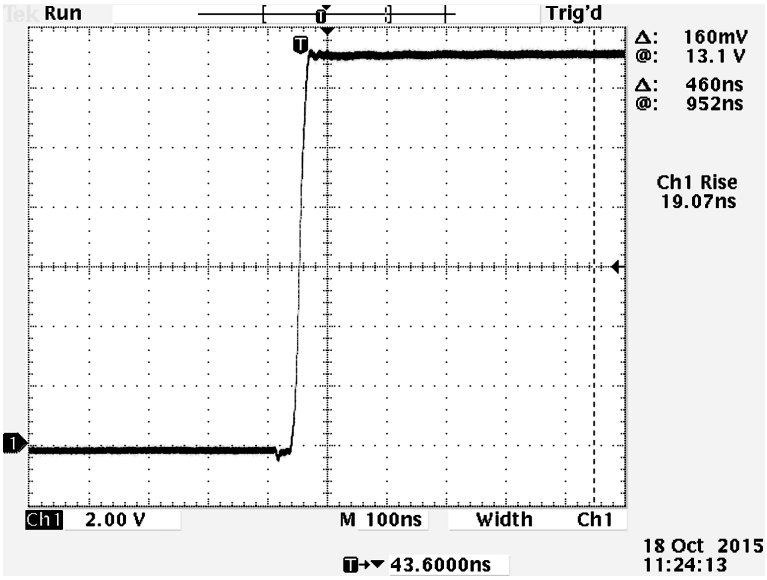
$$\begin{aligned} t_{delay} &= t_{DSPtoDAC} + t_{ADCtoDSP} \\ &= 0.96 + 0.0133 \times 19 + 0.386 \\ &= 1.6\ \mu s \end{aligned} \tag{3.11}$$

Under a sequential execution of the algorithm, the processing time and the total delay gets added to find the theoretical maximum operation speed of the algorithm. However, with careful planning, the algorithm execution and the data transfer times can be parallelized to achieve faster speeds. Thus, the final algorithm run speed becomes:

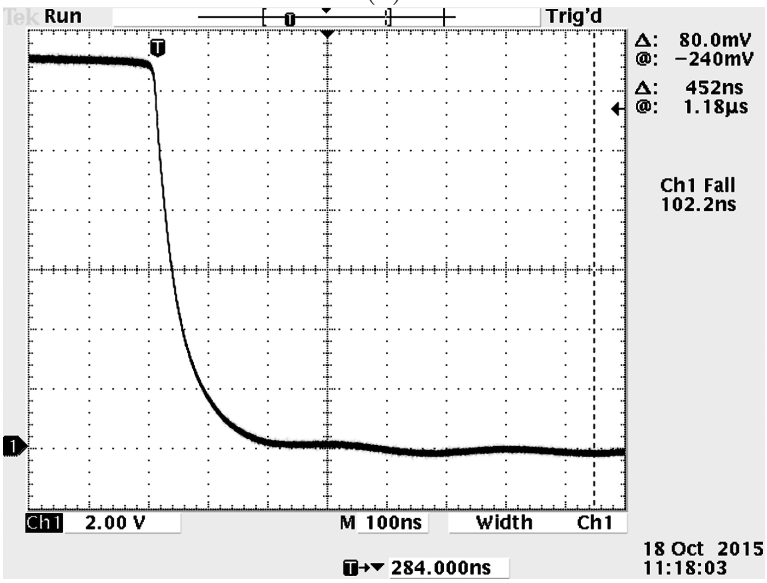
$$\begin{aligned} t_{SPGD} &= \frac{1}{\max(t_{proc}, t_{delay})} \\ &= 650,000\ steps\ per\ second \end{aligned} \tag{3.12}$$

3.3.4 Capacitive load test of the amplifiers

The circuit model and the rise & fall times for the amplifier modules were given in Section 3.2.2.2. The implemented modules are tested with a 4-ft coax cable attached to the amplifier output and the rise and fall times are measured. Figure 3.27 shows the measured rise and fall times to be 20 ns and 100 ns respectively which is close to the theoretical model.



(a)



(b)

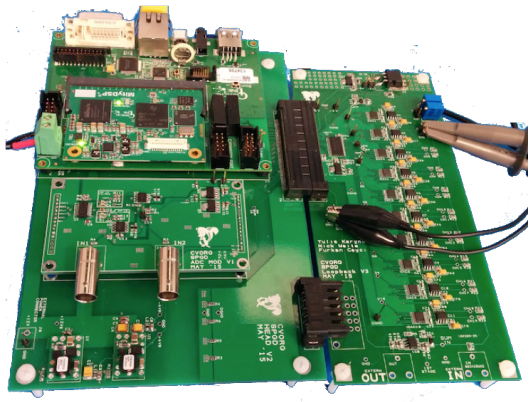
Figure 3.27: (a) Rise time of the amplifier, and (b) Fall time of the amplifier over full voltage swing

3.3.5 19-channel electrical loopback operation

The implemented processing back end does not include any analog outputs and depend on additional digital to analog converter modules to operate the phase shifters. For testing the phase-locking operation before interfacing directly to the phase shifters, a 19-channel electrical loopback module is developed according to the following equation:

$$J = \sum_{k} \frac{U_k}{19} + \eta \tag{3.13}$$

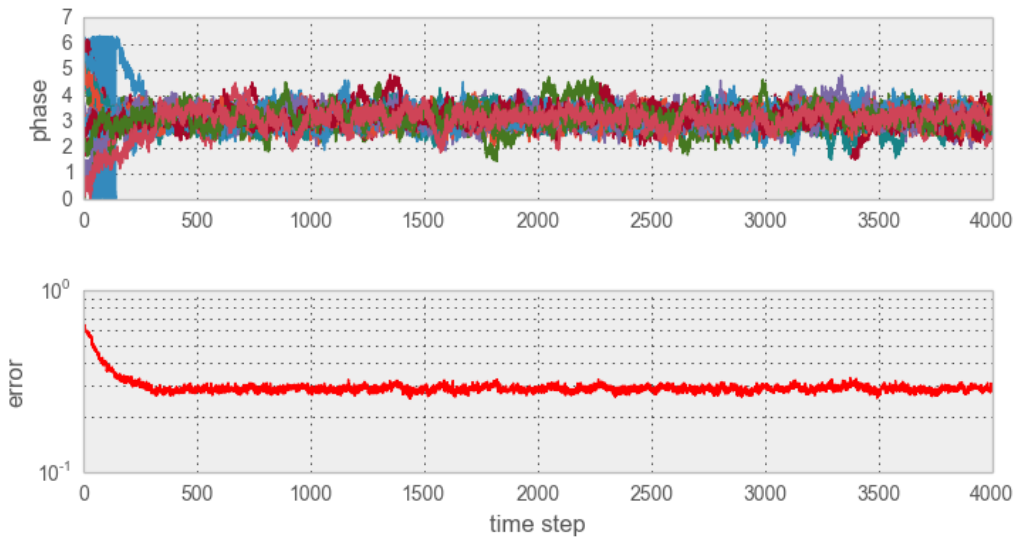
where k is the channel number, η is the random noise source, U is the output vector for the phase values. This module is also used for fine tuning the algorithm and establishing the interoperability functionality with the simulation framework. Figure 3.28 shows this electrical loopback module attached to the Processing Back end.



(a)

Figure 3.28: 19-channel electrical loopback module attached to the Processing Back end

The module consists of 10, 2-channel DACs with 2-channel amplifiers to generate the analog voltages. These voltages are then fed into the summing amplifier to generate the average voltage value of the U vector that would be used as the J metric for the system.



(a)

Figure 3.29: SPGD operation on 19-channel electrical loopback setup. The error is a log plot and presented as $1 - J_{norm}$

Figure 3.29 shows 4000 iteration of the SPGD algorithm running on this setup for phase-locking. The top graph shows the voltage values for each channel and the bottom graph is a log plot of the $1 - J_{norm}$ value for evaluating the operation in detail.

The coefficients for the operation are $\gamma_p = 0.02$ and $\gamma_l = 1000$ which are used based on the simulation framework feedback. The phase values are arbitrarily assigned to voltage values to emulate the phase shifter behavior. $(0 \rightarrow 0V,\ \pi \rightarrow 2V,\ 2\pi \rightarrow 0V)$. The maximum value of J is achieved when all the phases are at π where it maximizes the Equation 3.13 in 300 iterations with the given coefficient values.

3.3.6 7-channel optical loopback operation

Finally the controller platform is attached to an optical setup which consists of a low-power laser source, a 1x8 fiber laser splitter with phase-shifting capability, and a photo-detector to get the beam quality. Figure 3.30 shows the full setup.

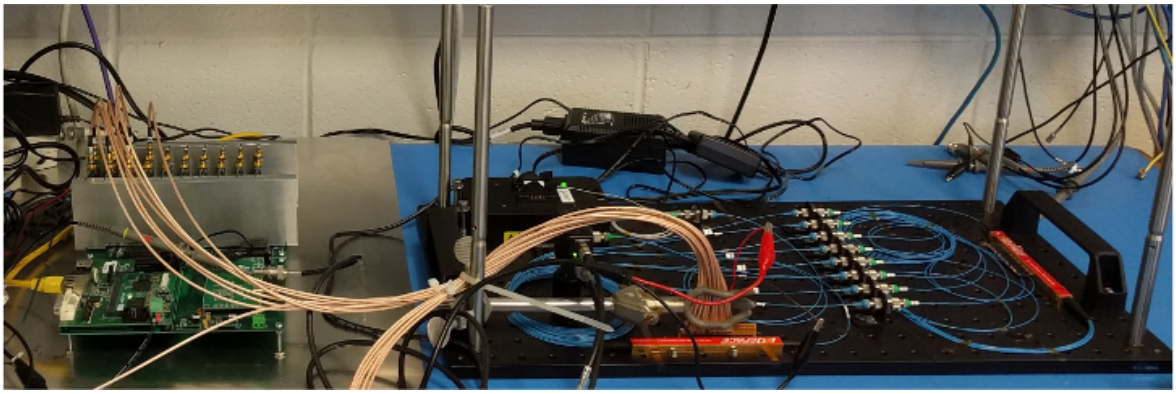


Figure 3.30: 7-channel optical testbed attached to the controller platform

The 1x8 fiber laser splitter requires voltages to control the phase values of channels. When the laser beams go through these crystals they undergo phase shifts due to the refractive index changes with the applied electric field [48]. In order to control the phases, voltages are mapped to the phase values. Full phase swing $(0 - 2\pi)$ requires

around 4.2 Volts with small variances between the channels. If the mapping is not properly done, it creates phase jumps when the phases wrap around, and the jump values depend on how imperfect the mapping is.

The first experimental run shown in Figure 3.31 is done using an outliner coefficients from the simulation framework to evaluate the system behavior with unoptimized coefficient values. The values chosen are $\gamma_p = 0.04$ and $\gamma_l = 8000$. The first plot in the figure shows the phases of each channel constantly jumping around due to high γ_l value. The second plot in the figure shows the J metric value normalized over the full range of the ADC converter, and it shows the unstable behavior of the system with the given parameters.

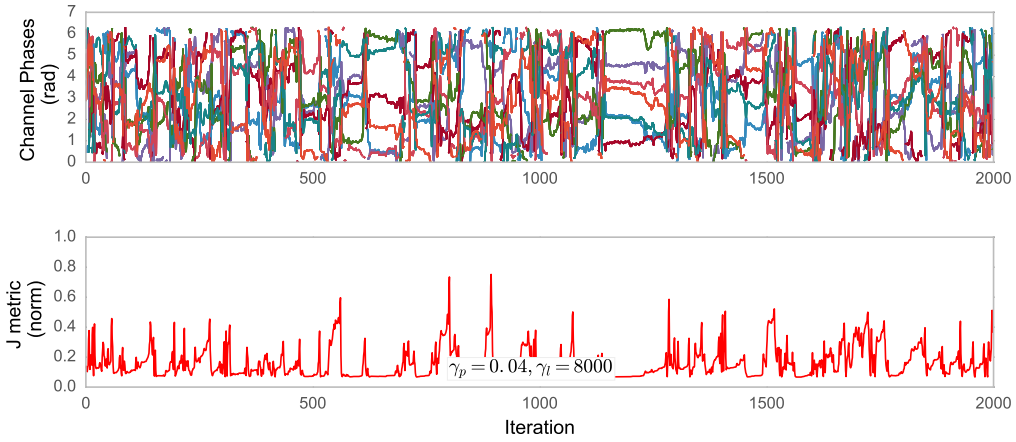


Figure 3.31: 2000 iteration SPGD operation using 7-channel optical setup with $\gamma_p = 0.04$ and $\gamma_l = 8000$

The second experiment presented in Figure 3.32 shows the phase-locking operation with coefficients $\gamma_p = 0.01$ and $\gamma_l = 1500$. The convergence speed is not as fast due to the small γ_p value. It takes around 800 iterations to get close to the maximum value of the J metric by aligning the phases together. The small jumps in iteration around 100 and 600 are due to the wrapping of the phases with imperfect voltage mapping for a single channel that is explained above.

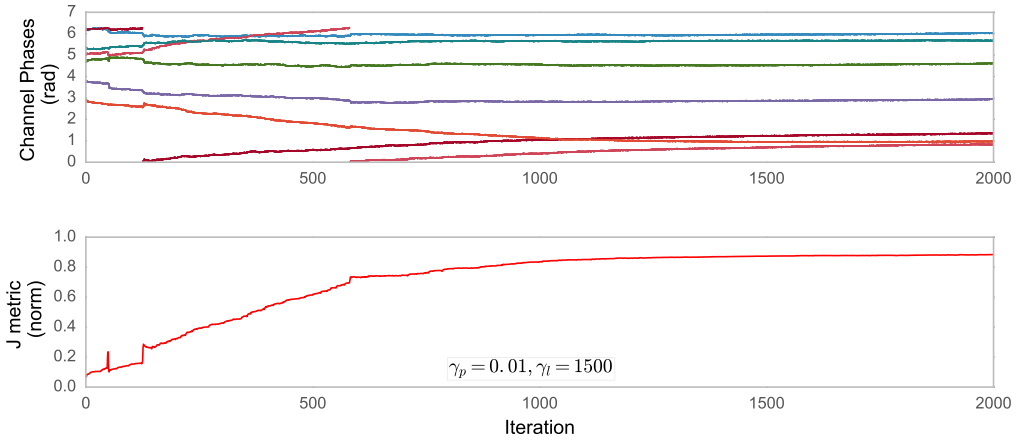


Figure 3.32: 2000 iteration SPGD operation using 7-channel optical setup with $\gamma_p = 0.01$ and $\gamma_l = 1500$

The third experiment shown in Figure 3.33 uses the coefficient values of $\gamma_p = 0.025$ and $\gamma_l = 1500$. These values are the optimized values from the simulation framework. The convergence speed in this run is greatly improved, and phases lock at around 250 iterations into the algorithm.

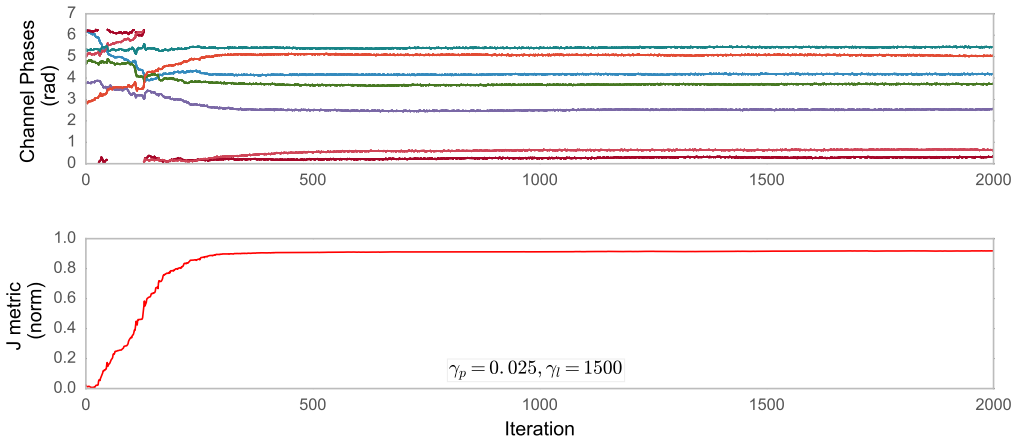


Figure 3.33: 2000 iteration SPGD operation using 7-channel optical setup with $\gamma_p = 0.025$ and $\gamma_l = 1500$

The forth and final experiment presented in Figure 3.34 shows the phase-locking operation with coefficients $\gamma_p = 0.03$ and $\gamma_l = 1500$. In this experimental run, the convergence successfully happens in less than 200 iterations which takes around 300 microseconds for the controller that runs at 650,000 iterations per second.

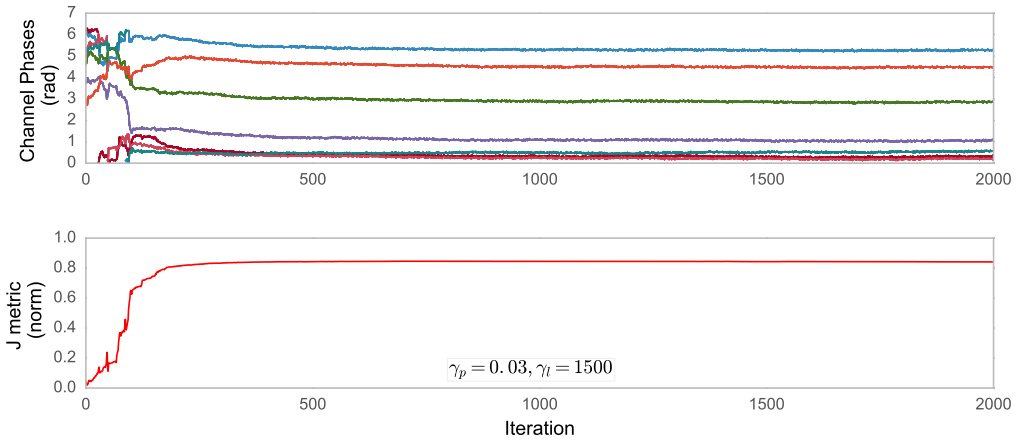


Figure 3.34: 2000 iteration SPGD operation using 7-channel optical setup with $\gamma_p = 0.03$ and $\gamma_l = 1500$

3.4 Conclusions

In conclusion, the methodology for the design of an innovative multi-channel controller platform is presented. The controller platform is fitted for a phase coherent fiber laser array needs. In order to develop algorithms for phase-locking or beam combining, and simulate and analyze algorithm performance under various distortion scenarios, a novel Analysis and Simulation Framework is developed. The developed framework is explained and as a case study, a multi-channel optimization algorithm called Stochastic Parallel Gradient Descent (SPGD) is implemented and it's performance is evaluated. The framework's visualization and analysis tools are used to optimize the algorithm coefficients under different beam propagation distortions.

Final boxed hardware ready for on-field deployment is shown in Figure 3.35

(a)

(b)

Figure 3.35: Final controller box

For field experiments, and controlling the phased-arrays, an innovative hardware engine is build with a powerful processing back end that is optimized for fast

multi-channel calculations and control. The implemented Hardware Engine is tied to the framework for a seamless algorithm development and testing operation. The implemented SPGD algorithm is then run on the Hardware Engine to demonstrate the phase-locking operation with different setups. The final experimental setup uses an optical testbed that is part of a phased-array. The platform operation for phase-locking is successfully demonstrated using the final optical testbed setup using multiple experimental runs with different coefficients. By parallelizing the algorithm runtime and the hardware latency, the controller platform can achieve high channel update speeds reaching up to 650,000 updates per second. Due to the scalable nature of the architectural design, the number of channels have minimal impact on the controller speeds.

Chapter 4

SUMMARY AND FUTURE WORK

4.1 Summary

In summary, two unique platforms are presented for mitigating atmospheric turbulence effects in long-range beam projection and imaging applications.

The first platform presented in Chapter 2 is a real-time imaging platform for long-range applications. The combination of hardware acceleration with specially designed platform-independent framework for turbulence mitigation enables real-time operation for on-field experiments. The methodology of the platform design is presented and the performance capabilities is given. As a case study, a modified version of an effective atmospheric turbulence mitigation algorithm called Lucky Region Fusion (LRF) that depends on capturing many short-exposure images is implemented. The operating speed of the algorithm is calculated to be 259 FPS for 4-channel 1024 x 1024 resolution image using direct memory transfer and 154 FPS by sequential memory copy.

The second platform presented in Chapter 3 is a controller platform to operate fiber laser phased-arrays for phase-locking through turbulent conditions. The developed platform consists of two unique parts. First, a novel analysis and simulation software framework is presented for developing algorithms for multi-channel control operations.

Second, an innovative and modular hardware design enables for easy scalability and upgrade of the platform. The developed platform is tested with a multi-channel optimization algorithm called Stochastic Parallel Gradient Descent (SPGD) method and simulation and hardware based results are given. The presented platform performance reaches up to 650,000 update speed for a 19-channel phased-array system. The novel architectural decisions enable easy scalability of the controller platform with little speed penalty. (13.3 ns per added channel.)

4.2 Future Work

The designed platforms are built for real-time operation under turbulent conditions for long-range applications, and the developed frameworks and designed hardware systems have many potential general-purpose uses where efficient multi-channel operations are required such as adaptive optics, and real-time imaging is required such as microscopy.

For the future work, these two platforms can be combined for a full scale laser beam projection system. The real-time imaging platform can be used as the far field metric detector for supplying the performance metric value with algorithm modifications and special front-ends while the controller platform controls the fiber laser phased-array for beam combining operation based on the received metric.

Additionally, the analysis and simulation framework that is developed for controller platform can be expanded for a general optimization tool for a given hardware system.

BIBLIOGRAPHY

[1] Edgar L Andreas. The refractive index structure parameter , Cn, for a year over the frozen Beaufort Sea. 24(5):667–679, 1989.

[2] Larry C Andrews and Ronald L Phillips. *Laser beam propagation through random media*, volume 52. SPIE press Bellingham, WA, 2005.

[3] J Roger P Angel. Ground-based imaging of extrasolar planets using adaptive optics. *Nature*, 368(6468):203–207, 1994.

[4] Mathieu Aubailly, Mikhail a. Vorontsov, Gary W. Carhart, and Michael T. Valley. Automated video enhancement from a stream of atmosphericallydistorted images: the lucky-region fusion approach. 7463:74630C–74630C–10, 2009.

[5] Andrea L. Bertozzi, Stefano Soatto, Sung Ha Kang, and Yifei Lou. Video stabilization of atmospheric turbulence distortion. *Inverse Problems and Imaging*, 7(3):839–861, sep 2013.

[6] Eric Bogatin. *Signal integrity: simplified*. Prentice Hall Professional, 2004.

[7] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer vision with the OpenCV library*. " O'Reilly Media, Inc.", 2008.

[8] Tyler Browning, Christopher Jackson, Furkan Cayci, Gary W. Carhart, J. J. Liu, and Fouad Kiamilev. Hardware acceleration of lucky-region fusion (LRF) algorithm for high-performance real-time video processing. page 94512G, jun 2015.

[9] Casey Campbell, Benjamin Mazur, Furkan Cayci, Nicholas Waite, Fouad Kiamilev, and Jony J Liu. Algorithm development, optimization, and simulation framework for a phase-locking fiber laser array. In *SPIE Defense+ Security*, pages 98460C–98460C. International Society for Optics and Photonics, 2016.

[10] V. M. Canuto and Gregory J. Hartke. Propagation of electromagnetic waves in a turbulent medium. *Journal of the Optical Society of America A*, 3(6):808, jun 1986.

[11] Furkan Cayci and Nicholas Waite. Modular Adaptive Phased-locked Fiber Array Controller Platform. In *IEEE Aerospace Conference Proceedings*, pages 1–6, 2016.

[12] P E Ciddor. Refractive index of air: new equations for the visible and near infrared. *Applied optics*, 35(9):1566–1573, 1996.

[13] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[14] Arnaud Doucet, Nando De Freitas, and Neil Gordon. An introduction to sequential monte carlo methods. In *Sequential Monte Carlo methods in practice*, pages 3–14. Springer, 2001.

[15] Khaled S M Essa and M Embaby. Atmospheric Turbulent Fluxes of Heat and Momentum. *Journal of Nuclear and Radiation Physics*, 2(2):111–121, 2007.

[16] T. Y. Fan. Laser beam combining for high-power, high-radiance sources. *IEEE Journal on Selected Topics in Quantum Electronics*, 11(3):567–577, 2005.

[17] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. A comprehensive performance comparison of cuda and opencl. In *2011 International Conference on Parallel Processing*, pages 216–225. IEEE, 2011.

[18] Barak Fishbain, Leonid P Yaroslavsky, and Ianir A Ideses. Real-time stabilization of long range observation system turbulent video. *Journal of Real-Time Image Processing*, 2(1):11–22, oct 2007.

[19] D.H. Frakes, J.W. Monaco, and M.J.T. Smith. Suppression of atmospheric turbulence in video using an adaptive control grid interpolation approach. *2001 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings (Cat. No.01CH37221)*, 3:1881–1884, 2001.

[20] D.L. Fried. Probability of getting a lucky short-exposure image through turbulence. *Optical Society of America*, 68(May 1977):1651–1658, 1978.

[21] Chao Geng, Wen Luo, Yi Tan, Hongmei Liu, Jinbo Mu, and Xinyang Li. Experimental demonstration of using divergence cost-function in SPGD algorithm for coherent beam combining with tip/tilt control. *Optics express*, 21(21):25045–55, 2013.

[22] Gregory D. Goodno and S. Benjamin Weiss. Automated co-alignment of coherent fiber laser arrays via active phase-locking. *Optics Express*, 20(14):14945, 2012.

[23] Christopher R Jackson, Garrett A Ejzak, Mathieu Aubailly, Gary W Carhart, and J Jiang Liu. Hardware acceleration of lucky-region fusion (LRF) algorithm for imaging. 2014.

[24] Brian W Kernighan and Dennis M Ritchie. The c programming language. 2006.

[25] Andrei N Kolmogorov. The local structure of turbulence in incompressible viscous fluid for very large reynolds numbers. In *Dokl. Akad. Nauk SSSR*, volume 30, pages 301–305. JSTOR, 1941.

[26] Andrey Nikolaevich Kolmogorov. Dissipation of energy in locally isotropic turbulence. In *Dokl. Akad. Nauk SSSR*, volume 32, pages 16–18. JSTOR, 1941.

[27] N. M. Law, C. D. Mackay, and J. E. Baldwin. Lucky imaging: high angular resolution imaging in the visible from the ground\n. *Astronomy and Astrophysics*, 446:739–745, 2006.

[28] Ling Liu, Dimitrios N. Loizos, Mikhail A. Vorontsov, Paul P. Sotiriadis, and Gert Cauwenberghs. Coherent combining of multiple beams with multi-dithering technique: 100KHz closed-loop compensation demonstration. *Proceedings of SPIE*, pages 67080D–67080D–9, 2007.

[29] William Maignan, David Koeplinger, Gary W. Carhart, Mathieu Aubailly, Fouad Kiamilev, and J. Jiang Liu. Hardware acceleration of lucky-region fusion (LRF) algorithm for image acquisition and processing. 8720:87200B, 2013.

[30] Yu Mao and Jérôme Gilles. Non rigid geometric distortions correction - Application to atmospheric turbulence stabilization. *Inverse Problems and Imaging*, 6(3):531–546, sep 2012.

[31] Mario Micheli, Yifei Lou, Stefano Soatto, and Andrea L. Bertozzi. A Linear Systems Approach to Imaging Through Turbulence. *Journal of Mathematical Imaging and Vision*, 48(1):185–201, jan 2014.

[32] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, 2008.

[33] Alejandro Oscoz, Rafael Rebolo, Roberto López, Antonio Pérez-Garrido, Jorge Andrés Pérez, Sergi Hildebrandt, Luis Fernando Rodríguez, Juan José Piqueras, Isidro Villó, José Miguel González, et al. Fastcam: a new lucky imaging instrument for medium-sized telescopes. In *SPIE Astronomical Telescopes+ Instrumentation*, pages 701447–701447. International Society for Optics and Photonics, 2008.

[34] Francois Roddier. *Adaptive optics in astronomy*. Cambridge university press, 1999.

[35] T M Shay, Vincent Benham, J T Baker, Benjamin Ward, Anthony D Sanchez, Mark A Culpepper, D Pilkington, Justin Spring, Douglas J Nelson, and Chunte A Lu. First experimental demonstration of self-synchronous phase locking of an optical array. *Optics Express*, 14(25):12015–12021, 2006.

[36] Dave Shreiner, Bill The Khronos OpenGL ARB Working Group, et al. *OpenGL programming guide: the official guide to learning OpenGL, versions 3.0 and 3.1*. Pearson Education, 2009.

[37] Andrew Smith, Jeremy Bailey, J. H. Hough, and Steven Lee. An investigation of lucky imaging techniques. *Monthly Notices of the Royal Astronomical Society*, 398(4):2069–2073, 2009.

[38] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(1-3):66–73, 2010.

[39] Arnold Tunick, Nikolay Tikhonov, Mikhail Vorontsov, and Gary Carhart. Characterization of optical turbulence (cn2) data measured at the arl a_lot facility. *Report, ARL-MR-625, ARL, Adelphi, MD*, 2005.

[40] Robert K Tyson. *Principles of adaptive optics*. CRC press, 2015.

[41] Guido Van Rossum et al. Python programming language. In *USENIX Annual Technical Conference*, volume 41, 2007.

[42] M. A. Vorontsov, G. W. Carhart, and J. C. Ricklin. Adaptive phase-distortion correction based on parallel gradient-descent optimization. *Optics Letters*, 22(12):907–909, 1997.

[43] M. a. Vorontsov and V. P. Sivokon. Stochastic parallel-gradient-descent technique for high-resolution wave-front phase-distortion correction. *Journal of the Optical Society of America A*, 15(10):2745, 1998.

[44] Xiaolin Wang, Pu Zhou, Yanxing Ma, Jingyong Leng, Xiaojun Xu, and Zejin Liu. Active phasing a nine-element 1.14 kW all-fiber two-tone MOPA array using SPGD algorithm. *Optics letters*, 36(16):3121–3, 2011.

[45] Xiong Wang, Xiao-Lin Wang, Pu Zhou, Rong-Tao Su, Chao Geng, Xin-Yang Li, Xiao-Jun Xu, and Bo-Hong Shu. Coherent beam combination of adaptive fiber laser array with tilt-tip and phase-locking control. *Chinese Physics B*, 22(2):024206, 2013.

[46] Thomas Weyrauch, Mikhail a. Vorontsov, Gary W. Carhart, Leonid a. Beresnev, Andrey P. Rostov, Ernst E. Polnau, and Jony Jiang Liu. Experimental demonstration of coherent beam combining over a 7 km propagation path. *Optics Letters*, 36(22):4455, 2011.

[47] Thomas Weyrauch, Mikhail a Vorontsov, Joseph Mangano, Vladimir Ovchinnikov, David Bricker, Ernst Polnau, and Andrey Rostov. Deep turbulence effects mitigation with coherent combining of 21 laser beams over 7 km. *Optics Letters*, 41(4):840, 2016.

[48] Amnon Yariv and Pochi Yeh. *Optical waves in crystals*, volume 10.

[49] C X Yu, S J Augst, S M Redmond, K C Goldizen, D V Murphy, a Sanchez, and T Y Fan. Coherent combining of a 4 kW, eight-element fiber amplifier array. *Optics letters*, 36(14):2686–2688, 2011.

[50] Pu Zhou, Zejin Liu, Xiaolin Wang, Yanxing Ma, Haotong Ma, Xiaojun Xu, and Shaofeng Guo. Coherent Beam Combining of Fiber Amplifiers Using Stochastic Parallel Gradient Descent Algorithm and Its Application. *IEEE Journal of Selected Topics in Quantum Electronics*, 15(2):248–256, 2009.