EFFICIENT COMPUTING FOR THREE-DIMENSIONAL QUANTITATIVE PHASE IMAGING

A Dissertation
Presented to
The Academic Faculty

by

Ji Ye Chun

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
August 2021

EFFICIENT COMPUTING FOR THREE-DIMENSIONAL QUANTITATIVE PHASE IMAGING

Approved by:

Dr. Thomas K. Gaylord, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Shyh-Chiang Shen
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Christopher J. Rozell
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: July 28, 2021

ACKNOWLEDGEMENTS

The completion of this thesis would not have been possible without the support and guidance of many people along the way. First, I would like to express my sincere gratitude to my advisor, Dr. Thomas K. Gaylord, for his guidance and advice throughout my undergraduate and graduate career at Georgia Tech. His patience, passion for teaching, and endless research endeavor deserve great appreciation. I would like to thank my thesis committee members, Dr. Shyh-Chiang Shen and Dr. Christopher J. Rozell, for their time and advice on my thesis. I would also like to thank the staff at the school of electrical and computer engineering, including my academic advisor, Tasha M. Torrence, for their hard work and dedication to the excellent ECE programs.

I would like to thank Dr. Micah H. Jenkins, who did a pioneer work for this research, and Dr. Yijun Bao, who helped me shape my understanding of previous research. I want to thank Joshua Long, with whom I had great pleasure working together. I am also grateful to my fellow graduate student, Pranav P. Kulkarni, for his support and encouragement during times of struggle in research.

Finally, I would like to thank my family. My parents have raised me to dream big and sacrificed a lot for me to pursue my dreams. I cannot thank them enough for that. I would also like to thank my American family, Bill and Jan Toner, for their generous support and love. I am fortunate to have them in my life. I am also grateful to have an adorable canine companion, Lucky Charm, who kept me physically and mentally healthy during the quarantine in 2020. Lucky reminds me to smile and cherish every moment in life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

2D	Two-Dimensional
3D	Three-Dimensional
ACML	AMD Core Math Library
ADMM	Alternating Direction Method of Multipliers
ALU	Arithmetic Logic Unit
AOFT	Absorption Optical Transfer Function
API	Application Programming Interface
BLAS	Basic Linear Algebra Subprograms
CPU	Central Processing Unit
CT	Computed Tomography
CUDA	Compute Unified Device Architecture
DDR	Double Data Rate
DFT	Discrete Fourier Transform
DHM	Digital Holographic Microscopy
DIC	Differential Interference Contrast
DRAM	Dynamic Random Access Memory
FBG	Fiber Bragg Grating
FFT	Fast Fourier Transform
FFTW	Fastest Fourier Transform in the West
FPGA	Field-Programmable Gate Array
FPM	Fourier Ptychographic Microscopy
GDDR	Graphics Double Data Rate

GHz	Gigahertz
GPU	Graphic Processing Unit
HDF	Hierarchical Data Format
iGPU	Integrated Graphic Processing Unit
ITDPM	Iterative Tomographic Deconvolution Phase Microscopy
KB	Kilobyte
LAPACK	Linear Algebra PACKage
LCPM	Liquid Crystal Phase Modulator
LPDDR	Low-Power Double Data Rate
LPFG	Long-period Fiber Grating
MB	Megabyte
MiB	Mebibyte
MHz	Megahertz
MKL	Intel Math Kernel Library
MPI	Massage Passing Interface
MRI	Magnetic Resonance Imaging
NA	Numerical Aperture
NMRSE	Normalized Root-Mean-Square Error
ODT	Optical Diffraction Tomography
OpenCL	Open Computing Language
OpenMP	Open Multi-Processing
OPT	Optical Projection Tomography
OTF	Optical Transfer Function
PCF	Photonic-Crystal Fiber
PCM	Phase Contrast Microscopy

PMF	Polarization-Maintaining Fiber
POTF	Phase Optical Transfer Function
PSF	Point Spread Function
QPI	Quantitative Phase Imaging
RAM	Random Access Memory
RBC	Red Blood Cell
RI	Refractive Index
RID	Refractive Index Distribution
SLIM	Spatial Light Interference Microscopy
SM	Streaming Multiprocessor
SMF	Single-Mode Fiber
SoC	A System on a Chip
SPMD	Single Program, Multiple Data
SSBPM	Split-step Beam Propagation Method
TDPM	Tomographic Deconvolution Phase Microscopy
TIE	Transport-of-Intensity Equation
TSUM	OpenMP Tasking and CUDA Streaming with Unified Memory
TV	Total Variation
UM	Unified Memory
UPM	Unified Physical Memory
UVM	Unified Virtual Memory
WOTF	Weak Object Transfer Function

LIST OF SYMBOLS

$A(\boldsymbol{r})$	Imaginary part of scattering potential (spatial absorption function, (μm^{-2})
$\mathrm{A}(\boldsymbol{\rho})$	Fourier transform of imaginary part of scattering potential (μm)
A	Linear imaging operator representing a convolution with PSF
α	Regularization parameter (unit depends on application)
B	Background intensity (μm^{-2})
γ	Update parameter (dimensionless)
D_r	Discrete derivative operator
$\delta(\cdot)$	Dirac delta function
ε	A small number (unit depends on application)
ϵ^{abs}	Absolute tolerance (dimensionless)
ϵ^{dual}	Dual feasibility tolerance (dimensionless)
ϵ^{pri}	Primal feasibility tolerance (dimensionless)
ϵ^{rel}	Relative tolerance (dimensionless)
H	Optical transfer function (dimensionless)
$H_A(\boldsymbol{\rho})$	Absorption optical transfer function (dimensionless)
$h_P(\boldsymbol{r})$	Phase point spread function (dimensionless)
$H_P(\boldsymbol{\rho})$	Phase optical transfer function (dimensionless)
$H_P^*(\boldsymbol{\rho})$	Phase optical transfer function conjugate (dimensionless)
$h(\boldsymbol{r})$	Point spread function (μm^{-3})
$h_A(\boldsymbol{r})$	Absorption point spread function (μm^{-3})
θ_c	Maximal half-angle of light exiting condenser lens (radian)
θ_j	Object rotation angle (radian)

Θ	Rotation operator (dimensionless)
$I(\boldsymbol{r})$	3D Intensity (μm^{-2})
i	Imaginary unit $\sqrt{-1}$
ι_C	Indicator function (dimensionless)
$J(v)$	Regularization function for ITDPM (dimensionless)
j	Index
k_o	Freespace wavevector magnitude (μm^{-1})
L_ρ	Augmented Lagrangian (μm^{-2})
$\boldsymbol{\lambda}$	Wavelength (μm)
λ	Regularization parameter for ADMM (dimensionless)
M	Mask (dimensionless)
m	Angle number
N	Number of angles
NA_c	Numerical aperture of condenser lens (dimensionless)
NA_o	Numerical aperture of objective lens (dimensionless)
n	Refractive index (dimensionless)
n_o	Background refractive index (dimensionless)
n_{ideal}	Ideal refractive index (dimensionless)
n_{rec}	Recovered refractive index (dimensionless)
P_B	Background phase (dimensionless)
$\boldsymbol{\rho} = (\rho_x, \rho_y, \rho_z)$	Spatial frequency (μm^{-1})
ρ	Penalty parameter (dimensionless)
$\boldsymbol{r} = (x, y, z)$	3D spatial coordinate (μm)
$\|\|r\|\|_2$	Primal residual

$\|\|s\|\|_2$	Dual residual
τ	The maximum difference allowed in magnitude
$V(\boldsymbol{\rho})$	Fourier transform of scattering potential (μm)
$v(\boldsymbol{r})$	Scattering potential (μm^{-2})
x	Spatial coordinate perpendicular to optical axis (z) and rotational axis (y) (μm)
$\psi(v, \mathrm{I})$	Cost function (μm^{-1})
y	Spatial coordinate defining rotational axis (μm)
z	Spatial coordinate defining the optical axis (μm)
$\|\| \cdot \|\|_2$	ℓ_2 norm
$\|\| \cdot \|\|_{2,1}$	ℓ_2 norm followed by ℓ_1 norm
$\|\| \cdot \|\|_{TV}$	Total Variation regularization which is the ℓ_1 norm of the magnitude of discrete gradient computed at each voxel
∇	Gradient operator
$\|\nabla v\|$	Gradient magnitude of ∇v (μm^{-3})
$(\cdot)^T$	Transpose
$\odot$	Point-wise multiplication operator
$*$	Convolution
$(\cdot)^*$	Complex conjugate

SUMMARY

Quantitative Phase Imaging (QPI) is a powerful imaging technique for measuring refractive index distribution of transparent objects such as biological cell and optical fibers. The quantitative, label-free approach of QPI provides preeminent advantages in biomedical application and the characterization of optical fibers. Tomographic Deconvolution Phase Microscopy (TDPM) is a promising 3D QPI method that combines diffraction tomography, deconvolution, and through-focal scanning with object rotation to achieve isotropic spatial resolution.

This thesis presents significant improvements in the implementation of TDPM. OpenMP Tasking and CUDA Streaming with Unified Memory (TSUM) leverage CPU multithreading and GPU computing on a System on a Chip (SoC) with unified memory to achieve up to 1.74x speedup over the original 3D TDPM. Furthermore, an efficient iterative algorithm, Alternating Direction Method of Multipliers (ADMM), is applied to 3D TDPM to reconstruct phase objects that are shift-variant in three spatial dimensions. ADMM-TDPM achieves speedups of 5x in image acquisition time and greater than 10x in image processing time with accompanying higher accuracy compared to TDPM.

CHAPTER 1. INTRODUCTION

1.1 Motivation and Impact

Quantitative Phase Imaging (QPI) is a powerful scientific imaging technique that measures the optical path length and the Refractive Index (RI) distribution of transparent objects such as biological cells and optical fibers. Three-Dimensional (3D) QPI has vast potential in biomedical applications as it is non-invasive and provides quantitative data [1-13]. QPI does not require invasive cutting or harmful contrast agents such as fluorescence protein or dyes that could perturb the natural state of biological cells and tissues. Also, QPI can provide quantitative measurements of morphological, chemical, and mechanical parameters of biological cells and tissues. QPI has been used to study morphological profiles [14-19], intracellular mass transport [20-22], and cytoskeletal/organelle interactions [23]. Cell homeostasis has been investigated with QPI. The nonlinear elasticity of Red Blood Cell membranes (RBCs) caused by osmotic pressure has been measured with QPI [24]. 3D red blood coagulation structure has been reconstructed with QPI [18]. QPI can be used to study the therapeutic effects of drugs on biological cells [25-28]. Drug-induced deformability of human red blood cells has been investigated with 3D QPI [12].

QPI has also been used in clinical diagnostics, such as in cancer screening [29-38], infection detection [39, 40], and psychological disorder diagnosis [41, 42]. QPI has been proven to be an adequate tool to investigate cancer cells [43, 44]. Circulating tumor cells that are present in the blood of cancer patients have been studied with QPI to diagnose ovarian cancer [45]. Also, in neuroscience, QPI has been used to map the dry mass density of the neuronal network, investigate neuronal mass transport and growth over time [46],

and detect neuronal death [47]. Furthermore, the activities of neuronal membrane proteins that transport ion and water have been monitored with QPI [48].

More recently, QPI has been combined with artificial intelligence to automate cell detection [49, 50] and the diagnosis of diseases [51-54]. With deep learning and QPI, T cell activation has been studied [55], and automatic RBC segmentation has been performed [56]. By combining QPI, machine learning, and an augmented reality device, automatic classification and visualization of cells have been realized [57].

Another application of QPI is optical fiber characterization. QPI has been used to measure RI and residual stress profiles of optical fibers. The Fiber Bragg Grating (FBG), in particular, is widely used in telecommunications and sensing. In telecommunications, FBGs are used as band-rejection filters in wavelength-selective devices [58]. Wavelength multiplexing/demultiplexing have been realized with FBGs [58-60]. FBGs are used as sensors for measuring temperature [61, 62], strain [63, 64], pressure [65], and 3D positioning [66] in various applications. In recent years, medical devices with FBG-based haptic sensors have been fabricated for minimally invasive diagnosis and surgery [67-73] and neural interfaces [74]. Scientists and researchers have been developing and fabricating various types of FBGs for different purposes [75]. The characterization of FBGs is a crucial step in the design of high-performance FBGs. QPI can be a useful tool for profiling the physical RI of FBGs in the fabrication step [76].

1.2 Thesis Objective and Overview

The objective of the research presented in this thesis is to improve the performance of a 3D QPI technique, Tomographic Deconvolution Phase Microscopy (TDPM) [77], by

leveraging the latest technologies and implementing an iterative algorithm. TDPM, which will be discussed in detail in Chapter 2, has drawbacks in that it requires a long image acquisition time and extensive computation power with large memory. In this thesis, the disadvantages of TDPM are overcome by two methods: 1) OpenMP Tasking and CUDA Streaming with Unified Memory (TSUM) and 2) Alternating Direction Method of Multipliers TDPM (ADMM-TDPM). TSUM combines CPU and GPU parallel computing with unified memory that eliminates data transfer overhead to speed up the computation of tomographic angles in 3D TDPM. ADMM-TDPM, coupled with the Augmented Lagrangian, is an efficient iterative algorithm that optimizes the image fidelity by using total variation regularization with non-negativity and known zeros constraints. ADMM-TDPM shortens its image acquisition time by 5x and achieves a speedup greater than 10x in processing time while simultaneously improving its accuracy.

In Chapter 2, QPI, 3D TDPM, and Iterative TDPM (ITDPM) are defined and described in detail. Chapter 2 also discusses how advances in computer architectures and computation methods has impacted image processing. In Chapter 3, the advantages and disadvantages of CPU and GPU parallel computing are compared. The most frequently used arithmetic operation in TDPM, Fast Fourier Transform (FFT), and memory operation, array shift, are tested as benchmarks on various combinations of hardware and Application Programming Interfaces (APIs). In addition, the capability of TSUM for 3D TDPM RI reconstruction is demonstrated. In Chapter 4, ADMM-TDPM is developed and reconstructs a series of representative 3D objects. Both TSUM and ADMM-TDPM have great potentials to realize real-time imaging. The possible approaches to real-time imaging are described as future work in Ch. 5.

CHAPTER 2. BACKGROUND

2.1 Quantitative Phase Imaging

Transparent objects, also known as phase objects, like biological cells and optical fibers can be imaged through either intrinsic (endogenous) or extrinsic (exogenous) contrast. Imaging the intrinsic contrast of a phase object is challenging under conventional illumination because the object scatters and absorbs light weakly. One remarkable solution to this challenge is fluorescence microscopy in which cells are labeled with fluorescent proteins or dyes that produce extrinsic contrast. Despite its influence in biomedicine, however, fluorescence microscopy has the limitations of photobleaching, phototoxicity, and potential interference of fluorescent proteins with the properties of the cells.

In the 1930s, Zernike developed Phase Contrast Microscopy (PCM) to image the phase object with its intrinsic contrast. PCM enhances the contrast of interference patterns of scattered and unscattered reference light by shifting the phase of the reference light by 90° [78]. Based on PCM, several variant methods, such as differential interference contrast (DIC) microscopy and Hoffmann modulation contrast microscopy, have been developed. However, PCM suffers from optical an artifact known as the halo effect, which causes bright spots at the edge of the object and directional shadows. Also, PCM and its variant methods provide a nonlinear measure of the intensity, which cannot be inverted to provide quantitative phase data.

QPI has been developed to overcome these limitations. 2D QPI methods measure the optical path length of a phase object which is integrated along the direction of light.

Interference-based QPI is one of the most widely researched and developed 2D QPI methods. Digital Holographic Microscopy (DHM), also known as off-axis holography, is a well-known interference-based QPI. Conventional DHM captures an interference pattern (hologram) between a sample beam and an off-axis reference beam separated by an angle using a Mach-Zehnder interferometer [79]. Phase can be recovered from the hologram by numerically calculating the Fresnel diffraction patterns [80]. Unlike PCM, DHM provides the quantitative distribution of the optical path length across the object, which contains refractive index and morphologic information about the sample. The drawbacks of DHM are that it is sensitive to speckle noise, and recovering the phase distribution is computationally expensive. Spatial Light Interference Microscopy (SLIM) combines the phase-shifting principle of PCM using a reflective Liquid Crystal Phase Modulator (LCPM) and white light illumination [81]. LCPM shifts the phase in increments of 90°, and four images corresponding to each phase shift are captured. The intensities from the four images are combined to retrieve the phase using autocorrelation. The spatial uniformity associated with white light and the short coherence length of the illumination light allow speckle-free imaging with only sub-nanometer spatial background noise.

Another 2D QPI method is scanning-based. Fourier Ptychographic Microscopy (FPM) combines light-field imaging and ptychography with iterative scanning [82, 83]. FPM captures multiple perspective low-resolution images of a sample illuminated by plane waves at a number of different angles using a low-NA objective. FPM randomly initializes a high-resolution image. Fourier transformation (circular low-pass filters) is applied at a region in the low-resolution image that corresponds to a particular angle of illumination. The regions of the high-resolution image in Fourier space are replaced with the Fourier

transform of the same regions that have the square root of the intensity in the corresponding low-resolution image. Applying the low-pass filter and replacing the regions of the high-resolution image are repeated several times to reconstruct the high-resolution image. The advantage of FPM is that it offers wide-field images with simple, inexpensive hardware. However, the iterative phase recovery is computationally expensive and can be time-consuming.

Defocus-based methods utilize Abbe's theory that an image itself is the interference phenomenon instead of creating interference patterns using optical systems. For defocus methods, a number of intensity images of the sample are captured in and out of focus using a standard bright-field microscope. The phase can be reconstructed by either iterative algorithms or deterministic methods that linearizes the relation between the phase and the defocused images. The Transport-of-Intensity Equation (TIE) [84-86] is a popular linearizing method. Streibl first proved that the phase gradient of the phase object was equal to the logarithmic intensity derivative. Using this relationship, phase can be retrieved from the intensities of defocused images. TIE assumes the propagation of light to be paraxial and requires a sufficient degree of spatial coherence. The major advantage of defocus methods is that they can be implemented with a standard bright field microscope with partially coherent illumination [87].

3D QPI methods reconstruct the refractive index distribution by combining 2D QPI with optical tomography and/or deconvolution. Tomography is the most popular conventional approach to 3D QPI. The phase of the object can be measured using a 2D QPI method over a range of angles. Tomography requires rotation of either the object relative to the imaging system [88] or the illumination beam relative to the object and the optical

axis of the imaging system [89, 90]. Beam rotation can be faster than object rotation, but it cannot cover the entire range of angles due to the limited Numerical Aperture (NA) of the system. This limitation causes missing spatial frequencies, which is called the missing cone problem [91]. Object rotation can achieve isotropic spatial resolution, but it is slow and prone to misalignment and disturbance to the object during rotation.

After the phase is measured at different angles, either Optical Projection Tomography (OPT) [92] based on filtered backprojection or optical diffraction tomography (ODT) [93] based on filtered backpropagation can be used to reconstruct the RI of the object from the phase information. OPT is simpler and faster but less accurate than ODT because it does not consider the effects of diffraction and boundary refraction occurring when the object features are in the same order as the illumination wavelength. Although it produces more accurate results by accounting for the diffraction effects, ODT requires spatially and temporally coherent illumination as well as assumptions that the object has weak absorption and small RI contrast.

Deconvolution is another 3D QPI method that is based on through-focal scanning [94, 95]. A series of through-focal images can be obtained by sweeping the focal plane through the object along the optical axis of the system. Sweeping can be done with a piezoelectric objective scanner or electrically tunable lens [94, 96]. From the intensity of the object in the series of through-focal images, either iterative algorithms or linearized deconvolution model and Optical Transfer Function (OTF) inversion can be used to reconstruct the 3D RI distribution. Deconvolution can be implemented with a standard commercial microscope with partially coherent illumination. However, deconvolution also

suffers from the missing cone problem along the optical axis, which becomes a significant challenge when the object has complex RI distribution.

2.2 Tomographic Deconvolution Phase Microscopy (TDPM)

Tomographic Deconvolution Phase Microscopy (TDPM) combines diffraction tomography, 3D linearized deconvolution, and object rotation to achieve isotropic spatial resolution using a standard commercial microscope [77]. TDPM employs 3D Weak Object Transfer Function (WOTF) from the first-order diffraction tomography. The object can be represented by scattering potential

$$v(\boldsymbol{r}) \triangleq k_0^2[n(\boldsymbol{r})^2 - n_0^2] \tag{2.1}$$

where $k_0 = 2\pi/\lambda$ is the free-space wave vector magnitude for the wavelength λ, n is the RI of the object, n_0 is the background RI, and $\boldsymbol{r}$ is the 3D spatial coordinate. It can be also expressed as

$$v(\boldsymbol{r}) = P(\boldsymbol{r}) + iA(\boldsymbol{r}) \tag{2.2}$$

where $P(\boldsymbol{r})$ is the real part related to phase, and $A(\boldsymbol{r})$ is the imaginary part related to absorption. If the scattering potential is weak and RI contrast is small enough, the first-Born approximation can be used to approximate the scattered wave function by a plane wave. The 3D intensity distribution $I(\boldsymbol{r})$ can be expressed as a convolution of the scattering potential with the point-spread functions (PSFs)

$$I(\boldsymbol{r}) = B + A(\boldsymbol{r}) * h_A(\boldsymbol{r}) + P(\boldsymbol{r}) * h_P(\boldsymbol{r}) \tag{2.3}$$

where $h_A(\boldsymbol{r})$ and $h_P(\boldsymbol{r})$ are the PSFs for the absorption and phase part, respectively, and B is the uniform background intensity. The Fourier transform of the intensity spectrum can be written as

$$I(\boldsymbol{\rho}) = B\delta(\boldsymbol{\rho}) + A(\boldsymbol{\rho})H_A(\boldsymbol{\rho}) + P(\boldsymbol{\rho})H_P(\boldsymbol{\rho}) \tag{2.4}$$

where $\boldsymbol{\rho}$ is the 3D spatial frequency, $\delta(\boldsymbol{\rho})$ is the Dirac delta function, $H_A(\boldsymbol{\rho})$ is Absorption Optical Transfer Function (AOTF), $H_P(\boldsymbol{\rho})$ is Phase Optical Transfer Function (POTF), and $A(\boldsymbol{\rho})$ and $P(\boldsymbol{\rho})$ are the 3D Fourier transforms of $A(\boldsymbol{r})$ and $P(\boldsymbol{r})$, respectively. The background intensity can be removed by subtracting the average intensity. If phase objects are assumed to have negligible absorption, the intensity spectrum can be simplified as

$$I(\boldsymbol{\rho}) = P(\boldsymbol{\rho})H_P(\boldsymbol{\rho}). \tag{2.5}$$

The intensity of the object is measured N times at evenly spaced angles between 0 and 180 degrees where $N \geq \pi/\theta_c$. The marginal illumination angle θ_c is defined as $\theta_c = sin^{-1}(\frac{NA_c}{n_0})$ where NA_c is the NA of the condenser lens. Using a formal least-squares approach, the phase part of scattering potential in frequency domain can be solved as

$$V(\boldsymbol{\rho}) = \frac{\sum_{j=0}^{N-1}\left[\frac{I_{\theta_j}}{B}\right]H_P^*(\boldsymbol{\rho})}{\sum_{j=0}^{N-1}\left|H_{P_j}(\boldsymbol{\rho})\right|^2 + \alpha} \tag{2.6}$$

where j is an index associated with object rotation angle $\theta_j = j\Delta\theta$, I_{θ_j}/B are the zero-mean normalized 3D intensity spectra, $H_P^*(\boldsymbol{\rho})$ is POTF conjugate, and α is a regularization parameter.

In TDPM, two different processing steps for high and low spatial frequencies are used to ensure sufficient low-frequency resolution without aliasing. Figures 4 and 5 in [77] show the block diagrams of TDPM RI recovery for high spatial frequencies and low spatial frequencies respectively. The high-frequency algorithm includes background intensity normalization and subtraction, x- and z-slice registration, filtering with the POTF conjugate, rotation via bilinear interpolation as in Eq. (2.6). The small POTF in the denominator could cause a noise magnification problem. Thus, the transfer function should be regularized by either a hard cutoff or Wiener filtering. Finally, the scattering potential is converted to RI by:

$$n(\boldsymbol{r}) = \sqrt{\frac{V(\boldsymbol{r}) - P_B}{k_0^2} + n_0^2} \tag{2.7}$$

where k_0 is the freespace wavevector magnitude for the illuminating light and P_B is the background phase.

TDPM is less susceptible to noise as it utilizes partially coherent illumination, compared to other QPI methods that use coherent illumination and suffer from speckle noise. TDPM is inexpensive as it can be implemented on a standard microscope platform with minimal modification. However, TDPM requires a relatively long image acquisition time as a series of defocused images should be collected at a large number of angles to avoid the missing cone problem. Typically, TDPM collects 3D images at 15 angles. Processing 3D images from the 15 angles requires large memory space and expensive computational power to reconstruct high-resolution RI distributions.

2.3 Iterative Tomographic Deconvolution Phase Microscopy (ITDPM)

Iterative TDPM reconstructs the 3D RI distribution with an edge-preserving iterative regularization algorithm to reduce the image acquisition time and overcome the missing cone problem [97]. Instead of using direct deconvolution in the frequency domain, ITDPM reduces the number of illumination angles by estimating the expected image intensities as close to the measured images as possible and optimizing the estimation iteratively using gradient descent. Mathematically, the problem at hand can be represented as

$$v(\boldsymbol{r}) = \underset{v(\boldsymbol{r})}{\operatorname{argmin}} \left|\left|h(r) * v(\boldsymbol{r}) - I(\boldsymbol{r})\right|\right|_2^2, \tag{2.8}$$

where $\left|\left|f(\boldsymbol{r})\right|\right|_2$ is the ℓ_2 norm of $f(\boldsymbol{r})$, and $h(\boldsymbol{r})$ is the PSF. For simplicity, the convolution of h can be represented with a linear operator A (a detailed derivation is in [97]), and the convolution in the frequency domain becomes

$$Av = F^{-1}HFv, \tag{2.9}$$

where F is the Fourier transform, F^{-1} is the inverse Fourier transform, and H is the pointwise multiplication by the POTF. The minimization should be satisfied for the average value over all angles. Also, ITDPM considers the piecewise smoothness constraint that is described by a minimization of a regularization function $J(v) = \int \psi(|\nabla v|)dr$, where $|\nabla v|$ is the gradient magnitude of $v(\boldsymbol{r})$, and $\psi(|\nabla v|) = \sqrt{|\nabla v|^2 + \varepsilon^2}$. ε is a small number for preventing division by zero. The cost function including the edge-preserving regularization term becomes

$$\Psi(v, I) = \frac{1}{2N}\sum_m \left|\left|A\Theta_m v_m - I_m\right|\right|_2^2 + \alpha J(v), \quad (2.10)$$

where N is the number of angles, m is an angle number, Θ is a rotation operator and α is a regularization parameter.

ITDPM uses gradient descent which is a simple optimization algorithm but converges slowly. Thus, ITDPM reduces the image acquisition time, but it also increases the computation time. With the piecewise smoothness constraint, the current ITDPM approach can only be implementable for 2D objects as it assumes the object to be shift-invariant in one direction. ITDPM cannot be applied for shift-variant objects like FBGs and biological cells.

2.4 Computer Implementation for Image Processing

2.4.1 Advancement in Computer Architecture and Computation Methods

The development of transistors and integrated circuit technology enabled the development of single-core microprocessor which drastically improved computing performance from the 1970s until the early 2000s. However, the growth in single-core microprocessor performance has stagnated as Dennard scaling [98] reached its limit. Dennard scaling explains how the reduction of circuit size shortens the circuit delay time while maintaining its power density and how cooling becomes a major problem in a highly miniaturized circuit. Also, Moore's Law [99], which states that the number of transistors on a chip doubles every 18 months, no longer holds true as engineers face the challenges in decreasing the size of transistors close to the size of a few atoms. These challenges led

researchers to switch the focus from high-performance single-core processors to multicore processors. Moreover, these challenges have encouraged researchers to develop domain-specific processors that perform well on one specific computation rather than general-purpose processors.

The main advantage of multicore processors is that it allows programmers to control tread-level parallelism. Generally, parallelism can be classified into the following four types: bit-level parallelism, instruction-level parallelism, thread-level parallelism, and inter-program-level parallelism. Bit-level parallelism and instruction-level parallelism are exploited by hardware architects, and programmers can choose the appropriate hardware for their application. Inter-program-level parallelism occurs in an operating system, such as scheduling tasks and managing memory. Thread level parallelism can be controlled by programmers in their software. Thread-level parallelism on multicore processors is most widely applied for computationally intensive applications and data-intensive applications. However, writing effective thread-level parallel programs requires a high level of programming skill and effort. Many programmers and scientists struggle to improve the performance of parallel programs due to the challenges in controlling concurrency, managing data distribution, managing communication among processors, and balancing the computational load. Many libraries have been developed in various languages to facilitate the challenges of parallel computing. For example, OpenMP (Open Multi-Processing) is a widely used programming platform that allows parallelism on a multicore processor. Message Passing Interface (MPI) is another programming platform that allows parallelism on a heterogeneous distributed system such as a high-performance computer or supercomputer.

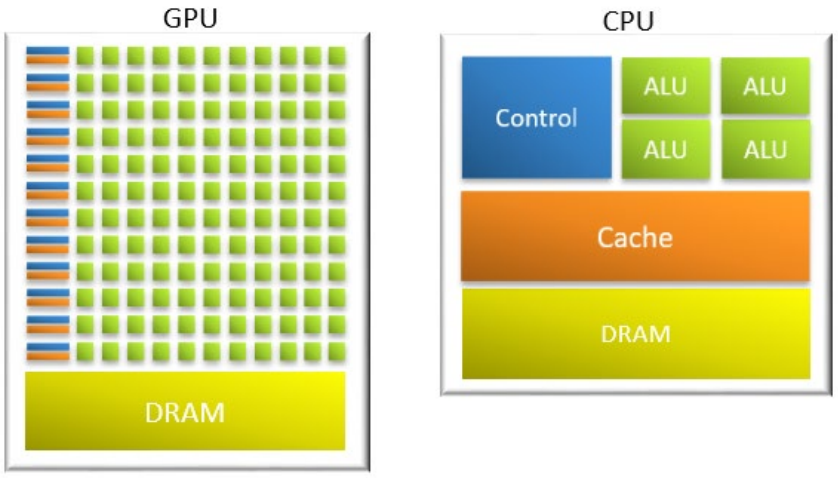

Figure 2.1— Comparison of GPU and CPU architectures. ALU: arithmetic-logic unit; DRAM: dynamic random-access memory

The Graphics Processing Units (GPUs) are specifically designed to compute the arrays of floating-points efficiently for real-time rendering. GPUs achieve high throughput by dividing a data pipeline in space, whereas Central Processing Units (CPUs) divide the pipeline in time [100]. Multithreading is much more efficient on GPUs than on CPUs; therefore, GPUs have much higher throughput than CPUs. However, GPU parallel computing still has a few disadvantages. GPU parallel computing's major challenges are in coordinating the scheduling of computation on the system processor (usually CPU) and GPU and the efficient data transfer between the system (host) memory and GPU (device) memory. Compute Unified Device Architecture (CUDA) is a parallel computing platform and programming language developed by NVIDIA to improve the productivity of GPU programming. OpenCL is another GPU programming language that is vendor-independent.

MATLAB is a powerful programming language and computing platform widely used by scientists and engineers. MATLAB's most computational functions are written in C/C++ and Fortran. MATLAB's basic linear algebra functions are highly optimized for a

specific processor with Intel Math Kernel Library (MKL) or AMD Core Math Library (ACML). MATLAB offers a parallel computing toolbox that can be used with multicore processors and GPUs. Although it is an easy and productive programming language, MATLAB does not provide programmers full control over parallelism and cannot achieve the best optimization and performance possible for a specific application.

An embedded system is a low-cost, low-power, high-throughput computer system that has a dedicated function. A typical embedded system includes a microprocessor, memory, and input/output modules, but it can be specially designed to accommodate a specific application.

2.4.2 High-Performance Computing for Image Processing

Imaging modalities have been benefited from the developments in computer architecture and computational technologies. GPU parallel computing has been used to accelerate Computed Tomography (CT) [101-103], Magnetic Resonance Imaging (MRI) [104-106], diffuse optical tomography [107], and ultrasound imaging [108, 109]. Various QPI methods have also been implemented on GPUs. For example, the phase unwrapping for diffraction phase microscopy on GPU has achieved the speedup of 41x over the CPU implementation [110]. Real-time 3D visualization using optical diffraction tomography on GPU has been demonstrated with 17x speedup [111]. The TIE has been solved on GPU in real-time [112]. In addition, for the common image operations such as fast Fourier transform and convolution, GPU outperforms CPU [113]. GPU-based 3D deconvolution for confocal microscopy also has been presented to achieve ~100x speedup [114].

Small embedded systems have been implemented for image processing in smart cameras in recent years. These smart cameras have capabilities from simple photo editing to object detection, face identification, and surveillance. Real-time video processing, such as gesture recognition, has been possible as the embedded systems become more powerful [115]. A highly optimized embedded system for 3D image processing has been developed to outperform the GPU and multicore CPU [116]. Furthermore, field-programmable gate arrays (FPGAs) have been utilized for medical image processing [117-119]. QPI of biological cell and classification has been implemented on eight FPGAs to achieve ~228x speedup compared against a single-core CPU and ~32x speedup over GPU [118]. The study has also proved that FPGAs have superior power efficiency compared to CPU or GPU.

CHAPTER 3. SPEEDUP OF 3D TDPM VIA PARALLEL COMPUTING FACILITATED BY UNIFIED MEMORY

3.1 Introduction

The original TDPM RID reconstruction program is written in MATLAB to run on a CPU. MATLAB is an easy, convenient, productive programming language that provides numerous mathematical functions. Most of MATLAB's computational functions are written in C/C++ and Fortran. MATLAB uses Basic Linear Algebra Subprograms (BLAS) and the Linear Algebra PACKage (LAPACK) included in highly optimized libraries for a specific CPU such as the Intel Math Kernel Library (MKL) and the AMD Core Math Library (ACML) [120]. MATLAB also offers a CPU/GPU parallel computing toolbox [121] that allows programmers to parallelize and accelerate their programs. However, as it is a high-level language, MATLAB does not provide programmers full control over parallelism and optimization. This is a major weakness of MATLAB because optimizing the use of hardware to a specific application can achieve a noticeable speedup. Moreover, MATLAB has unsupported and limited functions on GPU.

Parallel programming also has challenges, such as parallel overhead and programming complexity. Parallel overhead includes thread start-up/termination time, synchronization time, and overheads by compilers, libraries, etc. Parallel overhead can be reduced with better microarchitecture, compiler, and algorithms, but it cannot be entirely avoided. Furthermore, parallel programming is significantly more complex, time-consuming, and challenging than sequential programming. It requires a good

understanding of computer architecture and parallel Application Programming Interfaces (APIs) as well as customization for specific applications to take full advantage of parallelism. OpenMP [122] is a widely used parallel computing API that allows multithreading. OpenMP offers numerous constructs for users to control parallelization directly. OpenMP can be a powerful tool and is relatively easy to program for a simple parallelization. However, it still requires learning the various functionalities and understanding of the hardware to achieve the best performance improvement.

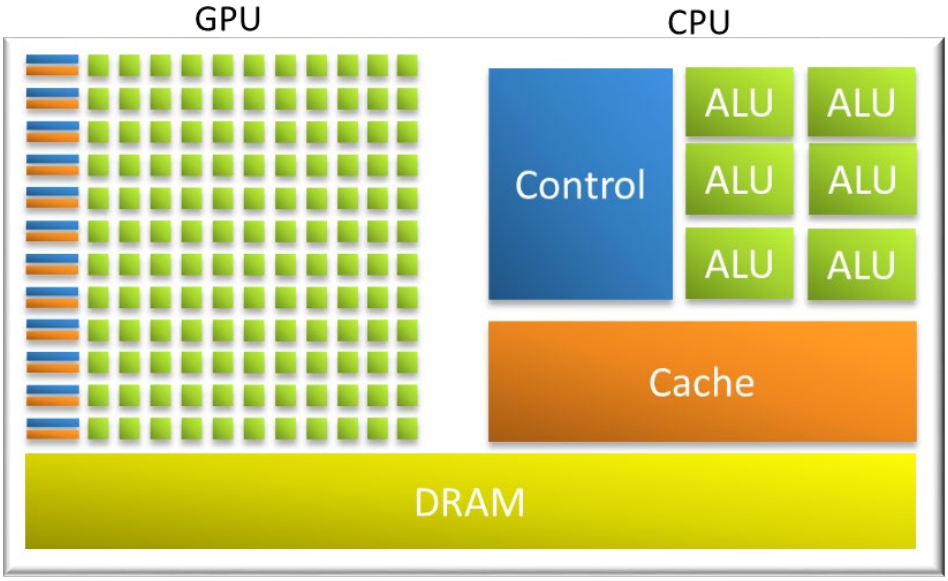

Figure 3.1 – An example diagram of a System on a Chip (SoC) with Unified Physical Memory (UPM).

In addition to the challenges of parallel computing, data transfer overhead is a major drawback of GPU computing. Most systems have CPU and GPU on separate chips, and each has its own memory (Fig 2.1). This physical separation requires data transfer between two memories, and the larger the data size, the larger the overhead. A solution to data transfer overhead is manufacturing CPU and integrated GPU (iGPU) on one chip along

with a memory that both CPU and iGPU can access, and this unified architecture is often called a System on a Chip (SoC) [123]. Figure 3.1 illustrates a simplified diagram of SoC. The memory in SoC is called Unified Memory (UM). UM is also referred to as physical SoC memory, unified shared memory, central memory, system shared memory, or global shared memory. NVIDIA's Jetson AGX Xavier is an embedded system that has unified architecture, Tegra SoC [124]. Jetson AGX Xavier is specially designed for autonomous machines and AI. AGX Xavier's unified memory allows both CPU and iGPU to access the memory and doesn't require data transfers. Also, AGX Xavier is low-cost and power-efficient compared to an average personal computer and a GPU.

In 2014, NVIDIA introduced UM as a virtual coherent memory that allows CPU and GPU to share the same memory address [125-127]. In this thesis, to avoid confusion, UM indicating a virtual memory is referred to as Unified Virtual Memory (UVM), UM indicating a physical memory is referred to as Unified Physical Memory (UPM), and UM refers to the method of using UVM facilitated by UPM. The UPM on Jetson AGX Xavier can be managed with CUDA UVM. Several studies have evaluated the performance of UM on the Tegra SoCs, TK1 [128-130], TK2 [131], and TX1 [132], which are predecessors of AGX Xavier. In this chapter, the challenges in CPU and GPU parallel computing are addressed, and the capability of UM on Jetson AGX Xavier for 3D TDPMM is demonstrated.

3.2 CPU vs. GPU

The performance of a program can vary greatly from one device to another. CPU and GPU have their strengths and weaknesses and are suitable for different applications. To

properly customize the use of CPU and GPU for TDPM, several different hardware and APIs have been tested with the most frequently used functions, the fast Fourier transform (FFT) and array shift. The CPU used in this study is Intel Xeon Silver 4110 with a base frequency of 2.10 GHz and 11 MB L3 cache, accompanied by 64 GB RAM (Table 3.1). The GPU is NVIDIA Titan RTX with 24 GB memory (Table 3.2). Also, the overhead of data transfer between the CPU and the GPU has been evaluated and compared with UM on Jetson AGX Xavier. AGX Xavier has a Carmel CPU with 8 cores and 4MiB L3 cache, a Volta iGPU, and 32 GB UM. Titan RTX is a superior GPU with more Streaming Multiprocessors (SM) and CUDA cores that can perform more operations in parallel than Volta. Furthermore, Titan RTX has larger caches, a higher memory clock rate of 7.001 GHz, and a 384-bit wide memory interface compared to the 2.133 GHz memory clock rate and 256-bit wide memory buses of Volta. More details about the CPUs and GPUs used in this study is in Appendix C.

Table 3.1 – CPU comparison.

	Intel Xeon Silver 4110 CPU	**NVIDIA Carmel CPU**
Instruction Set Architecture	x86-64	ARMx8
# of Cores	8	8
# of Threads	16	8
Base Frequency	2.100 GHz	2.265 GHz
Max Frequency	3.000 GHz	-
Cache	L1: 256 KB (data) L2: 8 MB L3: 11 MB	L1: 64 KB (data) L2: 2 MiB L3: 4 MiB
Memory	64 GB DDR4	32 GB LPDDR4x (UPM)

Table 3.2 – GPU comparison.

	NVIDIA Titan RTX GPU	**NVIDIA Volta GPU**
Streaming Multiprocessors (SM)	72	8
CUDA Cores	4608	512
Tensor Cores	576	64
Base Frequency	1.350 GHz	1.377 GHz
Memory Frequency	7.001 GHz	2.133 GHz
Memory Bus Width	384-bit	256-bit
Cache	L1: 64 KB per SM L2: 6144 KB	L1: 128 KB per SM L2: 512 KB
Memory	24 GB GDDR6	32 GB LPDDR4x (UPM)

3.2.1 Arithmetic Operation: Fast Fourier Transform

The most frequently used function in 3D TDPM is the Fast Fourier Transfer (FFT), and FFT is a good benchmark function to test various architectures as it requires a lot of multiplications and additions. The following combinations of hardware and APIs are tested to evaluate speeds of 1D FFT on CPU and GPU: 1) the FFTW library in C/C++ on CPU (denoted by Non-parallel), 2) the parallelized FFTW library with OpenMP (denoted by OpenMP), 3) the MATLAB `fft()` function on Intel Xeon (denoted by MATLAB), 4) the MATLAB `fft()` function on Titan RTX using `gpuArray()` (denoted by gpuArray), and 5) the cuFFT library in C/C++/CUDA on Titan RTX (denoted by cuFFT). The FFTW library computes the Discrete Fourier Transform (DFT). Both the MATLAB `fft` function and the cuFFT library are based on the FFTW library. MATLAB 2020b is used for the methods, MATLAB and gpuArray. The CUDA driver version 10.2 is used for Titan RTX.

The data, sized from 2^2 to 2^{20} with random values between 0 and 1, have been generated with the `rand()` function in the stdlib.h library, and the same data have been used for all five methods. The elapsed time was measured with `tic-toc` in MATLAB for the MATLAB versions, and the C/C++/CUDA versions are measured with the chrono library. The elapsed times measured do not include the data transfer time between the host (CPU) memory and the device (GPU) memory. For the OpenMP method, twelve threads are used with static scheduling.

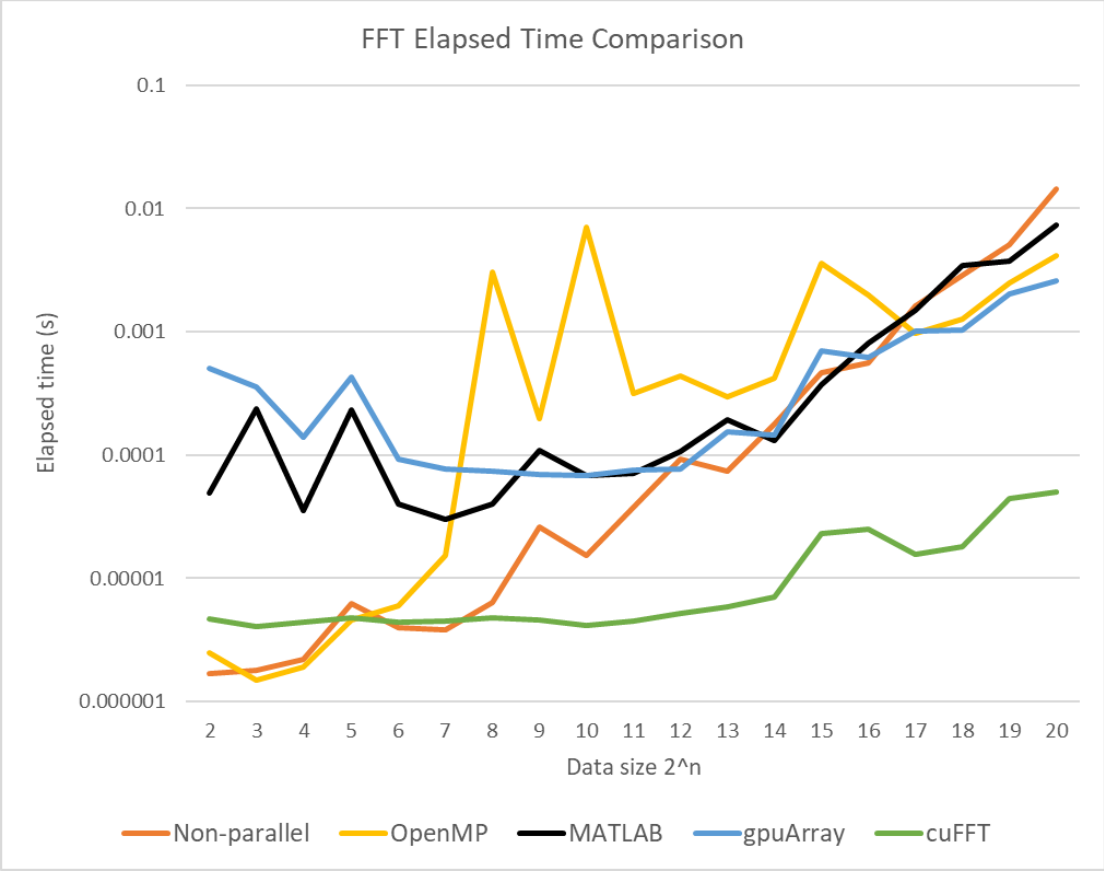

Figure 3.2 – FFT elapsed time comparison of the FFTW library in C/C++ (non-parallel), parallelized FFTW using OpenMP (OpenMP), the MATLAB `fft()` function (MATLAB), the MATLAB GPU `fft()` function using `gpuArray()` (gpuArray), and the cuFFT library (cuFFT). Elapsed time is plotted on a logarithmic scale.

Figure 3.2 shows that the cuFFT library in C/C++/CUDA on GPU is significantly faster for the large data than the other four methods. The FFTW library on CPU (Non-parallel) is the slowest for extensive data. The performance of the parallelized FFTW library using OpenMP improved as the data size increased. However, the MATLAB `fft()` function on GPU with `gpuArray()` performs slightly better than the CPU methods as the data size increases.

3.2.2 Data Transfer and Unified Memory

In MATLAB, transferring data between host memory and device memory can be performed by `gpuArray()` and `gather()`. In CUDA, `cudaMemcpy()` can be used, and memory copy type can be specified to indicate which way the data are copied. The following five combinations of hardware and APIs are compared to evaluate data transfer overhead: 1) the MATLAB `fft` function on Titan RTX with `gpuArray` excluding data transfer (gpuArray w/o memcpy), 2) the MATLAB `fft` function on Titan RTX with `gpuArray` including data transfer (gpuArray w/ memcpy), 3) the cuFFT library in C/C++/CUDA on Titan RTX excluding data transfer (cuFFT w/o memcpy), 4) the cuFFT library in C/C++/CUDA on Titan RTX excluding data transfer (cuFFT w memcpy), and 5) the cuFFT library in C/C++/CUDA on Jetson AGX Xavier with UM (Jetson UM). The data, sized from 2^2 to 2^{25} with random values between 0 and 1, have been generated with the `rand()` function in the stdlib.h library, and the same data have been used for all five methods. The elapsed time was measured with `tic-toc` in MATLAB for the MATLAB versions, the chrono library for cuFFT on Titan RTX, and `cudaEventRecord()` for Jetson UM.

For Jetson AGX Xavier, the data can be allocated on its UPM with `cudaMallocManaged()`. Although it saves data transfer time, UM has coherency maintenance overhead from managing cached memory on both CPU and iGPU [133]. The overhead can be reduced with a prefetching hint by attaching the data memory to CPU or iGPU using `cudaStreamAttachMemAsync()`.

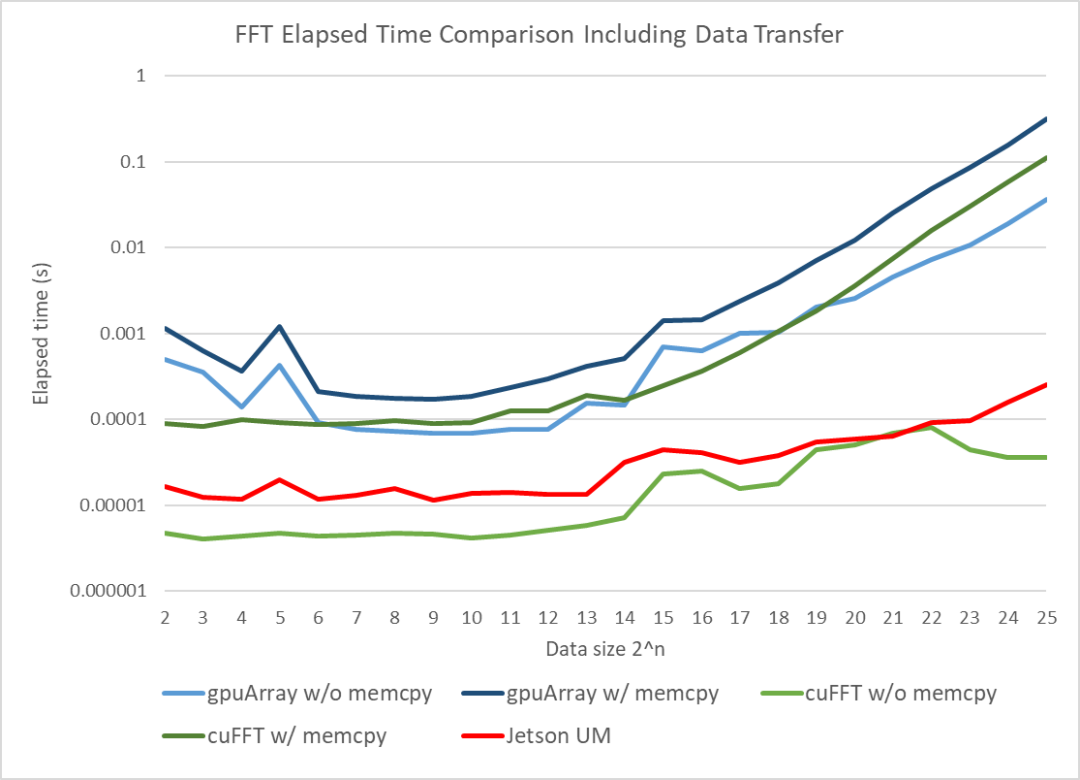

Figure 3.3 – FFT elapsed time comparison of the MATLAB GPU `fft` function using `gpuArray` excluding data transfer (gpuArray w/o memcpy), `gpuArray` including data transfer (gpuArray w/ memcpy), the cuFFT library excluding data transfer (cuFFT w/o memcpy), cuFFT including data transfer (cuFFT w/ memcpy), and the cuFFT on NVIDIA Jetson AGX Xavier using unified memory (Jetson UM). The MATLAB GPU `fft` function and the cuFFT library were ran on a NVIDIA Titan RTX GPU. Elapsed time is plotted on a logarithmic scale.

The results clearly show data transfer overhead exponentially increases as the data size increases (Fig 3.3). The method cuFFT w/o memcpy performs better than Jetson UM

as coherency management overhead slows down the performance on Jetson UM. Nevertheless, Jetson UM is clearly faster as data transfer is always required for the Titan RTX, and the elapsed times including data transfer should be compared with the Jetson UM. The results show that a program should be optimized to transfer data less frequently when using a GPU with separated memory like Titan RTX. For an iGPU with UM, data transfer optimization is not necessary, but one should be aware of concurrency maintenance overhead which can increase if the data is frequently used from both iGPU and CPU.

3.2.3 Memory Operation: Array Shift

The FFT data are often used with its zero-frequency component in the center of the array. Shifting the array requires memory operations which typically take more clock cycles than arithmetic operations. Memory speed is a major determinant of memory operation performance. High memory frequency and large bus width are preferable for applications with many memory operations. Moreover, CPUs and GPUs with large caches are advantageous as they can have fewer cache misses and memory accesses. However, if caches are too large, cache access typically slows down; thus, one should consider the optimal cache size for a specific application.

In this section, the array shift performances of four methods are evaluated. In MATLAB, the `fftshift()` or `ifftshift()` function, which calls `circshift()`, can be used to shift an array. The `fftshift()` function is performed on Intel Xeon (denoted by MALTAB) and Titan RTX with `gpuArray()` (denoted by gpuArray). Also, a custom array shift function, `cuShift()`, is programmed in C/C++/CUDA to run on Titan RTX (denoted by C/C++/CUDA) and Volta of Jetson AGX Xavier (Jetson UM). The

`cuShift()` launches a GPU kernel that shifts the elements of an array by indexing in parallel. The 3D single-precision floating-point data with sizes 32^3, 64^3, 128^3, 256^3, and 512^3 are generated with the `rand()` function in the stdlib.h library. The elapsed time was measured with `tic-toc` in MATLAB for the methods, MATLAB and gpuArray, and the methods, C/C++/CUDA and Jetson UM, are measured with the chrono library.

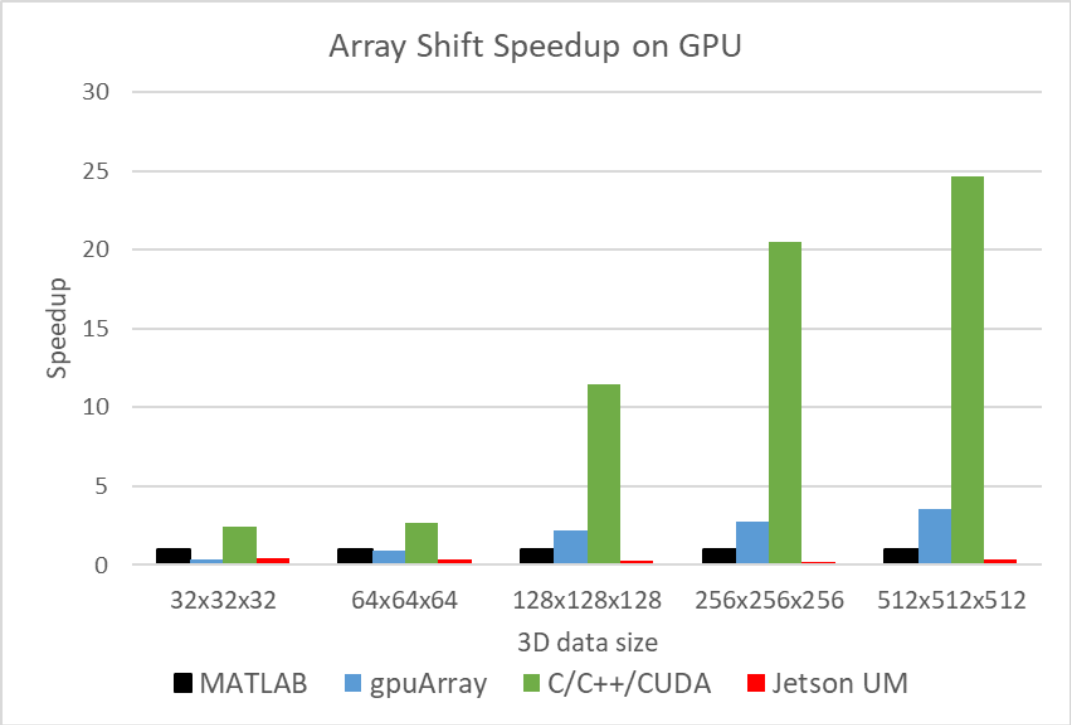

Figure 3.4 – Array shift speedup of MATLAB `gpuArray()` on Titan RTX (gpuArray), C/C++/CUDA on Titan RTX (C/C++/CUDA), and C/C++/CUDA on Volta GPU of Jetson with UM (Jetson UM) over MATLAB on Intel Xeon (MATLAB).

Figure 3.4 shows the speedups of array shift on Titan RTX (gpuArray and C/C++/CUDA) over on Intel Xeon (MATLAB) excluding data transfer time. The `cuShift()` function on Titan RTX (C/C++/CUDA) is 24.6x faster than the `fftshift()` function (MATLAB) for the data of size 512^3. On the other hand, the `cuShift()` function on Volta of AGX Xavier performs poorly with an average of 0.32x slowdown over

MATLAB. AGX Xavier's low memory clock rate seems to limit the memory operation performance. However, Jetson UM performs better than gpuArray and C/C++/CUDA when the data transfer time is included (Fig. 3.5).

It is important to note that having the data pre-loaded on caches before starting an operation can improve the performance significantly. In this experiment, the input data are generated, and the output data are pre-allocated in memory right before a timer starts; thus, some data are on caches when the array shift functions are called. The algorithm of the MATLAB `circshift()` function is proprietary, so it is difficult to analyze the results. However, the superior performance of MATLAB `fftshift()` (or `circshift()`) function could be explained with the large caches of Intel Xeon.

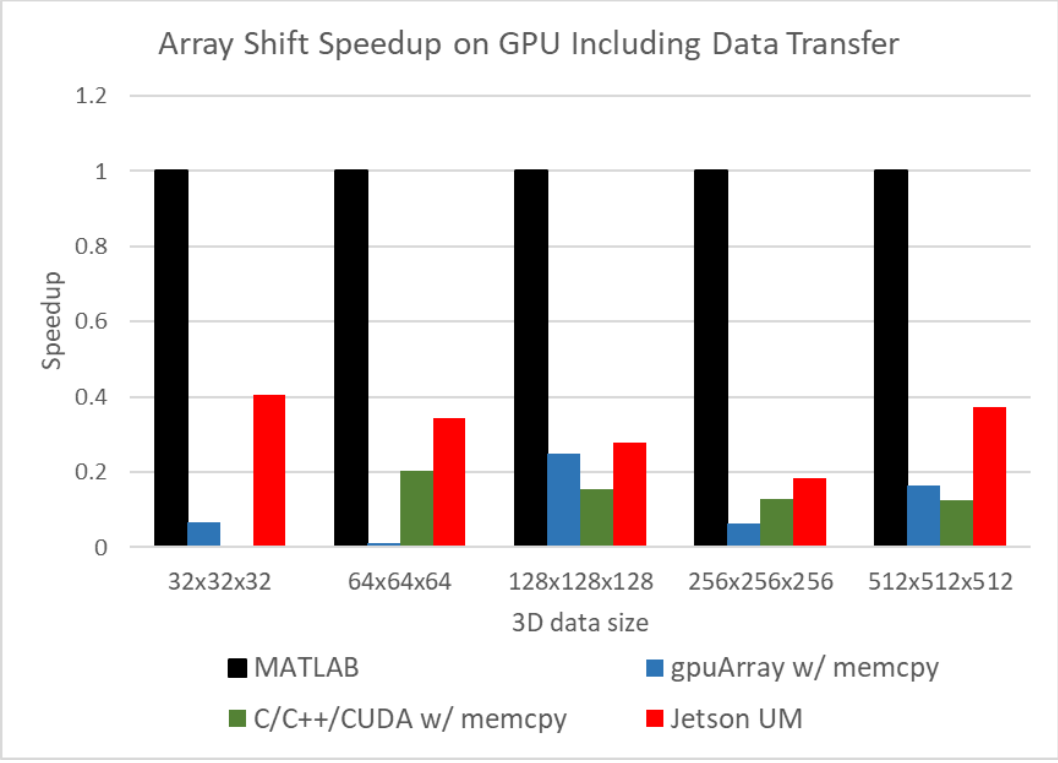

Figure 3.5 – Array shift speedup of MATLAB `gpuArray()` on Titan RTX including data transfer (gpuArray w/ memcpy), C/C++/CUDA on Titan RTX including data

transfer (C/C++/CUDA w/ memcpy), and C/C++/CUDA on Volta GPU of Jetson with UM (Jetson UM) over MATLAB on Intel Xeon.

The `cuShift()` function on Volta is also compared with a sequential CPU version and a parallel CPU version using OpenMP on the Carmel CPU of AGX Xavier. The sequential CPU version still utilizes UM, but shifting operations are performed by indexing sequentially in for loops. In the parallel CPU version, the three for loops, that iterate the indexes of three dimensions, are collapsed into one large iteration space using the `collapse(3)` clause with the for construct of OpenMP. Figure 3.6 shows that `cuShift()` on Volta is faster (a 3.4x speedup) than both sequential and parallel CPU versions on Carmel for the data of size 512^3.

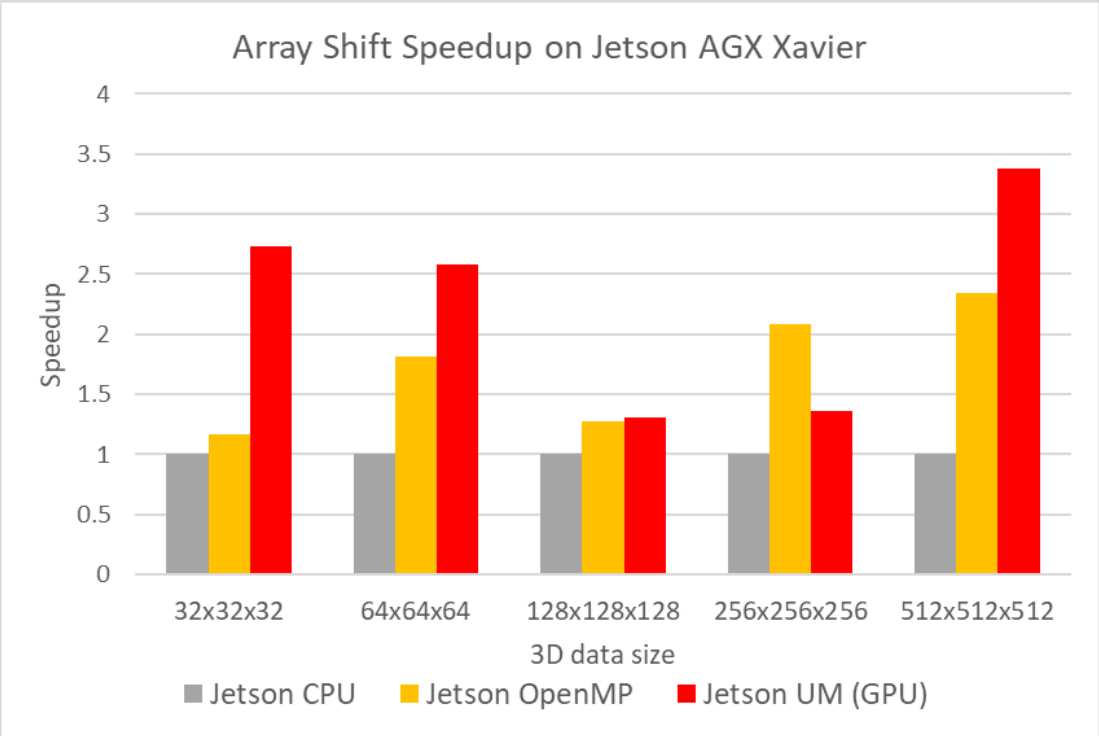

Figure 3.6 – Speedups of the parallelized array shift using OpenMP on Carmel CPU (denoted by Jetson OpenMP) and the `cuShift()` function on Volta GPU (denoted by Jetson UM (GPU)) over the non-parallel array shift on Carmel CPU (denoted by Jetson CPU).

3.3 OpenMP Tasking and GPU Streaming with Unified Memory (TSUM)

In 3D TDPM, the computation of each tomographic angle can be parallelized with a single program, multiple data (SPMD) paradigm. The operations for each tomographic angle include loading 3D intensity data, performing FFT as well as shifting, rotating, padding and masking arrays. Memory operations such as loading the intensity data and shifting are typically faster on CPUs, whereas arithmetic operations of arrays are faster on GPUs. Several studies have implemented OpenMP and CUDA together to parallelize programs on a CPU and GPU/s and achieve greater speedups over a single device [134-136]. In this study, OpenMP tasking and CUDA streaming is used to enable SPMD parallelism on both CPU and iGPU with UPM.

Tasks in OpenMP refer to the instances of executable code and data environment to be executed by specified threads [122]. Using the OpenMP task construct, we can parallelize each tomographic angle for TDPM to be run on each CPU thread. For GPU operations, CUDA streams [137] can be utilized. A CUDA stream is a sequence of operations to be run on GPUs. A CPU thread issues operations in the streams, and the GPU schedules the operations from the streams to be run when GPU threads are available. The operations in the different streams can be computed in parallel provided the threads and data are available. Figure 3.7 demonstrates the flow of operations and data of OpenMP Tasking and CUDA Streaming with Unified Memory (TSUM) for the computations of tomographic angles in 3D TDPM.

In parallel computing, a race condition often causes a bottleneck that limits performance improvement. Race conditions occur when a thread needs to wait for output

data from another thread or two or more threads perform memory operations for the same data. A simple way to avoid data race conditions is to create copies of the data so each thread can have its own data. The computations of tomographic angles in TDPM require several input data. In TSUM, before tasks are employed, copies of the input data are created, and the output data from each task are pre-allocated at separate locations to avoid the CPU and GPU threads competing for the same memory location.

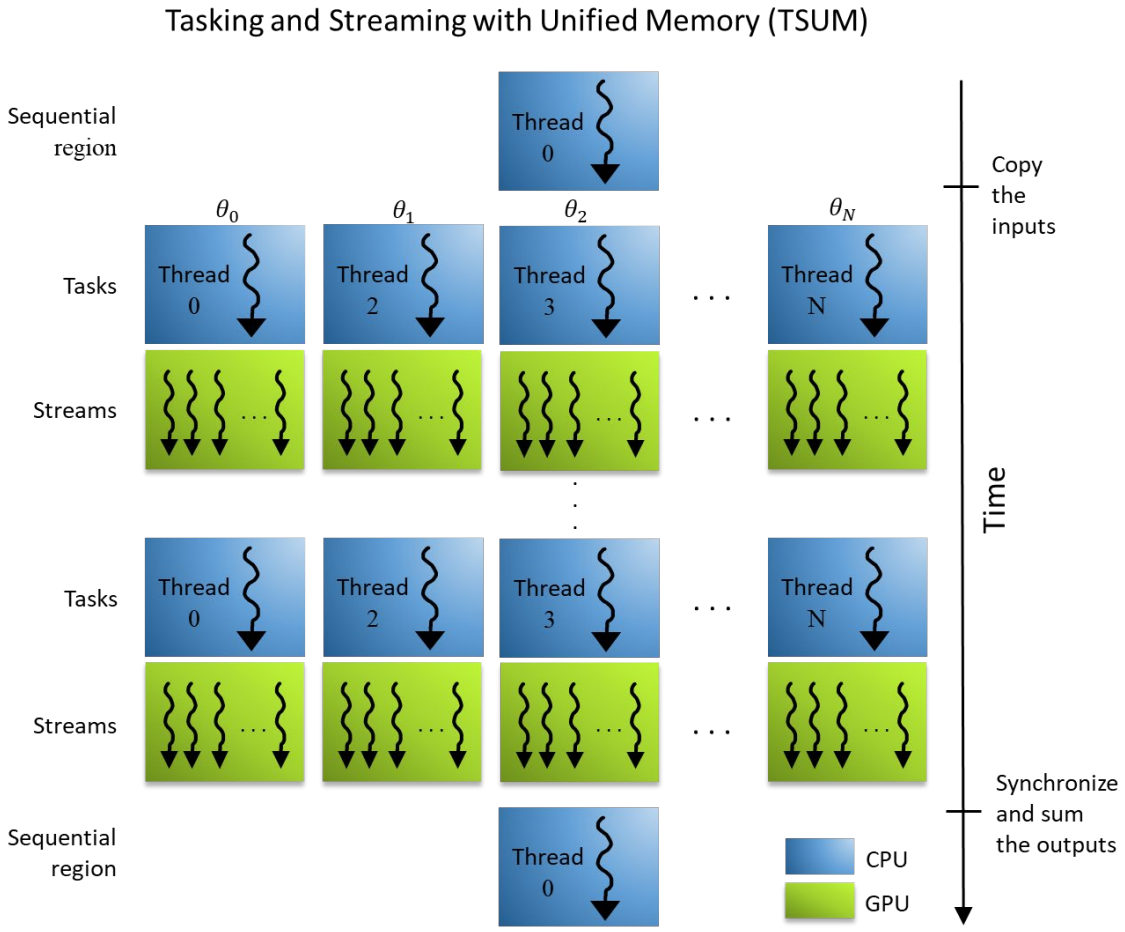

Figure 3.7 – The parallelized computations of tomographic angles in 3D TDPM with the OpenMP tasking construct and CUDA streaming facilitated by UPM (TSUM).

```
// Copy the input data
…
// Allocate memory for the output data
…
#pragma omp parallel num_threads(N)
{
   #pragma omp single
   {
      #pragma omp task
      {
         compute_angle(0,…);
      }
      #pragma omp task
      {
         compute_angle(12,…);
      }
      #pragma omp task
      {
         compute_angle(24,…);
      }
               …
   }
}
// Sum the output data from each angle
…
```

```
void compute_angle(angle, …)
{
   cudaStream_t stream;
   cudaCreateStream(stream);

   // load intensity data (CPU)
      …
   // compute scattering potential (GPU)
      …
  cudaStreamSynchronize(stream);
  // more computations on CPU and GPU
      …

   cudaDestroy(stream);
}
```

Figure 3.8 – A sample code of TSUM in 3D TDPM.

Figure 3.8 shows a sample code of TSUM for computing tomographic angles in 3D TDPM. AGX Xavier has eight available threads, and TSUM creates seven tasks (7-TSUM). The first task processes the first three angles, and each of the other tasks processes two angles. The first angle at 0° does not require rotations of intensity data, thus it is processed faster than the other angles. In each task, a CUDA stream is created, and the input and output data are attached to the stream using `cudaStreamAttachMemAsync()`. The data managed by UVM and attached to the stream can be accessed from both iGPU and CPU. Within a task, more than one stream can be created, and attached data can be shared by the streams within the task. At the end of each operation on GPU, the streams

should be synchronized with `cudaStreamSychronize()`. At the end of each angle computation, OpenMP implicitly synchronizes the CPU threads and memory.

The outputs of each task are scattering potentials of each angle. After the streams are destroyed, the tasks are terminated, the summation of scattering potentials is performed on GPU threads. After the completion of tomographic angle computation, two OpenMP tasks and CUDA streams (2-TSUM) are used for high and low frequency filtering. The filtered outputs are summed and converted to RID.

The performance of 7-TSUM on Jetson AGX Xavier (denoted by TSUM) was compared with the original MATLAB version on Intel Xeon (denoted by MATLAB), a MATLAB GPU version on Titan RTX (denoted by MATLAB GPU), and an optimized MATLAB GPU version on Titan RTX (denoted by Optimized MATLAB GPU). Revising a MATLAB program to run on a GPU can be done by simply using `gpuArray()`, and this is how the MATLAB GPU version was created. The optimized MATLAB GPU version, on the other hand, was further optimized for GPU. Unnecessary data transfers and memory operations such as `fftshift()` were removed. Also, the order of operations has been optimized to avoid cache misses as much as possible. These optimizations have been applied to TSUM as well.

3D TDPM was simulated with the four methods for the four different size intensity datasets (64x64x32, 128x128x64, 256x256x128, and 512x512x256). The intensity datasets were created using the modified split-step beam propagation method as in [77] with a simulated object, 3D Shepp-Logan phantom [138]. The intensity datasets were saved as .mat files and loaded by the `load()` function in MATLAB. In C/C++/CUDA, the datasets

were stored in .h5 files and loaded by the HDF5 library (Hierarchical Data Format version 5) [139]. The MALTAB's default data type, double-precision (8 byte), was used for the MATLAB versions, whereas single-precision (4 byte) was used for TSUM. The elapsed time was measured with `tic-toc` in MATLAB for the MATLAB versions and `cudaEventRecord()` for TSUM. Each method was executed five times, and the average elapsed time was recorded.

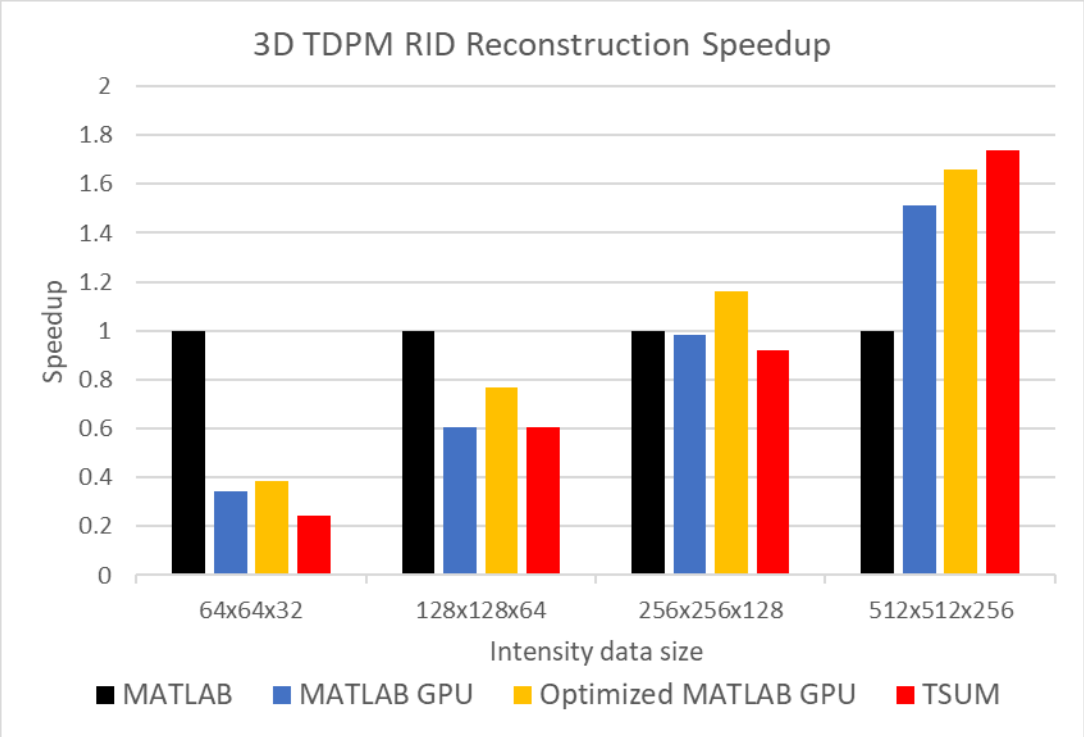

Figure 3.9 – 3D TDPM RID reconstruction speedups the MATLAB GPU version on Titan RTX (MATLAB GPU), the optimized MATLAB GPU version on Titan RTX (Optimized MATLAB GPU), and the C/C++/CUDA version on Jetson AGX Xavier (TSUM) relative to the MATLAB CPU version on Intel Xeon (MATLAB).

Table 3.3 – 3D TDPM RID reconstruction elapsed times in seconds.

	MATLAB	MATLAB GPU	MATLAB GPU Optimized	Jetson
64x64x32	1.5615	4.5341	4.0520	6.3616
128x128x64	8.9953	14.826	11.691	14.915
256x256x128	80.861	82.193	69.559	87.744
512x512x256	1267.1	837.84	763.24	729.64

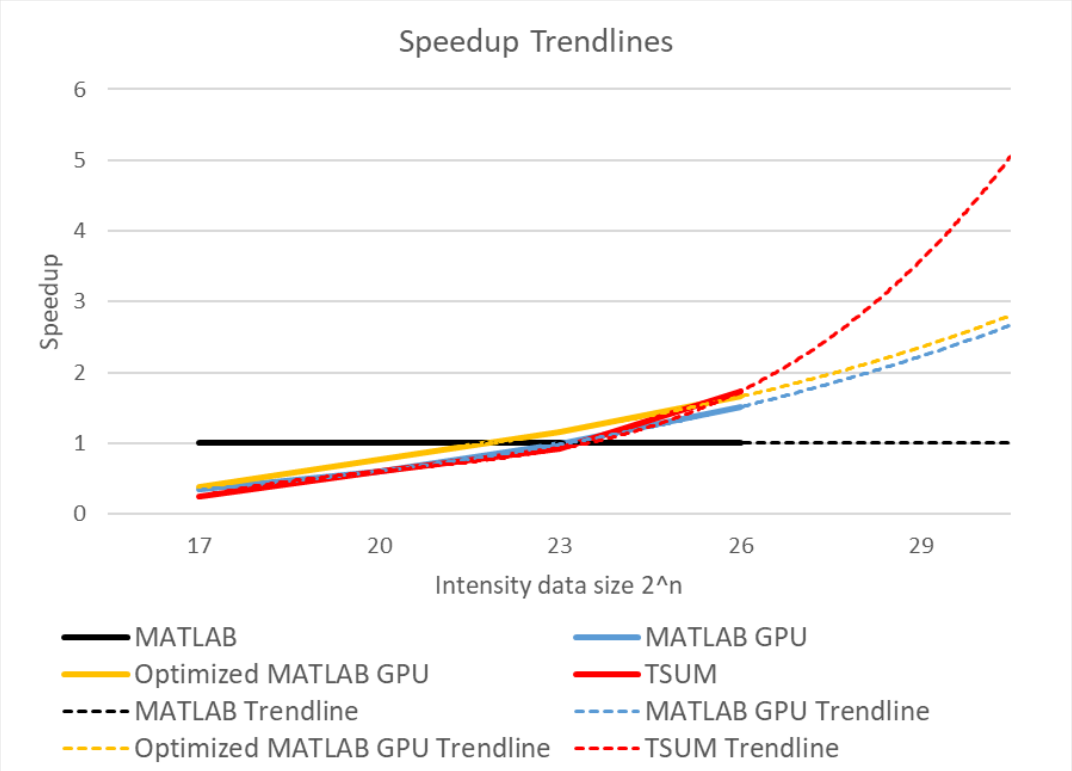

Figure 3.10 – Speedup trendlines. The actual speedups are represented in solid lines. the 5th order polynomial trendlines are drawn to predict the speedups for larger intensity datasets.

3.4 Results and Discussion

The results, presented in Fig 3.9 and Table 3.3, show that the MATLAB CPU version performs better for the small data (64x64x32 and 128x128x64). However, as the data size increases, the methods on GPU are faster than the CPU version. Moreover, TSUM performs the best with a 1.74x speedup over MATLAB for the intensity data of size 512x512x256.

In Fig. 3.10, the speedups are represented in solid lines over the size data, and the 5^{th} order polynomial trendlines of the speedups drawn in dotted lines. The trendline of TSUM has steeper positive slopes at larger data sizes than the other MATLAB GPU methods. However, it is important to note, that the speedup values depend on the elapsed time of the MATLAB CPU version. The elapsed time of the MATLAB CPU version is expected increase exponentially as the data size increases, and the trendline might not be the best prediction of the speedups. Nonetheless, it is clear that TSUM can perform better than the other methods.

The simulation demonstrates the capability of TSUM, but TSUM on AGX Xavier currently has several limitations. As it is a relatively new technology, UM are not supported by most APIs or have limited functionalities. Typically, the OpenMP shared-memory model can be used to share data among threads for traditional architecture, but it is not supported for UM. The OpenMP 5.0 and later versions offers unified memory management (`unified_shared_memory`) [122], but it is not supported on AGX Xavier yet. Due to this limitation, the data managed by UVM should be copied explicitly for each task in the program. Although creating the copies prevents race conditions, the copies of data can be

too large and overflow the available memory. In this case, only a few angles that the physical memory allows should be computed in parallel, and the rest of the angles should be scheduled for later. For the 512x512x256 intensity data, 7-TSUM occupies about 25GB of memory on AGX Xavier. Another shortcoming of AGX Xavier is that its Carmel CPU has a relatively small cache compared to the Intel Xeon silver 4110 CPU, which causes a great number of cache misses and memory accesses. These limitations will soon be overcome as more powerful SoCs and APIs for UM are developed.

3.5 Summary

In this chapter, the CPU and GPU parallel computing were compared, and the capability of TSUM for 3D TDPM was demonstrated. The most frequently used arithmetic operation, FFT, and memory operation, array shift, are tested as benchmarks on various combinations of hardware and APIs. The programs written in C/C++/CUDA to run on Titan RTX performed the best for both FFT and array shift. However, data transfer overhead negated its high performance. When the data transfer time is included, AGX Xavier with UM was significantly faster for FFT, and the Intel Xeon CPU was faster for memory operations.

In addition, 3D TDPM was simulated with 7-TSUM on Jetson AGX Xavier and compared in terms of speed with the MATLAB versions on Intel Xeon and Titan RTX for the various sizes of data. The methods on GPU performed better than the original 3D TDPM MATLAB as the data size increased. 7-TSUM performed the best with a 1.74x speedup over the original 3D TDPM, even though Titan RTX and its memory are far superior to the Volta of AGX Xavier and its UPM. This result shows the power of UPM.

With the development of hardware and software utilizing UM, TSUM has an even greater potential to further improve the performance of 3D TDPM and realize the goal of real-time imaging.

CHAPTER 4. ADMM APPROACH FOR EFFICIENT ITERATIVE TOMOGRAPHIC DECONVOLUTION RECONSTRUCTION

4.1 Introduction

Iterative Tomographic Deconvolution Phase Microscopy (ITDPM) [97], as introduced in Chapter 2.2, has been successful in reducing the image acquisition time and overcoming the missing cone problem without a large compromise in accuracy. However, ITDPM increases the computation time as its optimization method, gradient descent, converges slowly. Furthermore, ITDPM is only applicable to 2D objects as it assumes the object to be shift-invariant in one direction and thus cannot be applied to shift-variant objects like Fiber Bragg Gratings (FBGs), Long-Period Fiber Gratings (LPFGs), and biological cells.

The missing cone problem is a common issue in image reconstruction and has been computationally addressed with iterative algorithms [140-145] such as non-convex edge-preserving with half-quadratic optimization [146, 147], total variation [89, 148-151], maximum-likelihood expectation- maximization [152], multigrid algorithm [153], compressive sensing [154], and neural networks [155-157].

For 3D images, the gradient descent approach becomes even more challenging. The joint optimization of multiple image characteristics becomes unworkably slow. However, it has been recognized that invoking Alternating Direction Method of Multipliers (ADMM) can allow the separate optimization of the image parameters. This has been done

successfully, for example, by Chan *et al.* [158] in the deblurring of video images. These researchers treated a time series of 2D video images as a 3D image. They separately optimized 1) the data fidelity and 2) the total variation regularization to produce high quality video. In another successful application of ADMM, Ikoma *et al*. [159] treated low-photon-count 3D fluorescence images. In Ikoma's work, there is separate optimization of 1) the data fidelity by minimizing the Poisson noise, 2) the Hessian-Schatten norm, and 3) the indicator function given by the non-negative orthant.

In this chapter, ADMM is applied to TDPM to shorten its image acquisition and processing times while improving its accuracy. The resulting through-focal scanned images are processed using ADMM together with the Augmented Lagrangian Method to optimize separately 1) the data fidelity by minimizing Gaussian noise, 2) the scattering potential through total variation regularization, and 3) the indicator function consisting of non-negativity and known zeros in the image. The convergence of the ADMM in minimizing the Augmented Lagrangian Method is significantly improved by introducing a heuristic "varying penalty parameter" following the procedure described by Boyd [160, 161]. ADMM-TDPM can reconstruct phase objects that are shift-variant in three spatial dimensions. ADMM-TDPM achieves speedups of 5x in image acquisition time and greater than 10x in image processing time with simultaneously higher accuracy compared to TDPM. These results have been submitted for publication to *Applied Optics* [162].

4.2 ADMM-TDPM Algorithm

Alternating Direction Method of Multipliers (ADMM) is an algorithm for solving convex minimization problems of the following form [160]:

$$\min_{x,z} f(x) + g(z)$$
$$\text{s.t. } Ax + Bz = C \tag{4.1}$$

ADMM can be used to minimize an objective function $F(x) = f(x) + g(x)$ where minimization of F has no closed-form solution. This minimization is performed separately on f and g by introducing an equality constraint and variable z. This is often the case for a data-fidelity function f and regularization function g. This approach is similar to other operator splitting methods such as split-Bregman iterations, as utilized in [163] and half-quadratic splitting as in [146].

Using the image rotation objective from [97], we can formulate a convex minimization problem using total variation (TV) regularization with a constraint for non-negativity and known zeros in the solution. The following equation describes this problem for N angles. The quantity I_m is the intensity stack measured at angle m, v is the scattering potential, Θ_m is a rotation operator for an angle m, and A_{-m} is the convolution by the PSF, rotated by angle $-m$. M is a mask that is 1 where there are known zeros and 0 otherwise, and $\odot$ is a point-wise multiplication. That is,

$$\min_{v} \frac{1}{2N} \sum_{m} ||A_{-m} v - \Theta_m I_m||_2^2 + \alpha ||v||_{TV}$$
$$\text{s.t. } v \geq 0, \tag{4.2}$$
$$v \odot M = 0$$

The first term of the objective function is the data fidelity term, which ensures that the recovered scattering potential matches the data given the presence Gaussian noise. The second term is total variation regularization, which is the ℓ_1 norm of the magnitude of the

discrete gradient computed at each voxel. The quantities D_x, D_y, and D_z are discrete derivative operators in the x-, y-, and z-directions. Thus,

$$||v||_{TV} = \sum_i \left[(D_x v)_i^2 + (D_y v)_i^2 + (D_z v)_i^2\right]^{1/2} \tag{4.3}$$

For more details, see the discussion of the TV/ℓ_2 problem in [158]. When used in regularization, total variation constrains the magnitude of the gradient to be small while allowing for large jumps (sharp edges) to exist in the solution. This encourages the method to select a solution with a sparse gradient. The goal of including the regularization term is to help solve the missing-cone problem and recover the scattering potential using fewer tomographic angles, which can significantly improve acquisition time for TDPM. In contrast, TDPM in [77] uses ℓ_2 regularization on the scattering potential, resulting in a smooth solution that does not preserve edges and requires a greater number of angles to achieve good accuracy.

While total variation regularization has advantages, it is not smooth, and an objective using total variation as regularization with a quadratic data fidelity term has no closed-form solution. In order to minimize this objective efficiently, we used ADMM. To use ADMM on this problem, we restructured our objective to match the format of Eq. (4.1). First, the constraints can be replaced with an indicator function, $\iota_C(v)$, since ADMM only allows a matrix equality constraint:

$$\iota_C(v) = \begin{cases} \infty & v \geq 0 \; and \; v \odot M = 0 \\ 0 & otherwise \end{cases} \tag{4.4}$$

Including ι_C as part of the objective results in the same minimum value as the original objective. Any violation of the constraints results in the objective being infinite. Second, we must introduce equality constraints to split the quadratic, total variation, and indicator function terms. This allows us to take advantage of the operator splitting in ADMM. The resulting minimization problem is:

$$\begin{aligned} \min_{v,z_1,z_2} \ & \frac{1}{2N}\sum_m ||A_{-m}v - \Theta_{-m}I_m||_2^2 + \alpha||z_1||_{2,1} + \iota_C(z_2) \\ \text{s.t.} \quad & Dv = z_1 \quad D = \left[D_x^T D_y^T D_z^T\right]^T, \\ & v = z_2 \end{aligned} \tag{4.5}$$

where $||\cdot||_{2,1}$ is the ℓ_2 norm computed across the x-, y-, and z-dimensions followed by the ℓ_1 norm for the entire vector as in Eq. (4.3). The scaled augmented Lagrangian, for penalty parameter ρ, can be written as:

$$\begin{aligned} L_\rho(v, z_1, z_2, \mu_1, \mu_2) = & \frac{1}{2N}\sum_m \|A_{-m}v - \Theta_{-m}I_m\|_2^2 + \alpha\|z_1\|_{2,1} \\ & + \iota_C(z_2) + \frac{\rho}{2}\|Dv - z_1 + \mu_1\|_2^2 \\ & + \frac{\rho}{2}\left||v - z_2 + \mu_2|\right|_2^2 - \frac{\rho}{2}||\mu_1||_2^2 - \frac{\rho}{2}||\mu_2||_2^2 \end{aligned} \tag{4.6}$$

where $z_1 = \left[z_{1,x}^T z_{1,y}^T z_{1,z}^T\right]^T$, and $\mu_1 = \left[\mu_{1,x}^T \mu_{1,y}^T \mu_{1,z}^T\right]^T$ can be separated into corresponding x, y, and z components. ADMM-TDPM is thus

$$v^{k+1} = \underset{v}{\operatorname{argmin}}\, L_\rho(v, z_1^k, z_2^k, \mu_1^k, \mu_2^k) \tag{4.7}$$

$$z_1^{k+1} = \underset{z_1}{\operatorname{argmin}}\, L_\rho\left(v^{k+1}, z_1, z_2^k, \mu_1^k, \mu_2^k\right) \tag{4.8}$$

$$z_2^{k+1} = \underset{z_2}{\operatorname{argmin}}\, L_\rho\left(v^{k+1}, z_1^{k+1}, z_2, \mu_1^k, \mu_2^k\right) \tag{4.9}$$

$$\mu_1^{k+1} = \mu_1^k + Dv - z_1 \tag{4.10}$$

$$\mu_2^{k+1} = \mu_2^k + v - z_2 \tag{4.11}$$

To solve the minimization in Eq. (4.7), we can take the gradient with respect to v and set it to zero. This minimization has a closed-form solution since it is quadratic. For a full derivation see Appendix C. We can use three fast Fourier transforms to solve the minimization quickly because A, D_x, D_y, D_z and the identity matrix are all block circulant matrices:

$$\hat{A}^T\hat{I} \leftarrow \frac{1}{N}\sum_m A_{-m}^T \Theta_{-m} I_m \tag{4.12}$$

$$|\mathcal{F}\{\hat{A}\}|^2 \leftarrow \frac{1}{N}\sum_m |\mathcal{F}\{A_{-m}\}|^2 \tag{4.13}$$

$$|\mathcal{F}\{D\}|^2 \leftarrow |\mathcal{F}\{D_x\}|^2 + |\mathcal{F}\{D_y\}|^2 z + |\mathcal{F}\{D_z\}|^2 \tag{4.14}$$

$$v^{k+1} \leftarrow \mathcal{F}^{-1}\left\{\frac{\mathcal{F}\{\hat{A}^T\hat{I} + \rho D^T\left(z_1^k - \mu_1^k\right) + \rho(z_2^k - \mu_2^k)\}}{|\mathcal{F}\{\hat{A}\}|^2 + \rho(|\mathcal{F}\{D\}|^2 + 1)}\right\} \tag{4.15}$$

The quantities $\hat{A}^T\hat{I}$, $|\mathcal{F}\{\hat{A}\}|^2$, and $|\mathcal{F}\{D\}|^2$ do not depend on any variables being optimized, so they need to be computed only once prior to the iteration. Additionally, the entire denominator can be precomputed provided ρ does not change. Additionally, we implemented the D^T operator by taking the sum of $D_x^T v$, $D_y^T v$, and $D_z^T v$. The operators D_x^T,

D_y^T, D_z^T, in turn, were implemented using a circular convolution with a difference kernel, [0, -1, 1], in the x-, y-, and z-directions.

The minimization step in Eq. (4.8) is identical to the one presented in [158], except with a different scaling for the scaled version of ADMM. That is,

$$u_x \leftarrow D_x v^{k+1} + \mu_{1,x} \tag{4.16}$$

$$u_y \leftarrow D_y v^{k+1} + \mu_{1,y} \tag{4.17}$$

$$u_z \leftarrow D_z v^{k+1} + \mu_{1,z} \tag{4.18}$$

$$u \leftarrow [u_x^2 + u_y^2 + u_z^2]^{1/2} \tag{4.19}$$

$$z_{1,x}^{k+1} \leftarrow \max\{0, u - \frac{\lambda}{\rho}\} \cdot \frac{u_x}{u} \tag{4.20}$$

$$z_{1,y}^{k+1} \leftarrow \max\{0, u - \frac{\lambda}{\rho}\} \cdot \frac{u_y}{u} \tag{4.21}$$

$$z_{1,z}^{k+1} \leftarrow \max\{0, u - \frac{\lambda}{\rho}\} \cdot \frac{u_z}{u} \tag{4.22}$$

We implemented the D_x^T, D_y^T, and D_z^T operators by performing a circular convolution in the spatial domain with a difference kernel, [1, -1], in the x-, y-, and z-directions.

The minimization in Eq. (4.9), after discarding terms not involving z_2, is the z_2 that minimizes the following objective function:

$$\min_{z_2} \quad \iota_C(z_2) + \frac{\rho}{2}||v - z_2 - \mu_2||_2^2 \tag{4.23}$$

The solution is the value closest to $v + \mu_2$ that satisfies the constraints from the indicator function. To satisfy the non-negativity constraint, we take the maximum between $v + \mu_2$

and 0. To satisfy the known zeroes constraint, we set every voxel that is known to be 0 (where $M_i = 1$) to 0. That is,

$$z_2^{k+1} \leftarrow \max\{0, v^{k+1} + \mu_2^k\} \tag{4.24}$$

$$(z_2^{k+1})_i \leftarrow 0 \quad \text{where } M_i = 1 \tag{4.25}$$

In order to determine if the algorithm has converged, we used the ℓ_2 norm of the primal and dual residuals, $||r||_2$ and $||s||_2$ as suggested in [160]. Thus,

$$||r^{k+1}||_2 \leftarrow [||\rho^k(Dv^{k+1} - z_1^{k+1})||_2^2 + ||\rho^k(v^{k+1} - z_2^{k+1})||_2^2]^{1/2} \tag{4.26}$$

$$||s^{k+1}||_2 \leftarrow ||\rho^k(D^T(z_1^k - z_1^{k+1}) + z_2^k - z_2^{k+1})||_2 \tag{4.27}$$

The algorithm has converged when both$||r||_2 < \epsilon^{pri}$ and $||s||_2 < \epsilon^{dual}$. ϵ^{pri} and ϵ^{dual} are computed as

$$\begin{aligned} \epsilon^{pri} &\leftarrow p^{1/2}\epsilon^{abs} \\ &+ \epsilon^{rel} max\{[||v^{k+1}||_2^2 + ||Dv^{k+1}||_2^2]^{1/2}, [||z_1^{k+1}||_2^2 + ||z_2^{k+1}||_2^2]^{1/2}\} \end{aligned} \tag{4.28}$$

$$\epsilon^{dual} \leftarrow n^{1/2}\epsilon^{abs} + \epsilon^{rel}||D^T\mu_1^{k+1} - \mu_2^{k+1}||_2, \tag{4.29}$$

where p is the number of elements in z_1 and z_2, n is the number of elements in v, ϵ^{abs} is an absolute tolerance, and ϵ^{rel} is the relative tolerance.

The quantities $||r||_2$ and $||s||_2$ are used to dynamically update the penalty parameter, ρ, if the difference between the primal and dual residuals becomes large. A larger value for the penalty parameter, ρ, causes a violation of the constraints to cost more

in the objective function. A smaller value for ρ has the opposite effect. This dynamic updating can speed up convergence.

If the primal residual is much larger than the dual residual, $||r^{k+1}||_2 > \tau||s^{k+1}||_2$, then:

$$\rho^{k+1} \leftarrow \gamma\rho^k \tag{4.30}$$

$$\mu_1^{k+1} \leftarrow \frac{1}{\gamma}\mu_1^{k+1} \tag{4.31}$$

$$\mu_2^{k+1} \leftarrow \frac{1}{\gamma}\mu_2^{k+1} \tag{4.32}$$

where τ is the maximum allowed difference in magnitude and γ is the update parameter. If the dual residual is much larger than the primal residual, $||s^{k+1}||_2 > \tau||r^{k+1}||_2$, then:

$$\rho^{k+1} \leftarrow \frac{1}{\gamma}\rho^k \tag{4.33}$$

$$\mu_1^{k+1} \leftarrow \gamma\mu_1^{k+1} \tag{4.34}$$

$$\mu_2^{k+1} \leftarrow \gamma\mu_2^{k+1} \tag{4.35}$$

otherwise, the value for ρ remains the same, $\rho^{k+1} \leftarrow \rho^k$.

The full algorithm is represented by the flowchart in Fig. 4.1. In addition to the steps of ADMM, we subtracted the background from the intensity images. Additionally, we scaled $\mathcal{F}\{A\}$ before any computation by dividing it by the absolute value of its maximum element. After the termination of the algorithm, we divided v by the same value to recover the proper scale. This scaling was performed to prevent issues due to very large

and very small floating-point numbers. Additionally, we apply the constraints from Eqs. (4.24) and (4.25) to ensure that the v returned is subject to the constraints.

4.3 Simulation, Objects, and Evaluation

In order to validate the algorithm, we simulated intensity stacks using the modified split-step beam propagation method (SSBPM) as in [97] and [77]. We compared the results to the TDPM method in [77]. We simulated 512x512x256 intensity stacks from 15 different tomographic angles, equally spaced from $0°$ to $180°$. The objective numerical aperture, NA_o, was 0.75, and the condenser numerical aperture, NA_c, was 0.375. The wavelength of light was 546 nm, and the refractive index of the oil, n_c, was 1.458. For evaluating ADMM-TDPM, only three angles, $0°$, $60°$, and $120°$, were used, while 15 angles from $0°$ to $180°$ with increments of $12°$ were used for TDPM.

We simulated three different objects: a bead, a mixture of objects, and a modified Shepp-Logan phantom. The bead had a maximum refractive index difference with respect to the oil was 0.04, similarly to the bead used in [164]. The gel in the mixture of objects has a 0.01 refractive index difference between the gel and oil. The beads in the gel have a 0.01, 0.02, 0.03, 0.04, and 0.05 difference between the refractive index of each bead and the gel and are off-center from the axis of rotation. The modified Shepp-Logan phantom has a maximum refractive index difference with respect to the oil of 0.004. We created the modified Shepp-Logan phantom using the phantom3d function from MATLAB's file exchange [138].

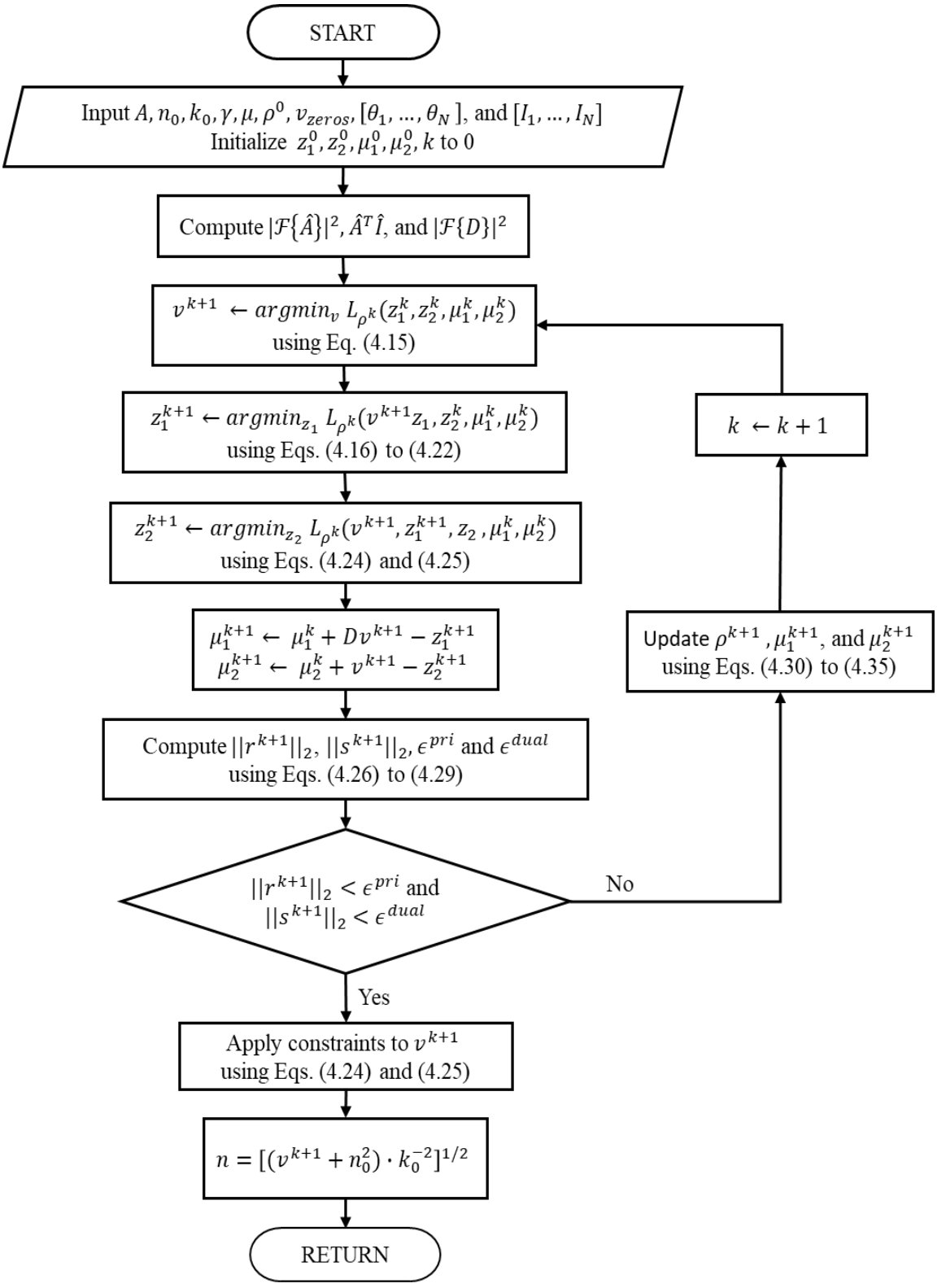

Figure 4.1 – Flowchart for the ADMM-TDPM algorithm.

Similarly to [97], the normalized root-mean-square error (NRMSE) is used to evaluate the performance of the ADMM-TDPM algorithm. We computed the NRMSE as the difference of the recovered refractive index from n_0, $\Delta n_{rec} = n_{rec} - n_0$ and the difference of the ideal refractive index from n_0, $\Delta n_{ideal} = n_{ideal} - n_0$ over each voxel, denoted by the index j.

$$NMRSE(\Delta n_{rec}, \Delta n_{ideal}) = \left[\frac{\sum_i \left[(\Delta n_{rec})_j - (\Delta n_{ideal})_j \right]^2}{\sum_i (\Delta n_{ideal})_j^2} \right]^{1/2} \tag{4.36}$$

Additionally, we report the number of iterations and total time in seconds for each algorithm in Tables 4.1, 4.2, and 4.3. The running time includes loading the intensity data from storage. The number of iterations for TDPM is denoted by a "-" since TDPM is non-iterative. Each algorithm was implemented using MATLAB R2021a on a CPU with an AMD Ryzen 5 5600x processor and 32 GB of RAM.

4.4 Results and Discussion

The results for the bead, the mixture of objects, and the phantom are shown in Tables 4.1, 4.2, and 4.3, respectively. Figures for the cross sections for each axis in each direction for each object and algorithm are shown in Figs. 4.2, 4.3, and 4.4.

TDPM from [77] was simulated using 15 angles for two difference choices of regularization parameter, α. Choosing $\alpha = 10^{-2}$ corresponds to not enough regularization, as can be seen in the artifacts produced in the reconstructions in Figs. 4.3 and 4.4. Choosing $\alpha = 10^{-1}$, on the other hand, removes the reconstruction artifacts but results the in an attenuated refractive index and halo artifacts.

Table 4.1 – Results for the bead object.

Method	Iteration	NRMSE	Elapsed Times(s)
TDPM, $\alpha = 10^{-1}$	-	0.4053	573.6
TDPM, $\alpha = 10^{-2}$	-	0.3612	557.2
ADMM-TDPM, $\gamma = 1$	600	0.3237	1863
ADMM-TDPM, $\gamma = 2$	551	0.3273	1722
ADMM-TDPM constrained, $\gamma = 1$	500	0.3098	3573
ADMM-TDPM constrained, $\gamma = 2$	198	0.3111	644.8

Table 4.2 – Results for the mixture of objects.

Method	Iteration	NRMSE	Elapsed Times(s)
TDPM, $\alpha = 10^{-1}$	-	0.5362	572.5
TDPM, $\alpha = 10^{-2}$	-	0.5192	564.1
ADMM-TDPM, $\gamma = 1$	600	0.7146	1935
ADMM-TDPM, $\gamma = 2$	18	0.7274	89.16
ADMM-TDPM constrained, $\gamma = 1$	600	0.3059	1973
ADMM-TDPM constrained, $\gamma = 2$	41	0.3129	169.5

Table 4.3 – Results for the modified Shepp-Logan phantom.

Method	Iteration	NRMSE	Elapsed Times(s)
TDPM, $\alpha = 10^{-1}$	-	0.5806	565.8
TDPM, $\alpha = 10^{-2}$	-	0.6476	556.3
ADMM-TDPM, $\gamma = 1$	382	0.7191	1222
ADMM-TDPM, $\gamma = 2$	14	0.7738	82.37
ADMM-TDPM constrained, $\gamma = 1$	320	0.1982	1070
ADMM-TDPM constrained, $\gamma = 2$	46	0.203	185.7

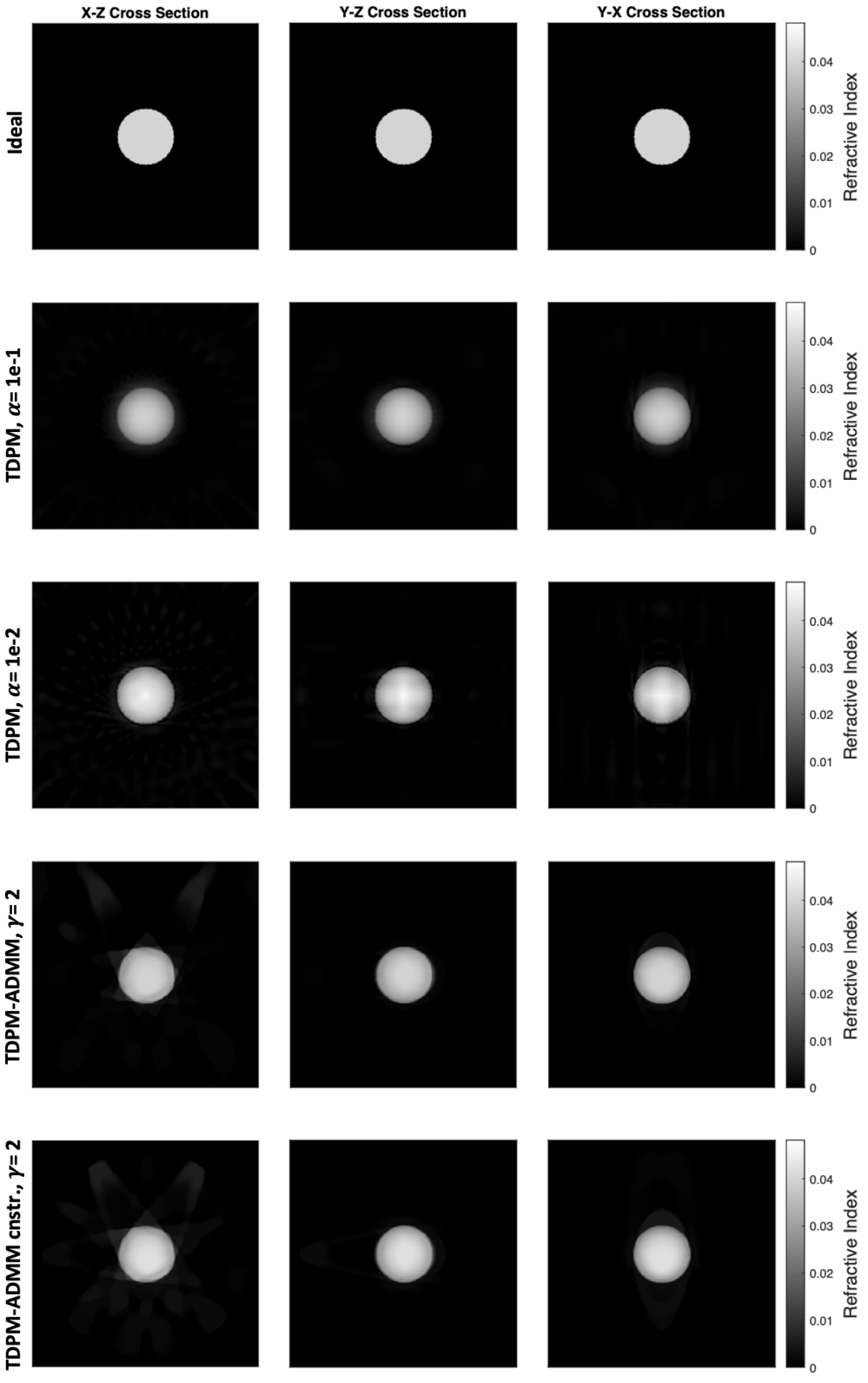

Figure 4.2 – Recovered refractive index for the bead object.

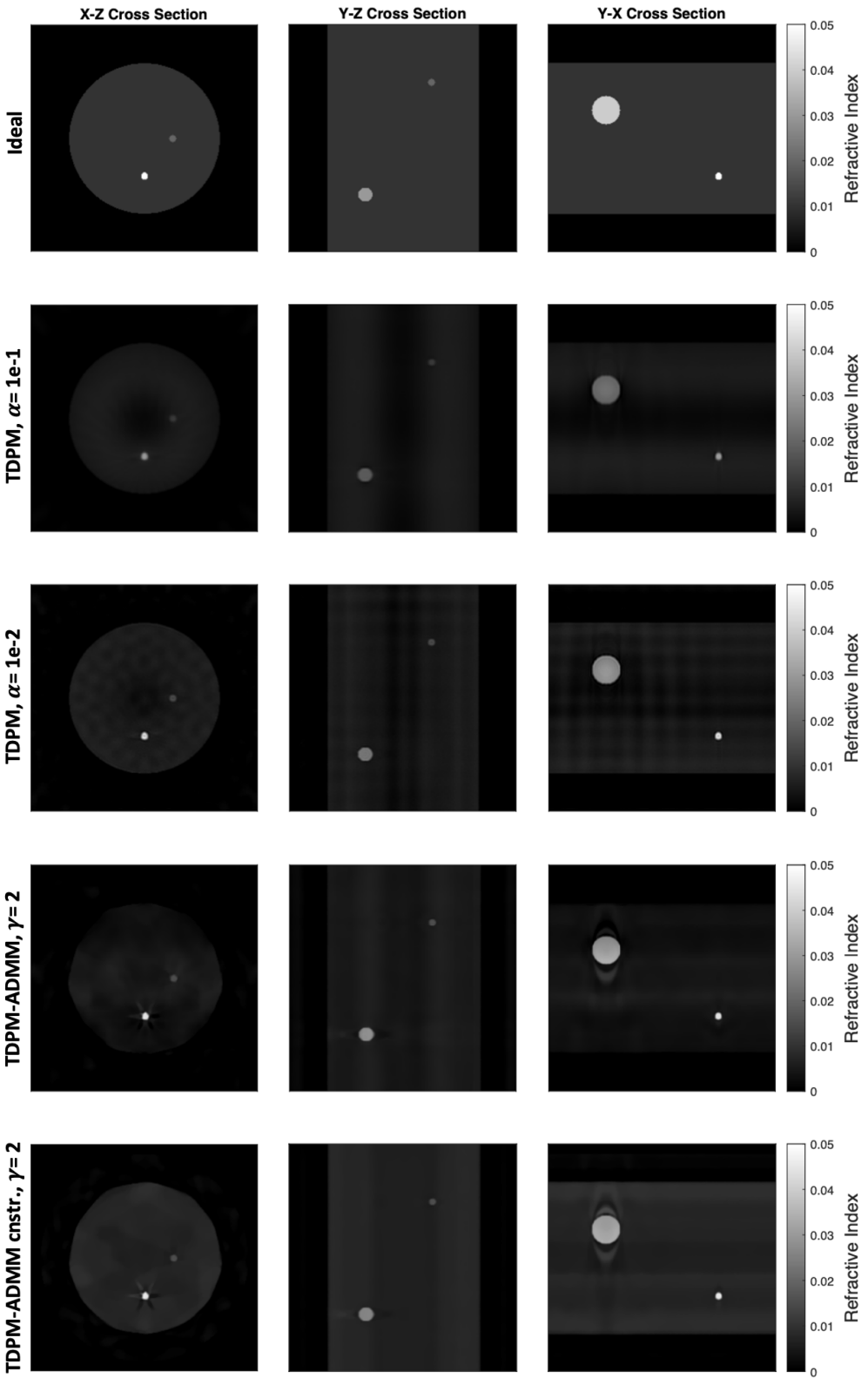

Figure 4.3 – Recovered refractive index for the mixture of objects.

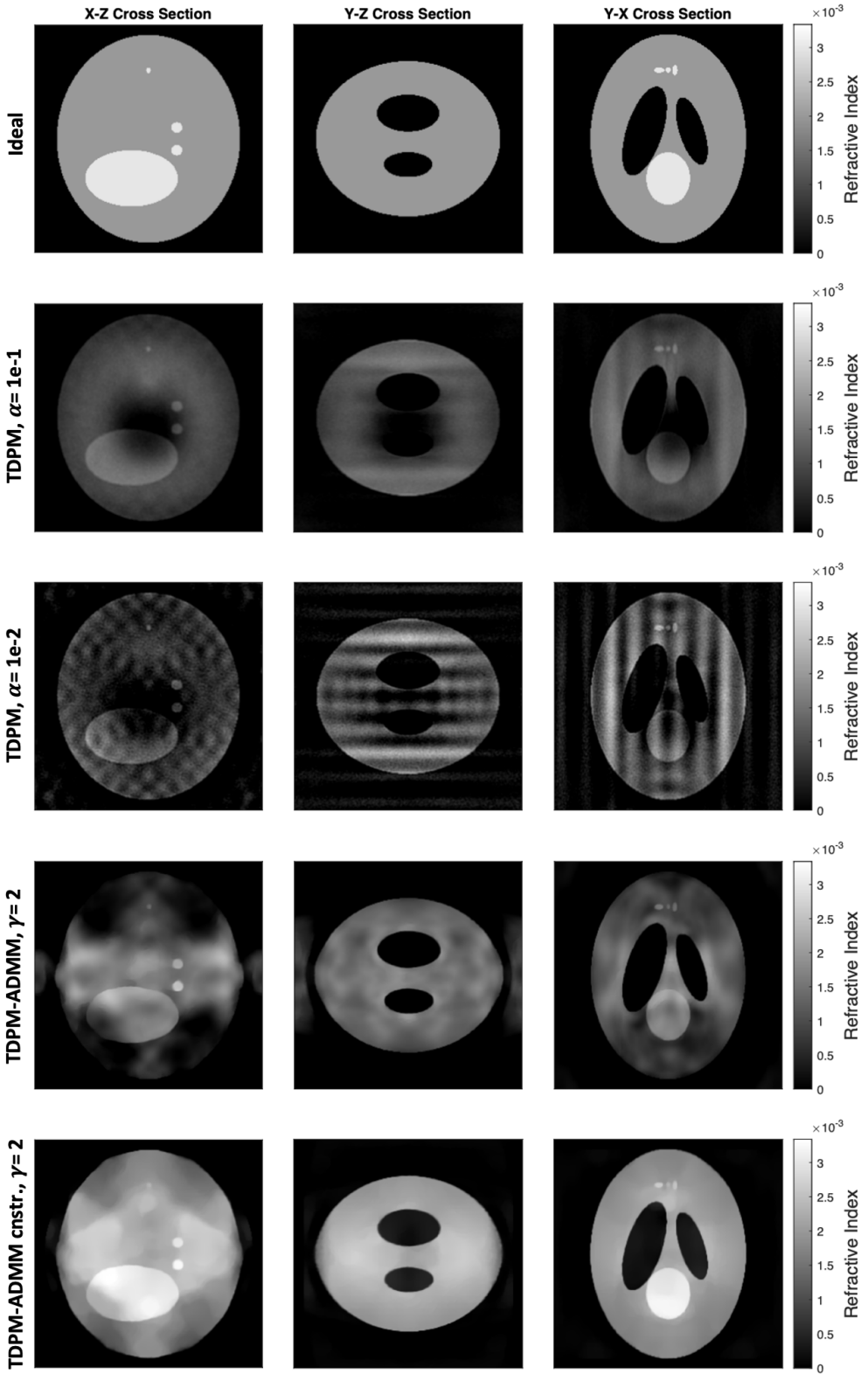

Figure 4.4 – Recovered refractive index for the modified Shepp-Logan phantom.

ADMM-TDPM was simulated with acceleration and constraints. For acceleration, we chose to run the algorithm with and without the update to the penalty parameter, ρ. We also ran it, accelerated and unaccelerated, with and without the constraints. We found the regularization parameter α for each object by searching for it in a smaller problem, 128x128x64. The regularization parameters found for the bead, the mixture of objects, and phantom were α = 0.03171, α = 0.002177, and α = 0.001397, respectively. The parameter found for the bead is higher because the total variation is minimal for the image, with the only gradient being on the surface of the bead. The other hyperparameters used for ADMM-TDPM were ρ_0= 0.6667, τ= 5, $\epsilon^{abs} = 10^{-5}$, and $\epsilon^{rel} = 10^{-3}$. Using M, the data were masked to be zero outside a cylinder within which the objects were located. We evaluated the algorithm using two different values of γ. Selecting of γ = 1 corresponds to no acceleration. Selecting γ = 2 corresponds to scaling ρ by two if the primal and dual residuals are not sufficiently close together, as in Eqs. (4.26) to (4.29). We limited the maximum number of iterations to 600. This limit was reached for the bead object reconstructed with ADMM-TDPM with γ = 1 (Table 4.1) and for the mixture of objects reconstructed with γ = 1 (Table 4.2). The limit was also reached for ADMM-TDPM constrained with γ = 1 (Table 4.2).

ADMM-TDPM with constraints outperformed TDPM using only three angles. TDPM with 15 angles and $\lambda = 10^{-1}$ had an NRMSE of 0.4053, 0.5362, and 0.5806 for the bead, the mixture of objects, and the phantom. ADMM-TDPM constrained with three angles and γ = 2 had an NRMSE of 0.3111, 0.3129, and 0.203 for the three objects. While using the acceleration resulted in a slightly higher NRMSE, it also resulted in speedups of 5.54x, 11.64x, and 5.78x for the bead, the mixture of objects, and phantom for the

constrained version of ADMM-TDPM. This speedup is significant and suggests that the acceleration should be used despite the slight increase in error. ADMM-TDPM unconstrained with three angles and $\gamma = 2$ had an NRMSE of 0.3273, 0.7274, and 0.7738, which shows that including the non-negativity and known zero constraints significantly improve the recovered image.

Even though ADMM-TDPM is iterative, the algorithm was faster than TDPM for the mixture of objects and phantom for the accelerated version. This is because loading the data for angles from storage and computing $\hat{A}^T\hat{I}$ grows linearly as angles are added. Since ADMM-TDPM only used three angles, this step of the algorithm is much faster. The algorithm took longer than the bead because the regularization parameter was larger, which makes the objective less like a quadratic and more difficult to minimize.

In Figs. 4.2 and 4.4 the x-z cross section is notably worse for the bead and phantom objects when using ADMM-TDPM. This is because the y-direction is the axis of rotation.

4.5 Summary

ADMM-TDPM with the Augmented Lagrangian Method has been applied to reconstruct 3D microscopic phase images. The optimizations of the data fidelity by minimizing Gaussian noise and the scattering potential through total variation regularization with the constraints of non-negativity and known zeros have been performed to reconstruct 3D RI distributions from the intensity images of three angles. The simulation results of reconstructing the mixture of 3D objects and the 3D modified Shepp-Logan phantom demonstrate that ADMM-TDPM can be applied to shift-variant objects such as FBGs and biological cells. ADMM-TDPM with the non-negativity and known-zeros

constraints achieves significantly faster convergence and smaller error than the original TDPM with 15 angles. ADMM-TDPM has the potential to realize high-resolution real-time 3D imaging with short image acquisition time and fast processing. These results have been submitted for publication to *Applied Optics* [162]

CHAPTER 5. FUTURE WORK

5.1 Real-time TDPM with TSUM

As mentioned in Ch. 3, TSUM has great potential to realize real-time quantitative phase imaging once superior SoCs and APIs for UM are developed. Meanwhile, TSUM can be applied to 3D TDPM with measured data. Unlike simulated data, measured data requires registration steps to align each 2D image in the 3D through-focal images. The registration is done using cross-correlation of two boundary images in the z-direction (along the illumination axis) and symmetry in the x-direction (perpendicular to the rotational axis). In the original MATLAB version of 3D TDPM, the registration is processed sequentially. TSUM can process the registration of each tomographic angle in parallel and accelerate the processing of measured data.

3D TDPM imaging can be divide into three steps: image acquisition, processing, and plotting. The original version of 3D TDPM performs these three steps sequentially. It collects intensity images at 15 different angles from a microscope, computes RID, and plots the results. As indicated in Fig. 5.1, pipelining the three steps could achieve up to a 3x speedup. For example, the intensity data at the first angle are collected, and computing the scattering potential for the first angle can start as soon as the data collection is finished. The intensity data at the second angle are collected while the scattering potential for the first angle is being computed. Converting the scattering potential to RID and rendering it can follow the processing at each angle, but it may improve the performance if it waits until scattering potentials from all angles are computed. Once all 15 angles are computed, a complete RID can be rendered. Moreover, the pipeline can continue to update the RID.

Time-variant objects, like live biological cells, can potentially be continuously investigated using the pipelined TDPM.

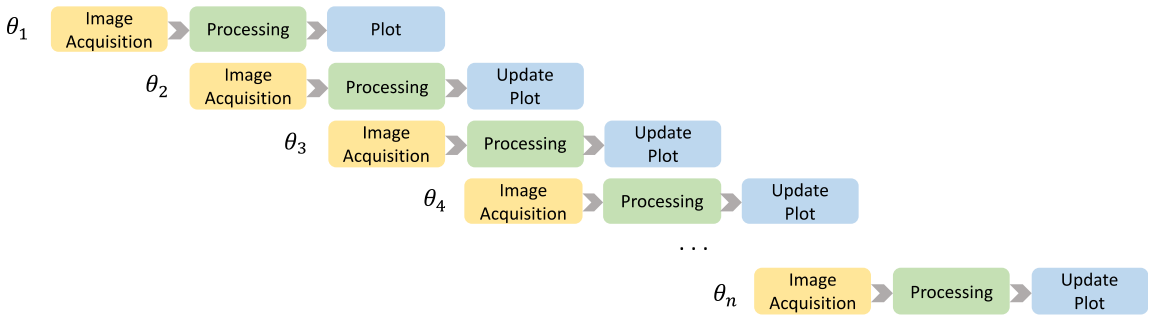

Figure 5.1 – 3D TDPM pipeline for real-time imaging.

5.2 Real-time imaging with ADMM-TDPM

ADMM-TDPM can be implemented for measured experimental data. The simulation results in Ch. 4 show ADMM-TDPM can achieve higher accuracy with three angles than the original TDPM can with 15 angles. Pipelining can also be applied to ADMM-TDPM for continuous imaging as shown in Fig 5.2. It can collect the first set of three angles (e.g., 0°, 60°, and 120°), compute RID with ADMM, and render the output. While RID are computed for the first three angles, the next set of three angles (e.g., 12°, 72°, and 132°) can be collected. Once the RIDs of both sets of angles are computed, the two outputs may be combined and rendered.

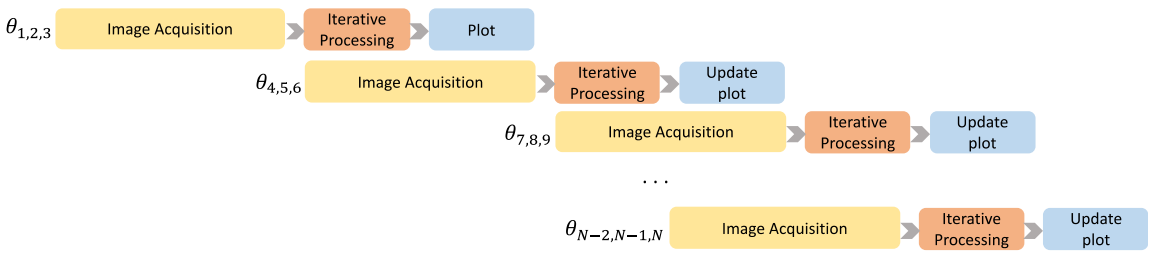

Figure 5.2 – 3D ITDPM pipeline for real-time imaging.

5.3 Real-time imaging with ADMM-TDPM-TSUM

ADMM-TDPM could be accelerated using a GPU. As ADMM-TDPM is an iterative method, an input of an iteration depends on the output of the previous iteration. Thus, the iteration loop cannot be parallelized, and a considerable speedup is not expected from GPU computing. However, arrays that do not have dependencies in each iteration can be computed in parallel. TSUM can be applied to these independent array computations in each iteration. For example, updating the Lagrange multipliers, z_1, z_1, μ_1, and μ_2, as well as computing of primal and dual residuals can be parallelized. Utilizing a UPM eliminates the data transfer overhead; thus, the speedup can be significant when the arrays are large.

APPENDIX A. TDPM 3D MATLAB 1.0 USER MANUAL

A.1 Introduction

The purpose of this manual is to provide instructions on the use of the 3D tomographic deconvolution phase microscopy (TDPM) MATLAB program developed by Micah Jenkins and Yijun Bao to reconstruct the 3D refractive-index distributions (RIDs) of optical fibers or capillaries. Flowcharts in A.2 represents the sequence of algorithms in the two main script files, TDPM_3D_measure_complete.m and TDPM_3D_simulate_complete.m. A.3 offers detailed explanations on parameters and algorithms in TDPM_3D_measure_complete.m, TDPM_3D_simulate_complete.m, and the two main functions, Idata_3D_from_measure.m and TDPM_from_Idata_3D.m. This manual mainly focuses on capillaries containing microspheres. However, the instructions can also be a guide for optical fibers with different parameter settings. For a first-time user, A.4 offers the steps to run the main script file, TDPM_3D_measure_complete.m, to recover the RIDs of microspheres in a microcapillary. The algorithms follow the RI recovery methods in Micah Jenkins' paper [77] with a few modifications. More details can be found Chapter 2.1 and in Jenkins', Bao's, and Noah's theses [165-167].

A.2 Flowcharts of TDPM 3D

A.2.1 TDPM_3D_measure_complete.m

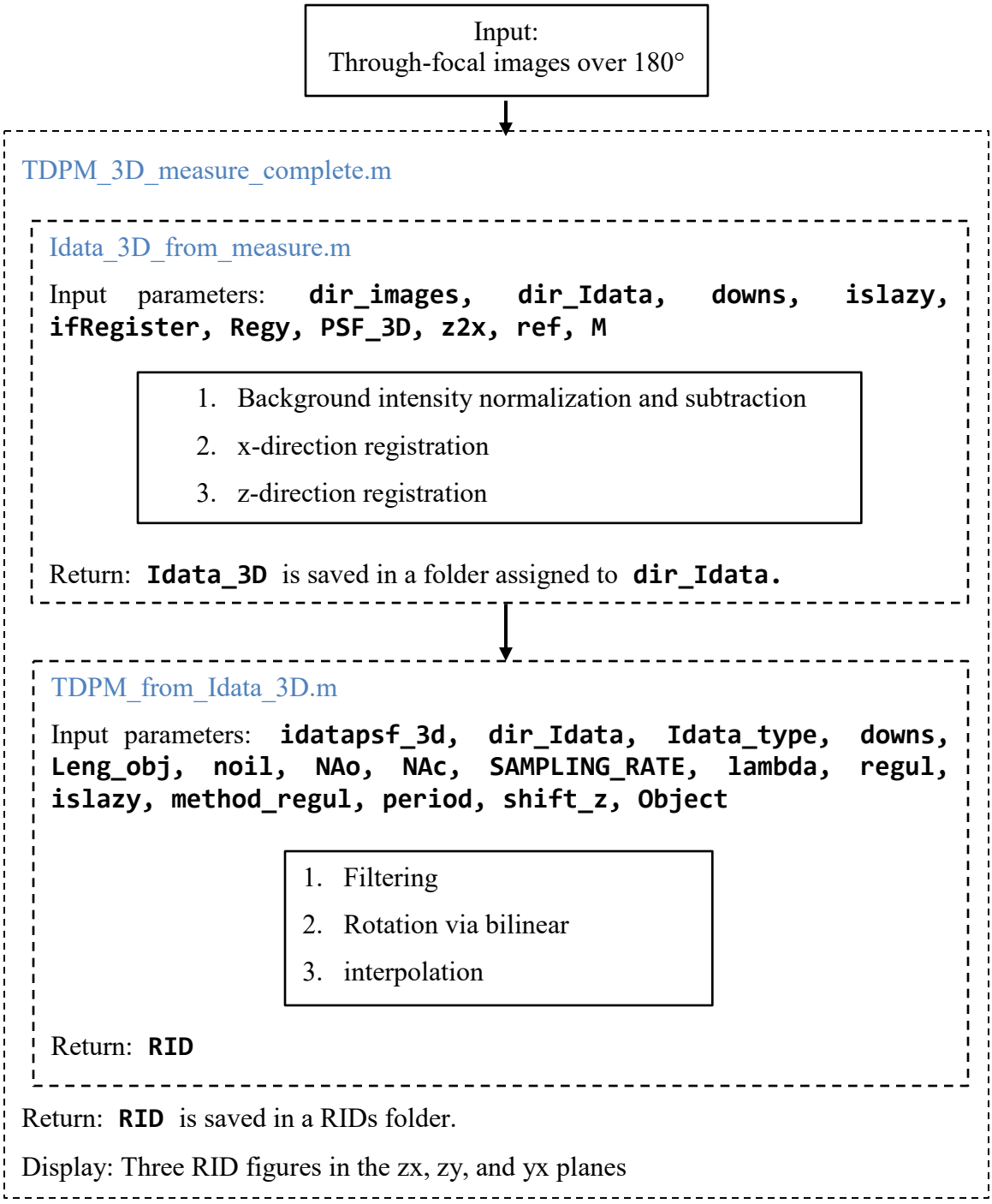

Figure A.1 – Flowchart of TDPM_3D_measure_complete.m

A.2.2 TDPM_3D_simulate_complete.m

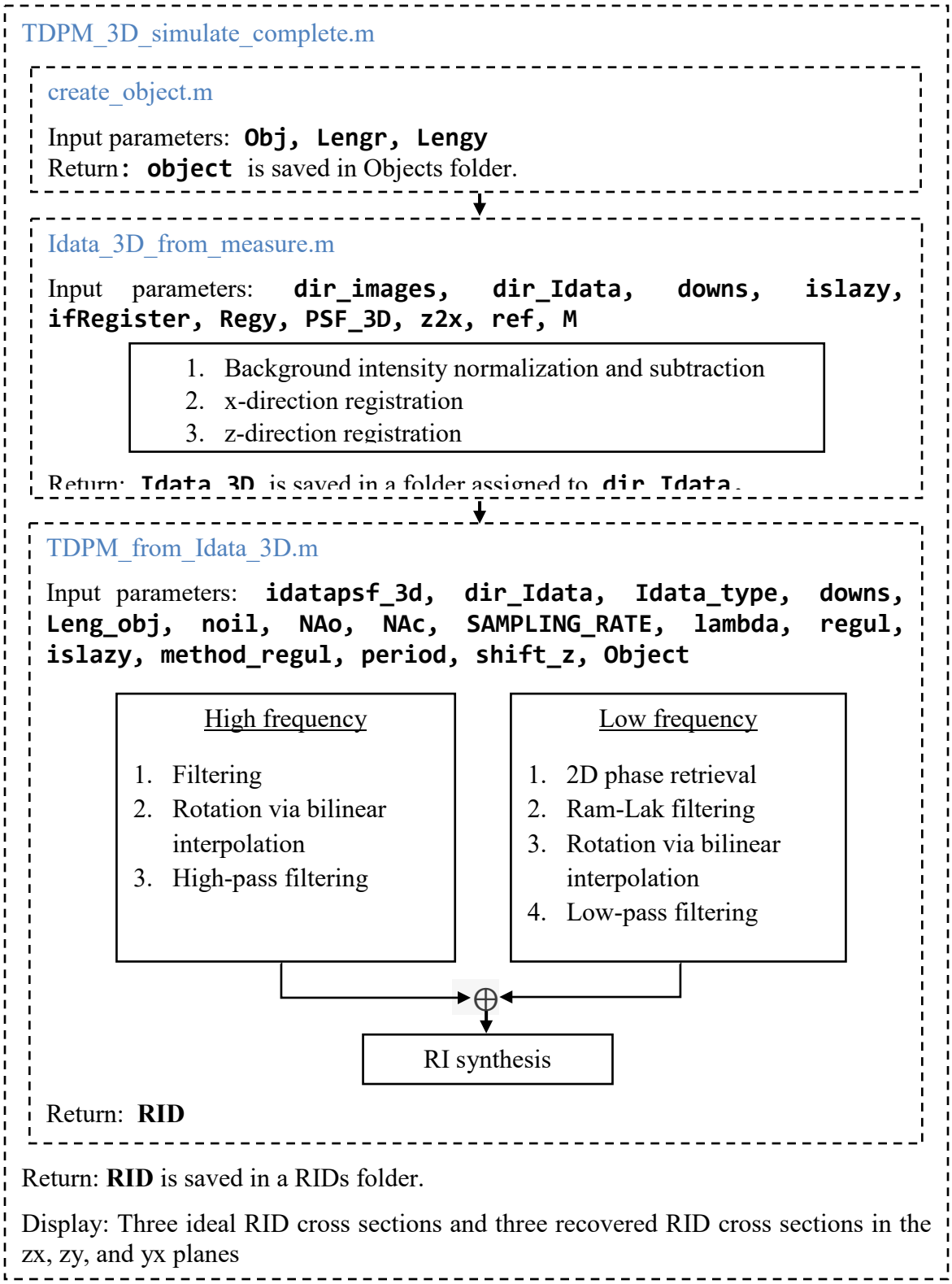

Figure A.2 – Flowchart of TDPM_3D_simulate_complete.m

A.3 Main Files

A.3.1 Main Script File TDPM_3D_measure_complete.m

TDPM_3D_measure_complete.m processes the entire 3D TDPM recovery, including reading measured 3D intensities, calculating or loading 3D PSF, and TDPM recovery.

- (Line 3 – 66) Parameter settings (heading)
  - **run_Idata_3D** = true to generate **Idata_3D** or false to load existing **Idata_3D**
  - **Object** = Type of object
    - 'SMF', 'PMF', 'PCF', 'LPFG', 'FBG' are for various fibers.
    - 'mix' contains capillary, gel, and microsphere.
    - 'gel' contains capillary and gel.
    - 'spheres' contains microspheres.
  - **noil** = Refractive index of immersion oil
  - **NAo** = The numerical aperture of an objective lens
  - **NAc** = The numerical aperture of a condenser lens
  - **NAci** = The inner numerical aperture of a condenser lens for annular source
  - **M** = Magnification of the objective lens
  - **SAMPLING_RATE** = Effective pixel size of a camera
  - **lambda** = Wavelength of a light source
  - **downs** = Downsampling ratio
  - **Leng** = Length of the object
  - **regul** = Regularization parameter

- **psf_type** = A type of PSF (point spread function; inverse Fourier transform of POTF, phase optical transfer function)
  - 'analytical' is from a rotation of 2D POTD calculated analytically.
  - 'SSBPM_Gaus' is from 3D SSBPM with Gaussian source.
- **source_type** = Type of source function
  - 'disk' is for disk source.
  - 'annular' is for annular source.
  - 'Gaus' is for Gaussian source.
- **Method_regul** = regularization method
  - 'Wiener' uses Wiener filter and provides spatially smoother results.
  - 'hard' uses a hard cutoff and provides more accurate results for acceptable frequency.
  - 'mix' uses Wiener filter for low-frequency part and the hard cutoff for high-frequency part. It should be chosen only for FBG.
- **shift_z** = Manual shift of the z direction in pixels
- **z2x** = Ratio between Delta_z and Delta_x where Delta_z is the distance between the neighboring through-focal images, and Delta_x is the pixel size of camera. z2x = 1 is used for FBG, and z2x = M/10 is used for other objects.
- **dir_Idata** = Directory of a folder to store intensity data
- **islazy** = Use of rotation angles
  - 'lazy' uses a single angle (the first angle) of data, assuming images are the same for every rotation angle. It can be used for a single-mode fiber or an empty capillary.

- ‘full’ uses different data for every rotation angle.

- **forder_measure** = Directory of a folder to store 3D intensity
- **ifRegister** = Registration method
  - ‘RegC’ uses cross-correlation with two boundary images (for fiber measurements).
  - ‘RegC4’ uses cross-correlation with two boundary images and 2x2 least squares fitting (for fiber measurements).
  - ‘RegCall’ uses cross-correlation with a full image.
  - ‘RegCall3’ uses 3D cross-correlation with an entire 3D image.
  - ‘RegCS’ uses cross-correlation with two boundary images in the z-direction and symmetry in the x-direction (The current best method for cell measurement).
  - ‘RegCS4’ uses cross-correlation with two boundary images and 2x2 least squares fitting in the z-direction and symmetry in the x-direction.
  - ‘RegCSall’ uses cross-correlation with a full image in the z-direction and symmetry in the x-direction.
  - ‘RegSSIM3’ uses 3D SSIM.
  - ‘RegS’ uses the symmetry of intensity times height (I x h) for the z-direction and intensity for the x-direction.
  - ‘noReg’ uses no registration.
- **Regy** = true if the y-direction is registered, or false
- (Line 55 - 66) Initialization of parameters
  - If the object is ‘mix’, ‘gel’, or ‘spheres’, reference intensity (**ref**) is ‘capillary’

- (Line 67 – 157) Load or Calculate PSF_3D (heading)
  - If the type of PSF (**psf_type**) is 'SSBPM_Gaus', pre-calculated 3D PSF data is loaded according to the magnification of objective lens (**M)** and the chosen downsampling rate (**downs)**.
  - If the type of PSF (**psf_type**) is 'analytical', build_2DOTF_analytical_disk, build_2DOTF_analytical_annular, or build_2DOTF_analytical_Gaus is called according to **source_type** to build 2D PSF (**PSF_2D**) and 2D POTF (**POTF_2D**) in the xz plane. calulate_3d_psf_rotate function calculates 3D PSF (**PSF 3D)** by rotating **POTF_2D** along the z-axis. The aliased pattern in **PSF_3D** is removed.

- (Line 158 – 205) Calculate 3D intensity (heading)
  - The directory of measurement images (**dir_images**) should be specified under the chosen object. If the object is 'gel', rotation is not required. If the object is 'mix', the directory should be specified under the correct **NAc** and **noil**.
  - If **run_Idata_3D** = true in the parameter setting, Idata_3D_from_measure function is called.
- (Line 206 – 218) TDPM recovery (heading)
  - TDPM_from_Idata_3D is called to calculate 3D refractive index distribution (**RID).**
  - If the downsampling rate is 1, the size of **RID** is likely to be larger than 2GB, and the format of **RID** is required to be v7.3 by MATLAB. (Line 219 – 256) Plot the recovered RID cross sections in 3 View angles (heading).

- Refractive index distributions in the zx, zy, and yx planes are plotted.

A.3.2 Main Script File TDPM_3D_simulate_complete.m

TDPM_3D_simulate_complete.m simulates 3D TDPM, including simulating 3D intensities, calculating or loading 3D PSF, and TDPM recovery.

- (Line 3 – 34) Parameter setting (heading)
  - **Run_idata_3D =** true to generate **Idata_3D** or false to load existing **Idata_3D**
  - **noil** = Refractive index of immersion oil
  - **NAo** = The numerical aperture of an objective lens
  - **NAc** = The numerical aperture of a condenser lens
  - **NAci** = The inner numerical aperture of a condenser lens for annular source
  - **M** = Magnification of the objective lens
  - **SAMPLING_RATE** = Effective pixel size of a camera
  - **lambda** = Wavelength of a light source
  - **Obj** = type of simulation object
    - ‘mix’ contains capillary, gel, and microsphere.
    - ‘gel’ contains capillary and gel.
    - ‘spheres’ contains microspheres.
    - ‘squares’, ‘squares2’, ‘diamond’, and ‘diamonds’ are different patterns.
  - **shift_z** = manual shift in the z-direction in pixels
  - **lengr** = length of the object in the x- and z-direction
  - **lengy** = length of the object in the y- direction
  - **islazy** = Use of rotation angles

        - ‘lazy’ uses a single angle (the first angle) of data, assuming images are the same for every rotation angle. It can be used for an empty capillary.
        - ‘full’ uses different data for every rotation angle
    - **regul** = Regularization parameter
    - **Method_regul** = regularization method
        - ‘Wiener’ uses Wiener filter and provides spatially smoother results.
        - ‘hard’ uses a hard cutoff and provides more accurate results for acceptable frequency.
        - ‘mix’ uses Wiener filter for low-frequency part and the hard cutoff for high-frequency part. It should be chosen only for FBG.
    - **psf_type** = A type of PSF (point spread function; inverse Fourier transform of POTF, phase optical transfer function)
        - ‘analytical’ is from a rotation of 2D POTD calculated analytically.
        - ‘SSBPM_Gaus’ is from 3D SSBPM with Gaussian source.
    - **source_type** = Type of source function
        - ‘disk’ is for disk source.
        - ‘annular’ is for annular source.
        - ‘Gaus’ is for Gaussian source.

- (Line 35 - 55) Simulate the intensity images (heading)
    - Create_object function is called to generate a chosen object, **Obj**.
    - If **run_Idata_3D** = true, and intensity data does not exist in the intensities folder, SSBPM_simulate_3D is called to generate **Idata_3D.**

- (Line 57 – 111) load PSF_3D (heading)
  - If the type of PSF (**psf_type**) is 'analytical', build_2DOTF_analytical_disk, build_2DOTF_analytical_annular, or build_2DOTF_analytical_Gaus is called according to **source_type** to build 2D PSF (**PSF_2D**) and 2D POTF (**POTF_2D**) in the xz plane. calulate_3d_psf_rotate function calculates 3D PSF (**PSF 3D)** by rotating **POTF_2D** along the z-axis.
  - If the type of PSF (**psf_type**) is 'SSBPM_Gaus', either pre-calculated 3D PSF data is loaded or POTF_3D_in_TDPM_SSBPM function is called to generate a new **PSF_3D.**
- (Line 112 – 124) TDPM recovery (heading)
  - TDPM_from_Idata_3D is called to calculate 3D refractive index distribution (**RID).**
  - RI of oil (**noil**) is subtracted from **RID**.
- (Line 125 – 130) Calculate errors (heading)
  - Normalized root-mean-square error (**NRMSE**) between the ideal RID and the recovered RID is calculated.
- (Line 131 – 164) Plot the ideal RID cross sections in 3 view angles (heading)
  - Three figures of the ideal RID (**object_center**) in the zx, zy, and yx planes are displayed.
- (Line 165 – 197) Plot the recovered RID cross sections in 3 view angles (heading)
  - Three figures of the recovered RID (**RID_small**) in the zx, zy, and yx planes are displayed.

A.3.3 Major Function File Idata_3D_from_measure.m

Function Idata_3D_from_measure(dir_images, dir_Idata, downs, islazy, ifRegister, Regy, PSF_3D, z2x, ref, M)

-- Idata_3D_from_measure.m calculates 3D images for different angles from the measurement capillary data.

-- The objective lens is set to be 50x, the pixel size of the camera (Delta_x) to be 196nm, and the measured object is a capillary.

-- The function takes the following parameters:

- **dir_image** (directory of the folder storing the measured data containing 'image' folder)
- **dir_Idata** (directory of the folder to store the calculated 3D intensity data)
- **downs** (downsampling rate)
- **islazy** ('lazy' or 'full', use of rotation angles)
- **ifRegister** (registration method)
- **Regy** (true if the y-direction is registered, or false)
- **PSF_3D** (3D PSF)
- **z2x** (Delta_z/Delta_x, 1 for FBG measurement and M/10 for other objects)
- **ref** ('fiber' or 'capillary', the reference object used for registration)
- **M** (Magnification of the objective lens)

- (Line 31 – 98) Set parameters (heading)

- The current parameters can be kept the same unless a change is made on purpose.
  - **name** = Name of image files
  - **fmt** = Format of the images
  - **method** = Downsampling method
  - **thetaf** = Measured angles (zero to 168 degree with the increment of 12 decrees)
  - **thetaB** = Background angles (180 degree)
  - **Ltheta** = Total number of angles
  - **stack** = Total number of images in the z-direction (-73:73)
  - **Lxo** = Total pixel of images taken from the camera in the x-direction
  - **Lyo** = Total pixel of images taken from the camera in the y-direction
  - **Lzo** = Number of images in the z-direction (147)
  - **cor1** = The y position of the first registration point
  - **cor2** = The y position of the second registration point
  - (line 90 – 97) **radius** = The radius of fiber or capillary in the unit of pixels
    - The actual inner radius is divided by the camera resolution (**SAMPLING_RATE**).
    - The actual inner radius should be changed if a different capillary is used.

- (Line 99 – 117) Initialization (heading)
  - Arrays for intensity data and vectors used for registration are initialized.

- (Line 118 – 137) Calculate the modulation transfer function (MTF) of the camera (heading)
    - **ff** = Fill factor of the camera
    - **MTF** = Modulation transfer function (See further explanations in [166])

- (Line 138 – 176) Load reference intensity for registration (heading)
    - A pre-generated .mat file is loaded for the reference intensity for different objects and the magnification of the objective lens.
    - Currently, compareall_cap_simu_50x_NAx0.375.mat is used for the capillary, and it contains three variables, **compareall**, **compared**, and **compareu**.
    - A new reference intensity should be generated for a different magnification or a different object using TDPM_2D_simulate_complete.m and create_compareall.m in TDPM 2D folder.

- (Line 177 – 194) Background processing (heading)
    - Background images are processed from ‘image_180_73a’ and ‘image_180_73b’ to ‘image_180_0a’ and stored in **Idata_measBG**.

- (Line 195 – 1344) Processing for different angles (heading)
    - The measured images at different angles are stored in **Idata_meas**.
    - A specified registration method is used. Currently, ‘RegCS’ works the best for the capillary measurement.

- (Line 567 – 709) 'RegCS' uses cross-correlation in the z-direction and symmetry in the x-direction to register.
  - For the x-direction registration
    - Averaged xz cross-sections are selected and resized at y = cor1 and y = cor2.
    - The xz cross-sections are cross-correlated with their flipped upside-down images to find the maximum points and the symmetry axis.
  - For the z-direction registration
    - Assuming the capillary is not tilted in the z-direction, averaged xz cross-section is selected and resized.
    - The best z-direction matches are found using the max cross-correlation between the xz cross-section and the upper and lower edges from the simulated reference intensity **(compareu** and **compared**).
  - Lateral and longitudinal positioning
    - **lat_adjust** and **long_adjust** are the numbers of pixels to be shifted in the x-direction and the z-direction respectively.
    - The images are upsampled by 10 to increase the accuracy of shifting.
    - The images are shifted circularly, but the newly entered columns or rows are assumed to be the same as their nearest neighbor.
- The output data is saved in the folder named intensities.

A.3.4 Major Function File TDPM_from_Idata_3D.m

function [RID] = TDPM_from_Idata_3D(idatapsf_3d, dir_Idata, Idata_type, downs, Leng_obj, noil, NAo, NAc, SAMPLING_RATE, lambda, regul, islazy, method_regul, period, shift_z, Object)

-- TDPM_from_Idata_3D.m calculates 3D refractive index distributions (**RID**) from the intensity data obtained from Idata_3D_from_measure.m.

-- For the measured data, the high spatial frequency recovery method is used for all spatial frequencies. The function divides images into high- and low-frequency regions, but the low-frequency region is null.

-- The function takes the following arguments:

- **idatapsf_3d** (3D PSF calculated in TDPM_3D_measure_complete.m)
- **dir_Idata** (directory of the folder storing the intensity data calculated from Idata_3D_from_measure.m)
- **Idata_type** ('measure' or 'simulate' to indicate how to get **Idata_3D**)
- **downs** (downsampling ratio)
- **Leng_obj** (the length of the object)
- **noil** (refractive index of immersion oil)
- **NAo** (the numerical aperture of an objective lens)
- **NAc** (the numerical aperture of a condenser lens)
- **SAMPLING_RATE** (the effective pixel size of the camera)
- **lambda** (wavelength of a light source)
- **regul** (regularization parameter)
- **islazy** (Use of rotation angles, 'full' or 'lazy')

- **method_regul** (regularization method, ‘Wiener’, ‘hard’, or ‘mix’)
- **period** (the period of grating, only used for FBG)
- **shift_z** (manual shift of the z-direction in pixels)
- **Object** (type of object)

- (Line 30 – 108) Set parameters (heading)
  - The current parameters can be kept the same unless a change is made on purpose or to debug.
    - **doesplot** = Whether to plot the six cross sections (Set it to be true when debugging)
    - **doesclear** = Whether to clear large matrices (Set it to be false when debugging)
    - **LPF** = A constant to eliminate frequencies that are too close to the boundary
    - **scale** = Unit conversion from meter to micrometer
    - **thetaf** = measured angles
    - **Ltheta** = Total number of angles
    - **SAMPLING_RATE** = Camera resolution
    - (line 70 - 74) **radius** = The radius of fiber or capillary in the unit of pixels
      - The actual inner radius is divided by the camera resolution (**SAMPLING_RATE**).
      - The actual inner radius should be changed if a different capillary is used.

- (Line 109 – 187) Initialization (heading)

- The regions of fiber or capillary and immersion liquid are defined.
    - **mask_out** = The region where only oil exists
    - **rhori** = Spatial frequency
    - **ring_i** = The region where low frequency recovery method is used (unit: spatial frequency)
        - The low-frequency method is used inside ring_i, whereas the high-frequency method is used outside ring_i.
        - However, for measurement, ring_i is null.
    - **mask_obj** = The region where the spatial frequency is lower than the maximum spatial frequency allowed by the microscope
    - **mask_obj_small** = The region where the spatial frequency is lower than the maximum spatial frequency allowed by the camera

- (Line 188-211) Low frequency algorithm preparation (heading)
    - **tf1** = 2D POTF
    - **deni =** Inverse of sum of **tf1**

- (Line 212 – 341) High frequency algorithm preparation (heading)
    - **idataphf_3df** = 3D POTF
    - **ifatapsf_3dc** = Auto correlation of 3D POTF
    - **acall** = Sum of all auto correlations
    - **acallf =** Fourier transform of acall
        - A regularization method (**method_regul**) is applied to **acallf**.

        - If **method_regul** = ‘hard’, it sets small POTFs to be zero.
        - If **method_regul** = ‘Wiener’, **acallf_max*regul** (α in [77]) is added to **acallf.**
        - If **method_regul** = ‘mix’, Wiener filtering is applied to low frequency part, and hard cutoff is applied to high frequency part. ‘Mix’ is only used for FBG.
    - **comp** = Inverse of sum of regularized 3D POTFs
    - **acallf_useful** = 3D POTFs for the region where **acallf** is not close to zero

- (Line 342 – 554) Processing for different angles (heading)
    - If **islazy** = ‘full’, the following steps occur at every angle. If **islazy** = ‘lazy’, the following steps occur once at a single angle.
    - The intensity data (**Idata_3D**) from Idata_3D_from_measure.m is loaded and scaled.
    - **Idata_3D** is manually shifted in the z-direction circularly if necessary.
    - High-frequency recovery
        - **Idata_rec1** = Fourier transform of **Idata_3D** for high frequency
        - **Idata_rec1_ex** = **Idata_rec1** padded in the z-direction with repeating boundary values to prevent cropping of the image after rotations
        - **Idata_3D_filt1 = Idata_rec1_ex** rotated by bilinear interpolation
    - Low-frequency recovery
        - **Idata_rec2** = Fourier transform of **Idata_3D** for low frequency

        - **Idata_rec2_ex** = **Idata_rec2** padded in the z-direction with repeating boundary values to prevent cropping of the image after rotations
        - **Idata_3D_filt2 = Idata_rec2_ex** rotated by bilinear interpolation

- (Line 555 – 709) Combination to final result and plot figures (heading)
    - Refractive index distribution is synthesized.
    - **Idata_3D_filt1f** = Fourier transform of **Idata_3D_filt1.**
    - **Vtemp1 =** the scattering potential of high frequency region before filtering
    - **Vtemp1i** = the scattering potential of high frequency region, outside **ring_i**
    - **Idata_3D_filt2f** = Fourier transform of **Idata_3D_filt2.**
    - **Vtemp2** = the scattering potential for the low frequency region before filtering
    - **Vtemp2i** = the scattering potential for the low frequency region, inside **ring_i.** (Null for the measured data)
    - **Vtemp** = the sum of scattering potentials of all regions and frequencies.
    - **RID =** 3D refractive index distribution that is converted from **Vtemp**.
    - If **doesplot** = true, six zx cross sections of different variables are displayed.
        - figure 91 = **Vtemp1**
        - figure 92 = **Vtemp2**
        - figure 93 = **Vtemp1i**
        - figure 94 = **Vtemp2i** (Null for the measured data)
        - figure 95 = **RID - noil**

- figure 96 = **acallf_usefull (**A black pixel in the plot indicates that the value of **acallf** is zero or close to zero. The recovery at this black region may not be reliable and could cause errors.)

A.4 Test Run

Instructions to run **TDPM_3D_measure_complete.m** for microspheres in a capillary

Note: Boldface indicates variable name.

Single quotes around name indicate a string.

1. First, the measured images from LabVIEW should be stored in folders with the correct names.
   a) One folder (e.g., 4.8.19_NAc0.375_n1.458 in Figure A.3) should contain a folder named ‘images’ which holds the measured images of the capillary with gel and microspheres at the angles from zero to 168 degrees. The images should be from image_0_0a to image_168_73b followed by background images (immersion liquid only) named from image_180_0a to image_180_73a and image_180_73b. (a => above, b => below, 0 => in focus)
   b) Another folder should be named with the refractive index of immersion liquid and a string, ‘No_Spheres_KL’ (e.g., RI_1.458_No_Spheres_KL in Figure A.3).
   It should contain a folder named ‘images’ which holds the images of the capillary with gel only at zero degree named from image_0_0a to image_0_73b and

background images (immersion liquid only) named from image_180_0a to image_180_73a and image_180_73b.

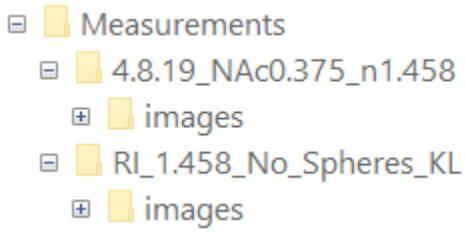

Figure A.3 — An example of the folders storing the measured images

2. TDPM 3D MATLAB code 1.0 is in the Optics O: drive. go to 'O:\JYChun\QPI MATLAB Code 1.0\'. Copy the folder named 'TDPM 3D' to the local disk. TDPM files in the O: drive should not be modified.

   Be sure to have five folders named Intensities, Objects, picture, PSFs, and RIDs in the TDPM 3D folder with TDPM_3D_measure_complete.m. Create them if they are missing.

3. Open **TDPM_3D_measure_complete.m.** in TDPM 3D.

4. The parameters should be specified correctly.

   a) Set **run_Idata_3D** = true. (Line 4) If Idata_3D has already been generated and does not need changes, then **run_Idata_3D** can be set to be false to save the computation time. If Idata_3D has already been generated but needs changes, then **run_Idata_3D** must be set to be true.

b) Choose the object to be ‘mix’. (Line 5)

c) Specify the refractive index of immersion oil, **noil**. (Line 10)

d) Specify the numerical aperture of an objective lens, **NAo**. (Line 11)

e) Specify the numerical aperture of a condenser lens, **NAc**. (Line 12)

f) Specify the magnification of the objective lens, **M**. (Line 14)

g) Choose downsampling rate, **downs**. (Line 17) 2 or 4 are recommended.

h) Specify the directory of the top folder, **folder_measure,** storing the measured images. (Line 37) This is the top folder containing the two folders created in step 1.

   - e.g. folder_measure = 'C:\measurement\';

i) Choose a registration method, **ifRegister**, to be ‘RegCS’. (Line 40)

j) Other parameters should remain unchanged unless the experimental setup has been altered on purpose.

k) Specify the directory of the folder storing the measured images under the correct cases of **object** (‘mix’) and **NAc** (Line 158 – 204) This is the folder created in step 1a.

   - e.g. dir_images = [folder_measure,'4.8.19_NAc0.375_n1.458'];

5. Run TDPM_3D_measure_complete.m.

   - The outputs are three refractive index distributions of the capillary, gel, and microspheres in the zx, zy, and yx planes (Figure A.4).

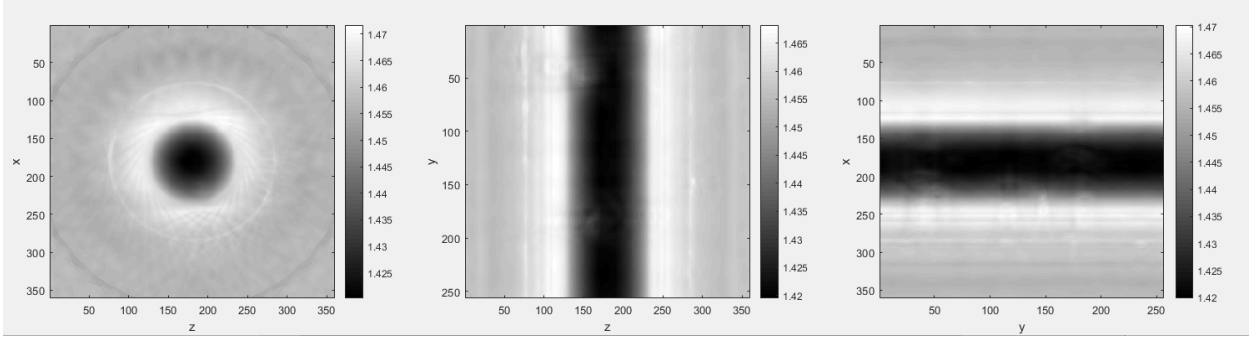

Figure A.4 – The RID cross sections of the capillary, gel, and microspheres (downs=2)

6. Change the object to be ‘gel’ (Line 5) and run TDPM_3D_measure_complete.m.
    - The outputs are three refractive index distributions of the capillary and gel in the zx, zy, and yx planes (Figure A.5).

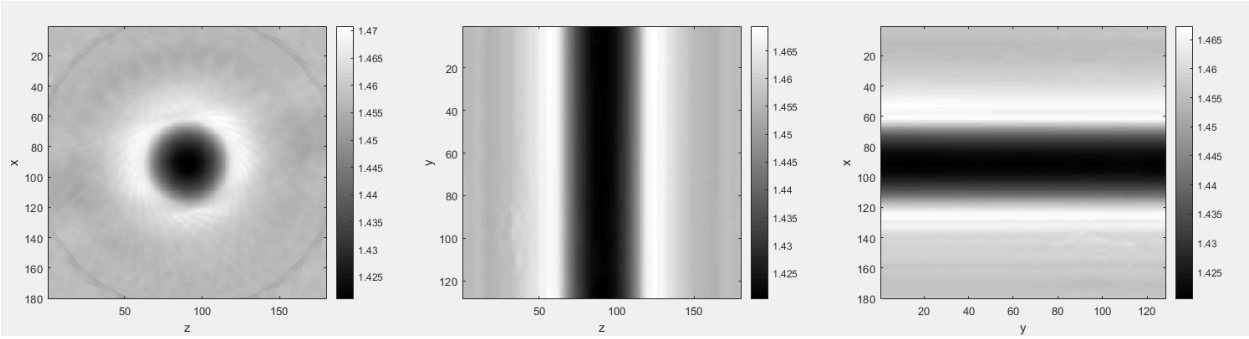

Figure A.5 – The RID cross sections of the capillary and gel (downs=2)

7. Open Idata_3D_diff_gel.m.
    a) Specify the parameters the same as in TDPM_3D_measure_complete.m. (Line 3 – 11)
    b) Run Idata_3D_diff_gel.m.

c) It calculates the 3D intensity difference between the measurements of the capillary with and without microspheres. The outputs are not displayed.

8. Change the object to be ‘spheres’ in TDPM_3D_measure_complete.m (Line 5) and run TDPM_3D_measure_complete.m.
   - The outputs are the three refractive index distributions of microspheres in the zx, zy, and yx planes (Figure A.6).

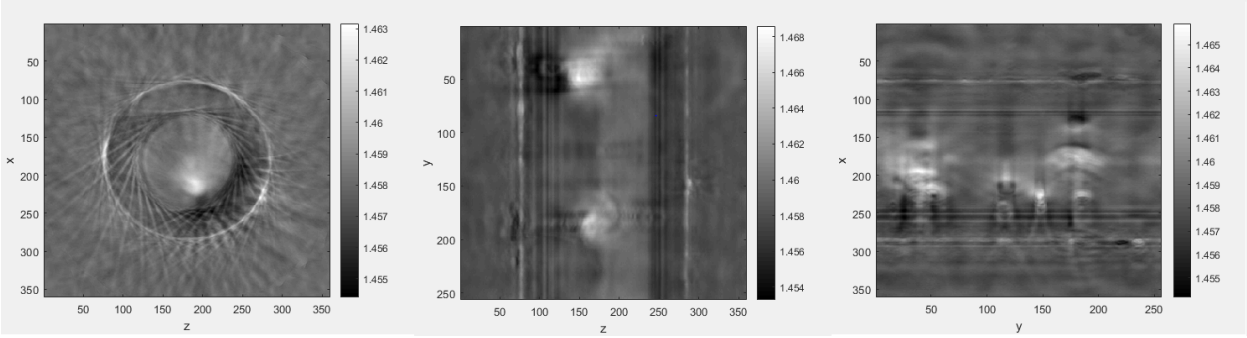

Figure A.6 – The RID cross sections of microspheres (downs=2)

❖ Common errors are caused by a missing file or an incorrect folder directory or name. Be sure to have the required folders in the correct locations.

A.5 List of TDPM 3D Files

A.5.1 .m (script)

do_something.m

Run some temporary code, such as plotting.

do_many.m:

Run some code using various parameters. I often use a series of loops to run a function with different parameters. If necessary, the scripts can also be converted to functions.

TDPM_3D_simulate_complete.m

Do the entire 3D TDPM simulation process, including simulating 3D intensities, calculating or loading 3D PSF, and TDPM recovery.

TDPM_3D_measure_complete.m

Do the entire 3D TDPM recovery process, including reading measured 3D intensities, calculating or loading 3D PSF, and TDPM recovery.

Idata_3D_diff_gel.m

Calculate 3D intensity difference between measurement of capillary with and without microspheres. For convenience, TDPM_3D_measure_complete can be the last sentence of this script to run TDPM recovery in one script.

Idata_3D_diff_simu.m

Calculate 3D intensity difference between simulation of capillary with and without microspheres. For convenience, TDPM_3D_measure_complete can be the last sentence of this script to run TDPM recovery in one script.

Plotting figures:

Check_Idata_3D.m

Show intensity cross section in each angle slice by slice using imshow3D.

view_slice.m

Show 3D RID (can be replaced by intensity) slice by slice using imshow3D.

compare_PSFs_downs.m

Compare PSFs from different downsampling strategies.

A.5.2 .m (function)

create_object.m

Create objects with different parameters, and then store the object in "Objects" folder.

SSBPM_simulate_3D.m

Simulate Idata_3D using a 3D refractive index distribution. Idata_3D is then stored in "Intensity" folder, and input into TDPM_from_Idata_3D.m for the next TDPM recovery.

Idata_3D_from_measure.m

Calculate Idata_3D, which contains the measured cross-sectional intensity data versus fiber rotation angle, using measured images. Idata_3D is then installed in "Intensity" folder, and input into TDPM_from_Idata_3D.m for the next TDPM recovery.

ifRegister has multiple choices. Currently the most accurate result is from 'RegCS', but currently fiber registration only allows 'RegC' and 'RegC4'.

TDPM_from_Idata_3D.m

Recover refractive index distribution using Idata_3D stored in a folder. The input Idata_3D can be got from Idata_3D_TDPM_measure.m using measured images, or from TDPM_simulate_phantom_3D.m from simulated images.

Downsampling:

downsample2_xy.m

Downsample an image in the xy-plane.

downsample2_xz.m

Downsample an image in the xz-plane.

downsample3.m

Downsample an image in 3D.

upsample3.m

Upsample a 3D PSF using interp3.

calculating 3D PSF:

rotate223.m

Rotate a 2d object axially to a 3d object using interp1. Notice that only the POTF can be rotated. PSF cannot be rotated.

calculate_3d_psf.m

calculate 3D PSF from 2D PSF or POTF by rotating 2D POTF. Downsampling is allowed.

POTF_3D_in_TDPM_SSBPM.m:

Performs a simulation on a central point scatterer to generate the 3D intensity (PSF_3D), and also may account for spherical aberration away from focus. Based on SSBPM described by Eq. (23) and (24) in Jenkins_2015b. If the point RI is replaced by an object, this function can be used to simulate TDPM.

build_2DOTF_analytical_disk.m

build_2DOTF_analytical_annular.m

build_2DOTF_analytical_Gaus.m

Builds the 3D phase or absorption optical transfer function (3D POTF or AOTF) in the kx-kz plane for a 2D result. Based on implementing analytical equations. A disk / annular / Gaussian source is used. They are copied from 2D QPI folder.

POTF_2D_in_TDPM_SSBPM.m:

Performs a simulation on a central line scatterer to generate the cross-sectional intensity (idatapsf). Copied from TDPM_2D folder.

Others:

ssim.m

ssim_wang.m

Computing SSIM of two images. ssim is the version introduced in MATLAB R2014a and is currently used. ssim_wang is an older version and is not currently used.

A.5.3 .mat

Grouped by formats. Some terms may be missing, which usually means default values.

Registration references:

compareu.mat:

compared.mat:

compareall_50X_NAc0.5.mat:

2D intensity array based on SSBPM simulations of fiber edges that are used for edge detection for registration. compareu is the upper part of the fiber edge. compared is the lower part of the fiber edge. compareall is the entire xz cross section. Previous three mats are based on 40X objective and NAc=0.375. compareall_capillary is based on 50X objective and NAc=0.5.

compareall_cap_simu_50X_NAc0.375.mat:

2D intensity array based on SSBPM simulations of capillary edges that are used for edge detection for registration. It is based on 50X objective and NAc=0.375.

compareall_cap_exp_50X_NAc0.375.mat:

2D intensity array based on experimental measurement of capillary edges that are used for edge detection for registration. It is based on 50X objective and NAc=0.375.

Sources:

source_Gaussion.mat:

Gaussian fitted source distribution S(**ρ'**). source_Gaussion.mat is centered.

source_Gaus_NAc(NAc).mat

Gaussian fitted source distribution with NAc. Both the fitted image and the fitted numbers are stored.

Objects to be simulated (in Objects folder):

(Object)_(Lengr)x(Lengy).mat

Simulated objects with size Lengr x Lengy x Lengr.

PSFs or POTFs calculated or simulated (in PSFs folder):

By default, SAMPLING_RATE=245e-9, NAc=0.375, NAo=0.75, lambda=546e-9.

PSF_2D_SSBPM_Guas_correct_scale.mat (avoid)

PSF and POTF simulated from SSBPM used for 2D TDPM. The source type is fitted Gaussian.

PSF_3D_256.mat (avoid)

PSF calculated by rotating and downsampling 2D POTF (rotate223_full_downs.m and PSF_2D_SSBPM_Guas_correct_scale.mat). This is used for experimental recovery when downsampling ratio is 4.

PSF_3D_256_simu_downs1.mat

PSF_3D_256_simu_downs4.mat (avoid)

PSF calculated by SSBPM simulation. Different downsampling ratios are used. downs=1 is used for simulation only, because it retains the camera resolution. downs=4 is used for experimental recovery, because it retains the physical length of the object.

PSF_3D_SSBPM_Gaus_downs(downs).mat

PSF calculated from SSBPM simulation with Gaussian source. It is used in experimental recovery.

PSF_3D_(Source_type)_(Leng)_ NAc_(NAc)_n(noil)_downs(downs).mat

PSF_3D_(Source_type)_(Leng)_

NAc_(NAc)_n(noil)_dx(SAMPLING_RATE)_downs(downs).mat

Computed PSF from rotating analytical 3D POTF with parameters in name.

PSF_3D_(Source_type)_SSBPM_(Leng)_dx(SAMPLING_RATE)_n(noil)

_lambda(lambda)_NAo(NAo)_NAc(NAc).mat

PSF_3D_(Source_type)_SSBPM_(Leng)_n(noil)_downs(downs)_dx(SAMPLING_R

ATE) _lambda(lambda)_NAo(NAo)_NAc(NAc).mat

PSF_3D_(Source_type)_SSBPM_(Leng)_n(noil)_dx(SAMPLING_RATE)

_lambda(lambda)_NAo(NAo)_NAc(NAc)_downs(downs).mat

Simulated PSF from SSBPM with parameters in name.

Simulated defocused images (in Intensity folder):

[folders] (Object)_(Lengr)x(Lengy)_n(noil)

[folders] (Object)_shift(zshift)_(Lengr)x(Lengy)_n(noil)

Simulated 3D intensity images

[folders] (Object)_(Leng)_NAc(NAc)_n(noil)

[folders] (Object)_(Leng)_NAc(NAc)_n(noil)_(isRegister)

Experimental 3D intensity images

Object: 'SMF', 'PMF', 'PCF' are various fibers; 'mix' is microspheres in capillary; 'gel' is capillary without microspheres; 'spheres' is the difference between 'mix' and 'gel', equivalent to only microspheres.

Recovered refractive index (in RIDs folder):

PMF144.mat

SMF144.mat

PCF144.mat

These stores the experimental refractive index and for different fibers, recovered by TDPM.

RID_(Object)_shift(zshift)_(Lengr)x(Lengy)_n(noil).mat

Recovered refractive index of (Object) from simulation.

RID_(Object)_(Leng)_NAc(NAc)_n(noil)_(psftype)_shift(zshift).mat

Recovered refractive index of (Object) from experiment.

APPENDIX B. TDPM 3D TSUM 1.0 DOCUMENATION

B.1 Introduction

TDPM3D_TSUM is a 3D tomographic deconvolution phase microscopy (TDPM) program developed in C/C++/CUDA to run specifically on NVIDIA Jetson AGX Xavier utilizing OpenMP Tasking and CUDA Streaming on Unified Memory. TDPM3D_TSUM leverages OpenMP multithreading, CUDA unified virtual memory, and Jetson AGX Xavier unified physical memory to accelerate the reconstruction of 3D refractive index from microscopic quantitative phase images. See Chapter 3 for details.

TDPM3D_TSUM has two major classes, cuMat and TDPM3D, and they are described in B.2 in details. The global functions of TDPM3D_TSUM and their descriptions are in B.3. TDPM3D_TSUM loads simulation objects, point spread functions, intensity data from storage. The data are stored in the HDF5 data format and required to follow a specific naming convention, which is explained in B.4. Compiling and running TDPM3D_TSUM is simple with a makefile. A short instruction on how to run TDPM3D_TSUM is in B.5.

B.2 Classes and Structs

B.2.1 class cuMat

The cuMat class is a data structure for 1D, 2D, and 3D data that is managed by CUDA unified virtual memory. The cuMat data can be real numbers or complex numbers.

The cuMat data are floats and stored as a vector in column-major order followed by row and depth.

Private member variable	
cudaEvent_t	**cudaStat**

Public member variables	
int	**rows**
int	**cols**
int	**depth**
int	**size**
int	**dim**
float2*	**data**
	Pointer for complex numbers allocated with cudaMallocManaged
float*	**rdata**
	Pointer for real numbers allocated with cudaMallocManaged
bool	**isComplex**

Public function members	
constructor	**cuMat()**
	Default constructor: Initiates variables to zero, null pointers, and isComplex to true
destructor	**~cuMat()**
	Calls destroy()
constructor	**cuMat(**int rows, bool isComplex, cudaStream_t stream = NULL**)**
	Creates a 1D cuMat object
constructor	**cuMat(**int rows, int col, bool isComplex, cudaStream_t stream = NULL**)**
	Creates a 2D cuMat object
constructor	**cuMat(**int rows, int cols, int depth, bool isComplex, cudaStream_t stream = NULL**)**
	Creates a 3D cuMat object

constructor	**cuMat(**const cuMat& src, bool copyData, cudaStream_t stream = NULL**)**
	Copy constructor: copies src. If (copy Data == false), it allocates memory, but does not copy data/rdata. If (copyData == true), it allocates memory and copies (deep copy) the data/rdata of src.
cuMat&	**operator= (**const cuMat& other**)**
	= operator overload: copies the input argument, other, including the data/rdata memory address.
void	**cuMalloc(**cuMat& dst, cudaStream_t stream = NULL**)**
	Allocates data/rdata using cudaMallocManaged. If (stream != NULL), it allocates data/rdata and attaches to a specific stream. If (stream == NULL), it allocates data/rdata to global to be accessed by any stream.
unsigned int	**get_rows(**const cuMat& src**)**
	Returns rows
unsigned int	**get_cols(**const cuMat& src**)**
	Returns cols
unsigned int	**get_depth(**const cuMat& src**)**
	Returns depth
unsigned int	**get_size(**const cuMat& src**)**
	Returns size
unsigned int	**get_dim(**const cuMat& src**)**
	Returns dim
void	**C2R()**
	Converts a complex array to a real array by removing imaginary part.
void	**R2C()**
	Converts a real array to a complex array by setting imaginary part equal to zero.
void	**print(**std::string printOption**)**
	Prints cuMat on terminal/console. If (printOption == 'all'), it prints all elements. If (printOption == 'preview'), it prints the first column of data/rdata.
void	**zeros(**int rows, int cols, int depth, bool isComplex, cudaStream_t stream = NULL**)**
	Sets data/rdata values to be zeros.
void	**destroy()**
	calls cudaFree() to free data/rdata memory.

B.2.2 class TDPM3D

Public member variables	
float	**noil**
	Refractive index of immersion oil Default value is 1.458.
float	**NAo**
	Numerical aperture of objective lens Default value is 0.75.
float	**NAc**
	Numerical aperture of condenser lens Default value is 0.375.
float	**Naci**
	Numerical aperture of inner condenser lens for annular source Default value is 0.
int	**M**
	Magnification of objective lens Default value is 50.
float	**SMAPLING_RATE**
	Effective pixel size of camera Default value is 196e-9.
int	**downs**
	Downsampling rate Default value is 1.
float	**lambda**
	Wavelength Default value is 546e-9.
enum class	**Option_Obj{** phantom, mix, gel, spheres, SMF, PMF, PCF, LPFG, FBG **}**
	• phantom: 3D phantom • mix: mix contains capillary, gel, and microspheres • gel: gel contains capillary and gel • spheres: spheres contain only microspheres, which means the difference between 'mix' and 'spheres' • SMF: single mode fiber • PMF: Polarization-maintaining fiber • PCF: Photonic crystal fiber • LPFG: Long-period fiber grating • FBG: Fiber Bragg grating
int	**shift_z**

	Number of pixels to be manually shifted in the z-direction Default value is 0.
int	**Leng**
	Length in the x- and z-direction Default value is 32.
int	**Lengr**
	Length in the x- and z-direction Default value is the same as Leng.
int	**Lengy**
	Length in the y-direction Default value is two times Leng.
enum class	**Option_islazy{** full, lazy **}**
	• full: all rotation angles are used. • lazy: a single angle is used.
	regul
	Regularization parameter Default value is 0.01.
float	**period**
	Default value is 0.
enum class	**Option_method_regul{** Wiener, hard, mix **}**
	The regularization method • Wiener: Wiener filter. It has spatially smoother results • hard: Hard cutoff. It has more accurate results for acceptable frequency • mix: Wiener filter for low frequency part and hard cutoff for high frequency part. It is used only for FBG.
enum class	**Option_psf_type{** analytical, SSBPM_disk **}**
	A type of point spread functions (PSF) • analytical: PSF is calculated analytically from rotation of 2D POTF using a disk, annular, or Gaussian source • SSBPM_disk: PSF is calculated with 3D SSBPM using a disk source
enum class	**Option_source_type{** disk, annular, Gaus **}**
	A type of source functions • disk: a disk source • annular: a annular source • Gaus: a Gaussian source
enum class	**Option_Idata_type{** simulate, measure **}**
	A type of intensity data

	• simulate: Intensity data are generated • measure: Intensity data are from measurements and loaded from storage
Option_Obj	**Obj**
	Default option is phantom.
Option_islazy	**islazy**
	Default option is full.
Option_method_regul	**method_regul**
	Default option is Wiener.
Option_psf_type	**psf_type**
	Default option is analytical.
Option_source_type	**source_type**
	Default option is disk.
Optioin_Idata_type	**Idata_type**
	Default option is simulate.
std::string	**IdataDir**
	Directory for intensity data See B.4 for naming convention.
std::string	**Idata_dataset_name**
	Intensity dataset name in H5 files See B.4 for naming convention.
cuMat	**PSF3D**
	3D point spread function
cuMat	**RID**
	Reconstructed 3D refractive index distribution

Public function members

constructor	**TDPM3D()**
	Default constructor: Initialize the member variables with default values.
destructor	**~TDPM3D()**
	Destructor

B.2.3 struct GPUTimer in cuMat.cuh

Private member variables

cudaEvent_t	**start_**
cudaEvent_t	**stop_**

Public member variable

float	**time**

Public function members

constructor	**GPUTimer()**
	Creates cuda events.
destructor	**~GPUTimer()**
	Destroys cuda events.
void	**start()**
	Records the start time.
void	**stop()**
	Records the end time and print out elapsed time.

B.3 Global Functions

Global functions in cuMat.cu

void	**gpuAssert(**cudaError_t code, const char #file, int line, bool abort**)**
	Asserts that there is no cudaError. If there is an error, exits. It is called by a macro function, cuErrorCheck(ans) { gpuAssert((ans), __FILE__, __LINE__); }
__global__ void	**warmup_kernel()**
	Performs simple addition on GPU.
void	**warmupGPU()**
	Launch a small kernel to warm up GPU.

void	**cuSynchronize(**cudaStream_t stream**)**
	Calls cudaDeviceSynchronize or cudaStreamSynchronize which waits for operations on device or stream finish.
void	**cuCopy2to3(**const cuMat& src, cuMat& dst, int index**)**
	Copies 2D data to 3D data Arguments • src: cuMat object with 2D data to be copied • dst: cuMat object with pre-allocated 3D data • index: index where copy starts in dst
void	**meshgrid(**int xStart, int xEnd, int yStart, int yEnd, cuMat& X, cuMat& Y**)**
	Creates 2D grids that are the same as MATLAB meshgrid output for 2D. It is memory operations and runs on CPU. Arguments: • xStart: starting value in x-axis (row) • xEnd: ending value in x-axis (row) • yStart: starting value in y-axis (column) • yEnd: ending value in y-axis (column) • X: cuMat object with 2D data • Y: cuMat object with 2D data
void	**ndgrid(**int xStart, int xEnd, int yStart, int yEnd, int zStart, int zEnd, cuMat& X, cuMat& Y, cuMat& Z**)**
	Creates 3D grids that are the same as MATLAB ndgrid output for 3D. It is memory operations and runs on CPU. Note: meshgrid and ndgrid has different output formats. Arguments: • xStart: starting value in x-axis (row) • xEnd: ending value in x-axis (row) • yStart: starting value in y-axis (column) • yEnd: ending value in y-axis (column) • yStart: starting value in y-axis (depth) • yEnd: ending value in y-axis (depth) • X: cuMat object with 3D data • Y: cuMat object with 3D data • Z: cuMat object with 3D data
__global__ void	**cuAdd_kernelC(**float2* src1, float2* src2, float2* dst, int n**)**
	Computes the addition of two complex arrays on GPU.
__global__ void	**cuAdd_kernelR(**float* src1, float* src2, float* dst, int n**)**
	Computes the additions= of two real arrays on GPU.

void	**cuAdd(**cuMat& src1, cuMat& src2, cuMat& dst, cudaStream_t stream = NULL**)**
	Launches a kernel to compute the addition of two arrays (dst = src1 + src2). src1, src2, and dst should have the same size and type. Arguments: • src1: input cuMat object • src2: input cuMat object • dst: output cuMat object • stream: CUDA stream. Default value is null.
__global__ void	**cuAdd3_kernelC(**float2* src1, float2* src2, float2* src3, float2* dst, int n**)**
	Computes the additions of three complex arrays on GPU.
__global__ void	**cuAdd3_kernelR(**float* src1, float* src2, float* src3, float* dst, int n**);**
	Computes the additions of three real arrays on GPU.
void	**cuAdd3(**cuMat& src1, cuMat& src2, cuMat& src3, cuMat& dst, cudaStream_t stream = NULL**)**
	Launches a kernel to compute the addition of three arrays (dst = src1 + src2 + src3). src1, src2, src3, and dst should have the same size and type. Arguments: • src1: input cuMat object • src2: input cuMat object • src3: input cuMat object • dst: output cuMat object • stream: CUDA stream. Default value is null.
__global__ void	**cuAddAS_kernelC(**float2* src, float a, float2* dst, int n**)**
	Computes the additions of a complex array and a complex number on GPU.
__global__ void	**cuAddAS_kernelR(**float* src, float a, float* dst, int n**)**
	Computes the additions of a real array and a real number on GPU.
void	**cuAddAS(**cuMat& src, float a, cuMat& dst, cudaStream_t stream = NULL**)**
	Launches a kernel to compute the addition of an array and a float (dst = src + a). a is added to each element of src. src and dst should have the same size and type. Arguments: • src: input cuMat object • a: a real or complex number

	• dst: output cuMat object • stream: CUDA stream. Default value is null.
__global__ void	**cuSubtract_kernelC(**float2* src1, float2* src2, float2* dst, int n**)**
	Computes the subtraction of two complex arrays on GPU.
__global__ void	**cuSubtract_kernelR(**float* src1, float* src2, float* dst, int n**)**
	Computes the subtraction of two real arrays on GPU.
void	**cuSubtract(**cuMat& src1, cuMat& src2, cuMat& dst, cudaStream_t stream = NULL**)**
	Launches a kernel to compute the subtraction of two arrays (dst = src1 - src2). src1, src2, and dst should have the same size and type. The order of input matters. Arguments: • src1: the first input cuMat object • src2: the second input cuMat object • dst: output cuMat object • stream: CUDA stream. Default value is null.
__global__ void	**cuSubtractAS_kernelC(**float2* src, float a, float2* dst, int n**)**
	Computes the subtraction of a complex number from a complex array on GPU.
__global__ void	**cuSubtractAS_kernelR(**float* src, float a, float* dst, int n**)**
	Computes the subtraction of a real number from a real array on GPU.
void	**cuSubtractAS(**cuMat& src, float a, cuMat& dst, cudaStream_t stream = NULL**)**
	Launches a kernel to compute the subtraction of a single number from an array (dst = src - a). a is subtracted from each element of src. src and dst should have the same size and type. Arguments: • src: input cuMat object • a: a real or complex number • dst: output cuMat object • stream: CUDA stream. Default value is null.
__global__ void	**cuSubtractSA_kernelC(**float a, float2* src1, float2* dst, int n**)**
	Computes the subtraction of a complex array from a complex number on GPU.
__global__ void	**cuSubtractSA_kernelR(**float a, float* src1, float* dst, int n**)**
	Computes the subtraction of a real array from a real number on GPU.

void	**cuSubtractSA(**float a, cuMat& src, cuMat& dst, cudaStream_t stream = NULL**)**
	Launches a kernel to compute the subtraction of an array from a single number (dst = a - src). Each element of src is subtracted from a. src and dst should have the same size and type. Arguments: • a: a real or complex number • src: input cuMat object • dst: output cuMat object stream: CUDA stream. Default value is null.
__global__ void	**cuMultiplyAS_kernelC(**float2* src, float a, float2* dst, int n**)**
	Computes the multiplication of a complex array and a real number on GPU.
__global__ void	**cuMultiplyAS_kernelR(**float* src, float a, float* dst, int n**)**
	Computes the multiplication of a real array and a real number on GPU.
void	**cuMultiplyAS(**cuMat& src, float a, cuMat& dst, cudaStream_t stream = NULL**)**
	Launches a kernel to compute the multiplication of an array and a real number (dst = a*src). Each element of src is multiplied by a. src and dst should have the same size and type. Arguments: • src: input cuMat object • a: a real number • dst: output cuMat object • stream: CUDA stream. Default value is null.
__global__ void	**cuMultiplyEE_kernelC(**float2* src1, float2* src2, float2* dst, int n**)**
	Computes the element-wise multiplication of two complex arrays on GPU.
__global__ void	**cuMultiplyEE_kernelCR(**float2* src1, float* src2, float2* dst, int n**)**
	Computes the element-wise multiplication of a complex array and a real array on GPU.
__global__ void	**cuMultiplyEE_kernelR(**float* src1, float* src2, float* dst, int n**)**
	Computes the element-wise multiplication of two real arrays on GPU.
void	**cuMultiplyEE(**cuMat& src1, cuMat& src2, cuMat& dst, cudaStream_t stream = NULL**)**

	Launches a kernel to compute the multiplication of two arrays (dst = src1.*src2). Each element of src is multiplied by a. src and dst should have the same size and type. Arguments: • src1: input cuMat object • src2: input cuMat object • dst: output cuMat object • stream: CUDA stream. Default value is null.
__global__ void	**cuMultiplyEEE_kernelC(**float2* src1, float2* src2, float2* src3, float2* dst, int n**)**
	Computes the element-wise multiplication of three complex arrays on GPU.
__global__ void	**cuMultiplyEEE_kernelR(**float* src1, float* src2, float* src3, float* dst, int n**)**
	Computes the element-wise multiplication of three real arrays on GPU.
__global__ void	**cuMultiplyEEE_kernelCRRC(**float2* src1, float* src2, float* src3, float2* dst, int n**)**
	Computes the element-wise multiplication of one complex array and two real arrays on GPU and outputs a complex array.
__global__ void	**cuMultiplyEEE_kernelRRRC(**float* src1, float* src2, float* src3, float2* dst, int n**)**
	Computes the element-wise multiplication of real complex arrays on GPU and outputs a complex array.
void	**cuMultiplyEEE(**cuMat& src1, cuMat& src2, cuMat& src3, cuMat& dst, cudaStream_t stream = NULL**)**
	Launches a kernel to compute the multiplication of three arrays (dst = src1.*src2.*src3). Each element of src is multiplied by a. src and dst should have the same size. Arguments: • src1: input cuMat object • src2: input cuMat object • src3: input cuMat object • dst: output cuMat object • stream: CUDA stream. Default value is null.
void	**cuMultiplyMM(**cublasHandle_t handle, cublasOperation_t transa, cublasOperation_t transb, const float2 alpha, cuMat& src1, cuMat& src2,

	const float2 beta, cuMat& dst**)**

	Calls the cuBLAS cublasCgemm3m function to perform the 2D matrix-matrix multiplication of two arrays (dst[m,n] = α(src1[m,k])*(src2[m,n]) + β(dst[m,n])). for (dst = src1*src2), alpha should be float2(1,0) and beta should be float2(0,0). See [168] for more details.
	Arguments:
	• handle: handle to the cuBLAS library context
	• transa: operation for A, op(src1)
	  o If (transa == CUBLAS_OP_N), op(src1) = src1
	  o If (transa == CUBLAS_OP_T), op(src1) = $src1^T$ (transpose)
	  o If (transa == CUBLAS_OP_C), op(src1) = $src1^H$ (Hermitian)
	• transb: operation for src2, op(src2). It has the same options as transa.
	• alpha: scalar for multiplication
	• src1: input cuMat object
	• src2: input cuMat object
	• beta: scalar for multiplication
	• dst: in/output cuMat object

	cuDivideAS_kernelC(float2* src, float a, float2* dst, int n**)**
	Computes the division of a complex array by a real number on GPU.
__global__ void	**cuDivideAS_kernelR(**float* src, float a, float* dst, int n**)**
	Computes the division of a real array by a real number on GPU.
void	**cuDivideAS(**cuMat& src, float a, cuMat& dst, cudaStream_t stream = NULL**)**
	Launches a kernel to compute the division of an array by a real number (dst = src1/a). Each element of src is divided by a. src and dst should have the same size and type. Arguments: • src: input cuMat object • a: a real number • dst: output cuMat object • stream: CUDA stream. Default value is null.
__global__ void	**cuAbs_kernelC(**float2* src, float2* dst, int n**)**

	Computes absolute values of a complex array on GPU and outputs a complex array with the imaginary parts equal to zeros.
__global__ void	**cuAbs_kernelCR(**float2* src, float* dst, int n**)**
	Computes absolute values of a complex array on GPU and outputs a real array.
__global__ void	**cuAbs_kernelR(**float* src, float* dst, int n**)**
	Computes absolute values of a real array on GPU.
void	**cuAbs(**cuMat& src, cuMat& dst, bool C2R, cudaStream_t stream = NULL**)**
	Launches a kernel to compute absolute values of an array. Arguments: • src: input cuMat object • dst: output cuMat object • C2R: whether to convert dst to real array ○ If (C2R == true) and dst is a complex array, dst is converted to real. • stream: CUDA stream. Default value is null.
__global__ void	**cuPow_kernelC(**float2* src, const int exp, float2* dst, int n**)**
	Raises a complex array to the power exponent on GPU and outputs a complex array
__global__ void	**cuPow_kernelR(**float* src, const int exp, float* dst, int n**)**
	Raises a real array to the power exponent on GPU and outputs a real array
__global__ void	**cuPow_kernelCR(**float2* src, const int exp, float* dst, int n**)**
	Raises a complex array to the power exponent on GPU and outputs a real array
void	**cuPow(**cuMat& src, cuMat& dst, const int exp, cudaStream_t stream = NULL**)**
	Launches a kernel to raise each element in an input array to the power exponent, exp, (dst = src^(exp)). src and dst should have the same size. Arguments: • src: input cuMat object • dst: output cuMat object • exp: exponent. It can be 2 or 3. • stream: CUDA stream. Default value is null.
__global__ void	**cuSqrt_kernelC(**float2* src, float2* dst, int n**)**
	compute the square root of each element in a complex array on GPU.

__global__ void	**cuSqrt_kernelR(**float* src, float* dst, int n**)**
	Computes the square root of each element in a real array onGPU.
void	**cuSqrt(**cuMat& src, cuMat& dst, cudaStream_t stream = NULL**)**
	Launches a kernel to compute square root of each element in an input array (dst = sqrt(src)). src and dst should have the same size and type. Arguments: • src: input cuMat object • dst: output cuMat object • stream: CUDA stream. Default value is null.
__global__ void	**cuInverseE_kernelC(**float2* src, float2* dst, int n**)**
	Computes the inverse of each element in a complex array on GPU
__global__ void	**cuInverseE_kernelR(**float* src, float* dst, int n**)**
	Computes the inverse of each element in a real array on GPU
void	**cuInverseE(**cuMat& src, cuMat& dst, cudaStream_t stream = NULL**)**
	Launches a kernel to compute the inverse of each element in an input array (dst = 1/src). src and dst should have the same size and type. Arguments: • src: input cuMat object • dst: output cuMat object • stream: CUDA stream. Default value is null
void	**cuReal(**cuMat& src**)**
	Sets the imaginary parts to be zeros. Argument: • src: in/output cuMat complex array To convert to a real array, use C2R().
void	**cuImag(**cuMat& src**)**
	Sets the real parts to be zeros. Argument: • src: in/output cuMat complex array To convert to a complex array, use R2C().
void	**cuConj(**cuMat& src, cuMat& dst**)**
	Computes the complex conjugate of each element in a complex array. Arguments: • src: input cuMat array • dst: output cuMat array

template <unsigned int blockSize> __global__ void	**cuMax_kernel(**float *g_idata, float *g_odata, unsigned int n**)**
	Finds the maximum value in a real array using reduction on GPU. When called, GPU block size should be specified as a template input, and shared memory size should be specified in a kernel launch.
void	**cuMax(**cuMat& src, float* max, cudaStream_t stream = NULL**)**
	Launches a kernel to find the maximum value in a real array using reduction. Arguments: • src: input cuMat object with real data • max: the maximum value to be stored • stream: CUDA stream. Default value is null.
__global__ void	**cuSumAlongDepth_kernelC(**float2* src, float2* dst, const unsigned int row, const unsigned int col, const unsigned int depth**)**
	Sums a complex array along the depth (z-direction) on GPU.
__global__ void	**cuSumAlongDepth_kernelR(**float* src, float* dst, const unsigned int row, const unsigned int col, const unsigned int depth**)**
	Sums a real array along the depth (z-direction) on GPU.
__global__ void	**cuSumAlongDepthReduce_kernel(**float2* src, float2* dst, const unsigned int row, const unsigned int col, const unsigned int depth**)**
	Sums a complex array along the depth (z-direction) using reduction on GPU.
void	**cuSum3to2(**cuMat& src, cuMat& dst, int dim, cudaStream_t stream = NULL**)**

	Launches a kernel to sum an array along the dimension, dim.
	Arguments:
	• src: input cuMat object with 3D data
	• dst: output cuMat object with 2D data
	• dim: the dimension which src is summed up along
	  o 1 = row (currently, not available)
	  o 2 = column (currently, not available)
	  o 3 = depth
	• stream: CUDA stream. Default value is null.

__global__ void	**cuSumAlongDepth_and_cuMultiplyEE_kernelR(**float* src1, float* src2, float* dst, const unsigned int row, const unsigned int col, const unsigned int depth**)**
	Sums a real array along the depth (z-direction) and computes the element-wise multiplication of the sums and another real array on GPU.

__global__ void	**cuSumAlongDepth_and_cuMultiplyEE_kernelRRC(**float* src1, float* src2, float2* dst, const unsigned int row, const unsigned int col, const unsigned int depth**)**
	Sums a real array along the depth (z-direction) and computes the element-wise multiplication of the sums and another real array on GPU and outputs a complex array (the imaginary parts are zeros).
__global__ void	**cuSumAlongDepth_and_cuMultiplyEE_kernelCRC(**float2* src1, float* src2, float2* dst, const unsigned int row, const unsigned int col, const unsigned int depth**)**
	Sums a complex array along the depth (z-direction) and computes the element-wise multiplication of the sums and a real array on GPU and outputs a complex array.
void	**cuSum3to2_and_cuMultiplyEE(**cuMat& src1, cuMat& src2, cuMat& dst, int dim, cudaStream_t stream = NULL**)**

	Launches a kernel to sum an array along the dimension, dim, and compute the element-wise multiplication of the sums and another array.
	Arguments:
	• src1: input cuMat object with 3D data, which is summed up to 2D
	• src2: input cuMat object with 2D data
	• dim: the dimension which src1 is summed up along
	  o 1 = row (currently, not available)
	  o 2 = column (currently, not available)
	  o 3 = depth
	• stream: CUDA stream. Default value is null.

template <unsigned int blockSize> __global__ void	**cuSumAll_kernelC(**float2 *g_idata, float2 *g_odata, unsigned int n**)**
	Sums all elements in a complex array using reduction on GPU. When called, GPU block size should be specified as a template input, and shared memory size should be specified in a kernel launch.
template <unsigned int blockSize> __global__ void	**cuSumAll_kernelR(**float *g_idata, float *g_odata, unsigned int n**)**
	Sums all elements in a real array using reduction on GPU. When called, GPU block size should be specified as a template input, and shared memory size should be specified in a kernel launch.
unsigned int	**nextPow2(**unsigned int x**)**
	Returns the next number that is power of two.
void	**getNumBlocksAndThreads(**int n, int maxBlocks, int maxThreads, int &blocks, int &threads**)**

	Finds appropriate block size and thread size for the device.
void	**cuSumAll(**cuMat& src, float2* sum, cudaStream_t stream = NULL**)**
	Launches a kernel to sum all elements in an array using reduction. Arguments: • src: input cuMat object with real or complex data • sum: output sum. It is a complex number. If src has the real array, the imaginary part of sum it zero. • stream: CUDA stream. Default value is null.
void	**cuMean(**cuMat& src, float* mean, int dim, cudaStream_t stream = NULL**)**
	Computes a mean value of an array along the dimension, dim. Currently, only computing a mean over all elements in src is available. • src: input cuMat object • mean: the output mean of src along dim • dim: dimension to operate along o 0: returns a mean over all elements • stream: CUDA stream. Default value is null.
void	**cufft(**cuMat& src, cuMat& dst, char direction, cudaStream_t stream = NULL**)**

	Performs fast Fourier transform with the cuFFT API [169].
	Arguments:
	• src: input cuMat object with complex data
	• dst: output cuMat object with complex data
	• direction:
	  o 'F': forward FFT
	  o 'I': inverse FFT
	• stream: CUDA stream. Default value is null.

__global__ void	**cuShift2D_kernelC(**float2* src, float2* dst, const unsigned int row, const unsigned col**)**
	Shifts a complex array to have zero-frequency in the center of the array.
__global__ void	**cuShift2D_kernelCR(**float2* src, float* dst, const unsigned int row, const unsigned col**)**
	Shifts a 2D complex array to have zero-frequency in the center of a real array. The imaginary part of the input array is lost.
__global__ void	**cuShift2D_kernelRC(**float* src, float2* dst, const unsigned int row, const unsigned col**)**

	Shifts a 2D real array to have zero-frequency in the center of a complex array. The imaginary part of the output array is zero.
__global__ void	**cuShift2D_kernelR(**float* src, float* dst, const unsigned int row, const unsigned col, const unsigned int depth**)**
	Shifts a 2D real array to have zero-frequency in the center of the array.
__global__ void	**cuShift3D_kernelC(**float2* src, float2* dst, const unsigned int row, const unsigned col, const unsigned int depth**)**
	Shifts a 3D complex array to have zero-frequency in the center of the array.
__global__ void	**cuShift3D_kernelCR(**float2* src, float* dst, const unsigned int row, const unsigned col, const unsigned int depth**)**
	Shifts a 3D complex array to have zero-frequency in the center of a real array. The imaginary part of the input array is lost.
__global__ void	**cuShift3D_kernelRC(**float* src, float2* dst, const unsigned int row, const unsigned col, const unsigned int depth**)**
	Shifts a 3D real array to have zero-frequency in the center of a complex array. The imaginary part of the output array is zero.
__global__ void	**cuShift3D_kernelR(**float* src, float* dst, const unsigned int row, const unsigned col, const unsigned int depth**)**
	Shifts a 3D real array to have zero-frequency in the center of the array.
void	**cuShift(**cuMat& src, cuMat& dst, cudaStream_t stream = NULL**)**
	Shifts an array to have zero-frequency in the center of the array on GPU. src and dst must have different memory addresses. src and dst should have the same size, but can have different types. Currently, only the array with even length in each dimension is accepted, so it can be used to shift the array back. Arguments: • src: input cuMat object **dst: output cuMat object** **stream: CUDA stream. Default value is null.**
void	**shift(**cuMat& src, cuMat& dst**)**
	Shifts an array to have zero-frequency in the center of the array on CPU. src and dst must have different memory addresses. src and dst should have the same size, but can have different types. Currently, only the array with even length in each dimension is accepted. Arguments: • src: input cuMat object

	• dst: output cuMat object
void	**squeeze(**cuMat& src, cuMat& dst, int dim, int plane**)**

	Removes a dimension of a 3D array.
	Arguments:
	• src: input cuMat object with 3D data
	• dst: output cuMat object with 2D data
	• dim: dimension to be removed
	  o 1: row (x)
	  o 2: column (y)
	  o 3: depth (z)
	• plane: plane which remains. plane can be a number in [0, length of row], [0, length of column], or [0, length of depth].
	e.i. squeeze(A, B, 2, 10) is the same as B = squeeze(A(:,10,:)) in MATLAB.

void	**cuRotate(**cuMat& src, cuMat& dst, int angle**)**
	Rotates an array by an angle in a counterclockwise with bilinear interpolation and crops the array to the same size as the input. Arguments: • src: input cuMat object • dst: output cuMat object • angle: rotation angle in degree
void	**reshape(**cuMat& src, cuMat& dst, int shape[3])
	Reshapes an array to a new shape. As data is stored as a vector, it simply redefines the lengths of each dimension. Arguments: • src: input cuMat object • dst: output cuMat object • shape: array of length, 3. [row, column, depth]

Global functions in TDPM3D.cuh

int	**readH5Data**(const H5std_string filename, const H5std_string dataset_name, cuMat& dst**)**
	Reads a HDF5 data file. See [139] for more details. See B.3 for data naming. Arguments: • filename: name of the .h5 file

	• dataset_name: name of dataset in the .h5 file • dst: output cuMat object with preallocated memory for data
void	**loadIdata(**cuMat& Idata, std::string IdataDir, std::string Idata_dataset_name, const int angle**)**
	Loads intensity data stored in .h5 files. See B.3 for data naming. Arguments: • Idata: cuMat object with preallocated memory for data • IdataDir: .h5 file name for intensity data including directory • Idata_dataset_name: dataset name in the .h5 file • angle: rotation angle in degree
__global__ void	**create_mask_out(**float* fxri, float* fzri, float radius, float* mask_out, int n**)**
	Creates mask_out on GPU.
__global__ void	**create_rhori(**float* fxri, float* fyri, float* fzri, float ax, float ay, float az, float* rhori, int n**)**
	Creates rhori on GPU.
__global__ void	**create_mask_out_0(**float* x, float*y, float radius, float* mask_out_0, int n**)**
	Creates mask_out_0 on GPU.
__global__ void	**create_mask_out_0_3D(**float* fxri, float* fzri, float radius, float* mask_out_0_3D, int n**)**
	Creates 3D mask_out_0 on GPU.
__global__ void	**create_final_rhori(**float* fxri, float* fzri, float scale, float* rhori, int n**)**
	Creates rhori on GPU.
__global__ void	**compute_Vtemp1f(**float2* Idata_3D_filt1f, float* ring_i, float* comp, float2* Vtemp1f, int n**)**
	Computes high-frequency scattering potential in frequency domain (Vtemp1f) on GPU.
__global__ void	**convert2RI(**float* Vtemp, float div, float add, float* RID, int n)
	Converts scattering potential (Vtemp) to refractive index distribution (RID) on GPU.
void	**compute_Idata_3D_filts(**cuMat& idatapsf_3df_unshifted_conj, cuMat& tfl_unshifted_conj, cuMat& mask_out_0, cuMat& deni_unshifted, cuMat& Idata_3D_filt1, cuMat& Idata_3D_filt2, int Leng,

	std::string IdataDir, std::string Idata_dataset_name, int rotateDegree, int ndz, int edge, int mask_out_0_count, int startAngle, int endAngle)
	Computes Idata_3D_filt1 and Idata_3D_filt2 from startAngle to endAngle (inclusive) on GPU streams and CPU. startAngle and endAngle are the index of thetaf which can be from 0 to 14. endAngle should be a larger number than startAngle.
void	**TDPM_from_Idata_3D**(TDPM3D& tdpm**)**
	Recovers refractive index distribution (RID) from intensity data.

B.4 Data Naming Convention

TDPM3D_TSUM loads simulation objects, point spread functions (PSF_3D), intensity data (Idata_3D) that are formatted as HDF5 files. It is important to have correct data names for HDF5 files to load them in TDPM3D_TSUM. See [139] to learn how HDF5 file works. For TDPM3D_TSUM, the following naming rules are used.

Intensity Data Names:

File name:

Idata_3D_<object type>_<object size>_n<RI of immersion liquid>.h5

Dataset name:

/ang<angl

Size:

[2 2 1]*(length of object)

Example:

file name: "Idata_3D_phantom_128_n1.485.h5"

Dataset name: "/ang0"

Size: [256 256 128]

PSF Data Names:

File name:

PSF_3D_<source type>_<size>_NAc_<numerical aperture of condenser>_n<RI of immersion liquid>.h5

Dataset name:

/PSF<size>

Size:

[2 2 2]*(length of object) = [1 1 1]*size

Example:

File name: "HDF5 PSF_3D_disk_256_NAc_0.375_n1.458.h5"

Dataset name: "/PSF256"

size: [256 256 256]

Object Data Names:

File name:

<object type>_<length1>x<length2>.h5

Dataset name:

/<object type>

Size:

[1 2 1]*(length of object) = [length1 length2 length1]

Example:

File name: "phantom_256x512.h5"

Dataset name: "/phantom"

size: [256 512 256]

B.5 How to Compile and Run

TDPM3D_TSUM has a makefile which describes how to compile the program and link source files and libraries.

To compile from terminal, simply type:

```
$ make
```

The `Make` command will create object files and an executable file.

To run:

```
$ .\TDPM3D_TSUM <Length>
```

For example, `.\TDPM3D_TSUM 128` will simulate 128x128x64 intensity data.

To clean the object files and the executable file:

```
$ make clean
```

APPENDIX C. HARDWARE SPECIFICATIONS

C.1 CPUs

	Intel Xeon Silver 4110 CPU	**NVIDIA Carmel CPU**
Instruction Set Architecture	x86-64	ARMx8
# of Cores	8	8
# of Threads	16	8
Base Frequency	2.100 GHz	2.265 GHz
Max Frequency	3.000 GHz	-
Cache	L1: 256 KB (data) L2: 8 MB L3: 11 MB	L1: 64 KB (data) L2: 2 MiB L3: 4 MiB
Memory	64 GB DDR4	LPDDR4x 32 GB (UPM)

C.2 GPUs

	Titan RTX	**Jetson AGX Xavier**
Architecture	Turing	Volta
CUDA Driver Version / Runtime Version	10.2 / 10.2	10.2 / 10.2
CUDA Capability	7.5	7.2
Global Memory	24576 MB	31927 MB
CUDA Cores	72 Multiprocessors, 4608 CUDA Cores	8 Multiprocessors, 512 CUDA Cores
GPU Max Clock rate	1770 MHz	1377 MHz
Memory Clock rate	7001 MHz	1377 MHz
Memory Bus Width	384-bit	256-bit
L2 Cache	6291456 bytes	524288 bytes
Max Texture Dim (x,y,z)	1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)	1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
Maximum Layered 1D Texture Size, (num) layers	1D=(32768), 2048 layers	1D=(32768), 2048 layers

Maximum Layered 2D Texture Size, (num) layers	2D=(32768, 32768), 2048 layers	2D=(32768, 32768), 2048 layers
Total amount of constant memory:	65536 bytes	65536 bytes
Total amount of shared memory per block:	49152 bytes	49152 bytes
Total shared memory per multiprocessor:		
Total number of registers available per block:	65536	65536
Warp size:	32	32
Maximum number of threads per multiprocessor:	1024	2048
Maximum number of threads per block:	1024	1024
Max dimension size of a thread block (x,y,z):	(1024, 1024, 64)	(1024, 1024, 64)
Max dimension size of a grid size (x,y,z):	(2147483647, 65535, 65535)	(2147483647, 65535, 65535)
Maximum memory pitch:	2147483647 bytes	2147483647 bytes
Texture alignment:	512 bytes	512 bytes
Concurrent copy and kernel execution:		Yes with 1 copy engine(s)
Run time limit on kernels:	Yes	No
Integrated GPU sharing Host Memory:	No	Yes
Support host page-locked memory mapping:	Yes	Yes
Alignment requirement for Surfaces:	Yes	Yes
Device has ECC support:	Disabled	Disabled
CUDA Device Driver Mode (TCC or WDDM):	WDDM (Windows Display Driver Model)	
Device supports Unified Addressing (UVA):	Yes	Yes
Device supports Compute Preemption:	Yes	Yes
Supports Cooperative Kernel Launch:	No	Yes
Supports MultiDevice Co-op Kernel Launch:	No	Yes
Device PCI Domain ID / Bus ID / location ID:	0 / 23 / 0	0 / 0 / 0

APPENDIX D. DERIVATION OF v

The gradient of L_ρ with respect to v is:

$$\begin{aligned}\nabla_v L_\rho(v, z_1, z_2, \mu_1, \mu_2) = \frac{1}{N}\sum_m A_{-m}^T A_{-m} v - A_{-m}^T \Theta_{-m} I_m \\ + \rho D^T D v + \rho D^T(\mu_1 - z_1) \\ + \rho v - \rho(z_2 + \mu_2)\end{aligned} \tag{D.1}$$

To find the minimum, we set Eq. (C.1) equal to zero and solve for v:

$$\begin{aligned}0 = \frac{1}{N}\sum_m A_{-m}^T A_{-m} v - A_{-m}^T \Theta_{-m} I_m + \rho D^T D v + \rho D^T(\mu_1 - z_1) + \rho v \\ - \rho(z_2 + \mu_2)\end{aligned} \tag{D.2}$$

$$\begin{aligned}\frac{1}{N}\sum_m A_{-m}^T A_{-m} v + \rho D^T D v + \rho D^T \\ = \frac{1}{N}\sum_m A_{-m}^T \Theta_{-m} I_m + \rho D^T(z_1 - \mu_1) + \rho(z_2 - \mu_2)\end{aligned} \tag{D.3}$$

$$\begin{aligned}\left(\frac{1}{N}\sum_m A_{-m}^T A_{-m} + \rho D^T D + \rho I\right) v \\ = \frac{1}{N}\sum_m A_{-m}^T \Theta_{-m} I_m + \rho D^T(z_1 - \mu_1) + \rho(z_2 - \mu_2)\end{aligned} \tag{D.4}$$

$\frac{1}{N}\sum_m A_{-m}^T A_{-m}$, $\rho D^T D$, and ρI are all block circulant matrices, so they are diagonalizable by the discrete Fourier transform. Therefore, this minimization can be solved efficiently using the fast Fourier transform:

$$\hat{A}^T \hat{I} \leftarrow \frac{1}{N}\sum_m A_{-m}^T \Theta_{-m} I_m \tag{D.5}$$

$$v \leftarrow \mathcal{F}^{-1}\left\{\frac{\mathcal{F}\{\hat{A}^T \hat{I} + \rho D^T(z_1 - \mu_1) + \rho(z_2 - \mu_2)\}}{\frac{1}{N}\sum_m |\mathcal{F}\{A_{-m}\}|^2 + \rho(|\mathcal{F}\{D\}|^2 + 1)}\right\} \tag{D.6}$$

REFERENCES

[1] C. Allier, L. Hervé, O. Mandula, P. Blandin, Y. Usson, J. Savatier, S. Monneret, and S. Morales, "Quantitative phase imaging of adherent mammalian cells: a comparative study," *Biomed. Opt. Express,* vol. 10, pp. 2768-2783, Jun. 1, 2019.

[2] M. Baczewska, K. Eder, S. Ketelhut, B. Kemper, and M. Kujawinska, "Refractive Index Changes of Cells and Cellular Compartments Upon Paraformaldehyde Fixation Acquired by Tomographic Phase Microscopy," *Cytom. Part A,* p. 24229 (11 pp.), Sep. 22, 2020.

[3] A. Butola, D. Popova, A. Ahmad, V. Dubey, G. Acharya, P. Banet, P. Senthilkumaran, B. S. Ahluwalia, and D. S. Mehta, "Classification of human spermatozoa using quantitative phase imaging and machine learning," in *Digital Holography and 3D Imaging*, 2019, p. Th4A.3.

[4] T. Cacace, V. Bianco, and P. Ferraro, "Quantitative phase imaging trends in biomedical applications," *Opt. Laser Eng.,* vol. 135, p. 106188 (9 pp.), Dec. 2020.

[5] V. L. Calin, M. Mihailescu, E. I. Scarlat, A. V. Baluta, D. Calin, E. Kovacs, T. Savopol, and M. G. Moisescu, "Evaluation of the metastatic potential of malignant cells by image processing of digital holographic microscopy data," *Febs. Open Bio.,* vol. 7, pp. 1527-1538, Oct. 2017.

[6] G. Caprio, M. A. Ferrara, L. Miccio, F. Merola, P. Memmolo, P. Ferraro, and G. Coppola, "Holographic imaging of unlabelled sperm cells for semen analysis: a review," *J. Biophotonics,* vol. 8, pp. 779-789, Dec. 9, 2015.

[7] S. Cohen-Maslaton, I. Barnea, A. Taieb, and N. T. Shaked, "Cell and nucleus refractive-index mapping by interferometric phase microscopy and rapid confocal fluorescence microscopy," *J. Biophotonics,* p. e202000117 (11 pp.), May. 21, 2020.

[8] D. Gillies, W. Gamal, M. Canel, Y. Reinwald, Y. Yang, A. J. E. Haj, A. Serrels, and P. O. Bagnaninchi, "Real-time and non-invasive quantitative phase imaging of pancreatic ductal adenocarcinoma cell mechanical properties," *Proc. SPIE,* vol. 10880, p. 108800H (9 pp.), Feb. 21, 2019.

[9] T. Go, H. Byeon, and S. J. Lee, "Label-free sensor for automatic identification of erythrocytes using digital in-line holographic microscopy and machine learning," *Biosens Bioelectron,* vol. 103, pp. 12-18, Apr. 30, 2018.

[10] P. Guo, J. Huang, and M. A. Moses, "Characterization of dormant and active human cancer cells by quantitative phase imaging," *Cytom. Part A,* vol. 91A, pp. 424-432, Mar. 17, 2017.

[11] T. Henser-Brownhill, R. J. Ju, N. K. Haass, S. J. Stehbens, C. Ballestrem, and T. F. Cootes, "Estimation of cell cycle states of human melanoma cells with quantitative phase imaging and deep learning," in *IEEE 17th I. S. Biomed. Imaging (ISBI)*, 2020, pp. 1617-1621.

[12] J. Hur, K. Kim, S. Lee, H. Park, and Y. Park, "Melittin-induced alterations in morphology and deformability of human red blood cells using quantitative phase imaging techniques," *Sci. Rep.,* vol. 7, p. 9306 (10 pp.), Aug. 24, 2017.

[13] J. Jung, S. J. Hong, H. B. Kim, G. Kim, M. Lee, S. Shin, S. Lee, D. J. Kim, C. G. Lee, and Y. Park, "Label-free non-invasive quantitative measurement of lipid contents in individual microalgal cells using refractive index tomography," *Sci. Rep.,* vol. 8, p. 6534 (10 pp.), Apr. 15, 2018.

[14] B. Kemper, L. Pohl, M. Kaiser, E. Dopker, J. Schnekenburger, and S. Ketelhut, "Label-free detection of global morphology changes in confluent cell layers utilizing quantitative phase imaging with digital holographic microscopy," *Proc. SPIE,* vol. 11076, p. 30, Jul. 22, 2019.

[15] S.-A. Yang, J. Yoon, K. Kim, and Y. Park, "Measurements of morphological and biophysical alterations in individual neuron cells associated with early neurotoxic effects in Parkinson's disease," *Cytom. Part A,* vol. 91, pp. 510-518, Apr. 20, 2017.

[16] F. Charriere, N. Pavillon, T. Colomb, C. Depeursinge, T. J. Heger, E. A. D. Mitchell, P. Marquet, and B. Rappaz, "Living specimen tomography by digital holographic microscopy: morphometry of testate amoeba," *Opt. Express,* vol. 14, pp. 7005-7013, Aug. 7, 2006.

[17] Z. El-Schich, A. Molder, H. Tassidis, P. Harkonen, M. F. Miniotis, and A. G. Wingren, "Induction of morphological changes in death-induced cancer cells monitored by holographic microscopy," *J. Struct. Biol.,* vol. 189, pp. 207-212, Mar. 2015.

[18] H. Funamizu and Y. Aizu, "Three-dimensional quantitative phase imaging of blood coagulation structures by optical projection tomography in flow cytometry using digital holographic microscopy," *J. Biomed. Opt.,* vol. 24, p. 6, Mar. 2019.

[19] P. Girshovitz and N. T. Shaked, "Generalized cell morphological parameters based on interferometric phase microscopy and their application to cell life cycle characterization," *Biomed. Opt. Express,* vol. 3, pp. 1757-1773, Aug. 1, 2012.

[20] K. G. Phillips, S. L. Jacques, and O. J. T. McCarty, "Measurement of single cell refractive index, dry mass, volume, and density using a transillumination microscope," *Phys. Rev. Lett.,* vol. 109, p. 118105 (5 pp.), Sep. 13, 2012.

[21] S. Ceballos, M. Kandel, S. Sridharan, H. Majeed, F. Monroy, and G. Popescu, "Active intracellular transport in metastatic cells studied by spatial light interference microscopy," *J. Biomed. Opt.,* vol. 20, p. 111209, Nov. 2015.

[22] B. Kemper, A. Bauwens, A. Vollmer, S. Ketelhut, P. Langehanenberg, J. Muething, H. Karch, and G. von Bally, "Label-free quantitative cell division monitoring of endothelial cells by digital holographic microscopy," *J. Biomed. Opt.,* vol. 15, p. 036009 (6 pp.), May. 2010.

[23] P. A. Sandoz, C. Tremblay, F. G. van der Goot, and M. Frechin, "Image-based analysis of living mammalian cells using label-free 3D refractive index maps reveals new organelle dynamics and dry mass flux," *PLoS. Biol.,* vol. 17, p. 22, Dec. 19, 2019.

[24] Y. Park, C. A. Best, T. Kuriabova, M. L. Henle, M. S. Feld, A. J. Levine, and G. Popescu, "Measurement of the nonlinear elasticity of red blood cell membranes," *Phys. Rev. E.,* vol. 83, p. 051925 (pp. 17), May. 27, 2011.

[25] D. Huang, K. A. Leslie, D. Guest, O. Yeshcheulova, I. J. Roy, M. Piva, G. Moriceau, T. A. Zangle, R. S. Lo, M. A. Teitell, and J. Reed, "High-speed live-cell interferometry: A new method for quantifying tumor drug resistance and heterogeneity," *Anal. Chem.,* vol. 90, pp. 3299-3306, Mar. 6, 2018.

[26] D. D. Nolte, R. An, J. Turek, and K. Jeong, "Tissue dynamics spectroscopy for phenotypic profiling of drug effects in three-dimensional culture," *Biomed. Opt. Express,* vol. 3, pp. 2825-2841, Nov. 1, 2012.

[27] M. Mir, A. Bergamaschi, B. S. Katzenellenbogen, and G. Popescu, "Highly Sensitive Quantitative Imaging for Monitoring Single Cancer Cell Growth Kinetics and Drug Response," *PLOS One,* vol. 9, p. e89000, Feb. 18, 2014.

[28] T. Yao, R. Cao, W. Xiao, F. Pan, and X. Li, "An optical study of drug resistance detection in endometrial cancer cells by dynamic and quantitative phase imaging," *J. Biophotonics,* vol. 10, p. e201800443, Feb. 15, 2019.

[29] Z. Wang, K. Tangella, A. Balla, and G. Popescu, "Tissue refractive index as marker of disease," *J. Biomed. Opt.,* vol. 16, p. 7, Nov, 2011.

[30] M. Takabayashi, H. Majeed, A. Kajdacsy-Balla, and G. Popescu, "Disorder strength measured by quantitative phase imaging as intrinsic cancer marker in fixed tissue biopsies," *PLOS One,* vol. 13, p. e0194320 (10 pp.), Mar. 21, 2018.

[31] W. Huaqin, L. Zhifang, L. Hui, and W. Shulian, "Quantitative phase imaging of Breast cancer cell based on SLIM," in *J. Phys., Conf. Ser.*, UK, 2016, p. 012003 (5 pp.).

[32] V. K. Lam, T. C. Nguyen, B. M. Chung, G. Nehmetallah, and C. B. Raub, "Quantitative assessment of cancer cell morphology and motility using telecentric digital holographic microscopy and machine learning," *Cytom. Part A,* vol. 93A, pp. 334-345, Mar. 2018.

[33] D. Claus, A. M. Maiden, F. Zhang, F. G. R. Sweeney, M. J. Humphry, H. Schluesener, and J. M. Rodenburg, "Quantitative phase contrast optimised cancerous cell differentiation via ptychography," *Opt. Express,* vol. 20, pp. 9911-9918, Apr. 23, 2012.

[34] H. Majeed, M. E. Kandel, K. Han, Z. Luo, V. Macias, K. Tangella, A. Balla, and G. Popescu, "Breast cancer diagnosis using spatial light interference microscopy," *J. Biomed. Opt.,* vol. 20, pp. 111210-1--111210-6, Nov. 2015.

[35] T. H. Nguyen, S. Sridharan, V. Macias, A. K. Balla, M. N. Do, and G. Popescu, "Prostate cancer diagnosis using quantitative phase imaging and machine learning algorithms," *Proc. SPIE,* vol. 9336, pp. 933619-1--933619-10, 2015.

[36] D. Roitshtain, L. Wolbromsky, E. Bal, H. Greenspan, L. L. Satterwhite, and N. T. Shaked, "Quantitative phase microscopy spatial signatures of cancer cells," *Cytom. Part A,* vol. 91, pp. 482-493, Apr. 20, 2017.

[37] W. J. Choi, "Quantitative Phase-Contrast Imaging for Distinction between Different States of Human Breast Cancer Cells," *J. Korean Phys. Soc.,* vol. 74, pp. 574-578, Mar. 1, 2019.

[38] Z. El-Schich, A. Leida Mölder, and A. Gjörloff Wingren, "Quantitative phase imaging for label-free analysis of cancer cells—focus on digital holographic microscopy," *Applied Sciences,* vol. 8, p. 1027 (16 pp.), Jun. 23, 2018.

[39] Y. Park, D. Ryu, Y. S. Kim, K. Hong, and H.-S. Min, "Method and apparatus for rapid diagnosis of hematologic malignancy using 3D quantitative phase imaging and deep learning," United States Patent Application Publication no. 2020/0394794 A1, 2020.

[40] A. Anand, V. K. Chhaniwal, N. R. Patel, and B. Javidi, "Automatic identification of malaria-infected RBC with digital holographic microscopy using correlation algorithms," *IEEE Photonics J.,* vol. 4, pp. 1456-1464, Oct. 2012.

[41] P. Marquet, K. Rothenfusser, B. Rappaz, C. Depeursinge, P. Jourdain, and P. Magistretti, "Quantitative phase-digital holographic microscopy: a new imaging modality to identify original cellular biomarkers of diseases," *SPIE BiOS,* vol. 9718, May. 2, 2016.

[42] P. Marquet, C. Depeursinge, and P. J. Magistretti, "Review of quantitative phase-digital holographic microscopy: promising novel imaging technique to resolve neuronal network activity and identify cellular biomarkers of psychiatric disorders," *Neurophotonics,* vol. 1, p. 020901, Sep. 22, 2014.

[43] B. Kemper, D. Carl, J. Schnekenburger, I. Bredebusch, M. Schaefer, W. Domschke, and G. von Bally, "Investigation of living pancreas tumor cells by digital holographic microscopy," *J. Biomed. Opt.,* vol. 11, p. 034005 (8 pp.), May. 2006.

[44] Z. Wang, K. Tangella, A. Balla, and G. Popescu, "Tissue refractive index as marker of disease," *J. Biomed. Opt.,* vol. 16, Nov. 1, 2011.

[45] K. G. Phillips, C. R. Velasco, J. Li, A. Kolatkar, M. Luttgen, K. Bethel, B. Duggan, P. Kuhn, and O. J. McCarty, "Optical quantification of cellular mass, volume, and density of circulating tumor cells identified in an ovarian cancer patient," *Front. Oncol.,* vol. 2, p. 72 (8 pp.), Jul. 18, 2012.

[46] C. Hu and G. Popescu, "Quantitative Phase Imaging (QPI) in Neuroscience," *IEEE J. Sel. Top. Quant.,* vol. 25, pp. 1-9, Jan./Feb. 2019.

[47] P. Marquet, D. Boss, P. Jourdain, P. Magistretti, N. Pavillon, C. Depeursinge, and Ieee, "Digital holographic microscopy applied to neurociences (Invited Paper)," in *11th Eur.-Am. Worksh. Info.*, 2012.

[48] P. Jourdain, N. Pavillon, C. Moratal, D. Boss, B. Rappaz, C. Depeursinge, P. Marquet, and P. J. Magistretti, "Determination of transmembrane water fluxes in neurons elicited by glutamate ionotropic receptors and by the cotransporters KCC2 and NKCC1: a digital holographic microscopy study," *J Neurosci,* vol. 31, pp. 11846-11854, Aug. 17, 2011.

[49] H. S. Park, M. Rinehart, K. A. Walzer, J.-T. A. Chi, and A. Wax, "Automated detection of P. falciparum using machine learning algorithms with quantitative phase images of unstained cells," *PLOS One,* vol. 11, p. 0163045 (19 pp.), Sep. 16, 2016.

[50] C. Trujillo and J. Garcia-Sucerquia, "Automatic detection and counting of phase objects in raw holograms of digital holographic microscopy via deep learning," *Opt. Laser Eng.,* vol. 120, pp. 13-20, Sep. 1, 2019.

[51] T. H. Nguyen, S. Sridharan, V. Macias, A. Kajdacsy-Balla, J. Melamed, M. N. Do, and G. Popescu, "Automatic Gleason grading of prostate cancer using quantitative phase imaging and machine learning," *J. Biomed. Opt.,* vol. 22, p. 036015 (12 pp.), Mar. 2017.

[52] I. Moon, K. Jaferzadeh, Y. Kim, and B. Javidi, "Noise-free quantitative phase imaging in Gabor holography with conditional generative adversarial network," *Opt. Express,* vol. 28, pp. 26284-26301, Aug. 31, 2020.

[53] G. Kim, Y. Jo, H. Cho, H. S. Min, and Y. Park, "Learning-based screening of hematologic disorders using quantitative phase imaging of individual red blood cells," *Biosens Bioelectron,* vol. 123, pp. 69-76, Jan. 2019.

[54] C. L. Chen, A. Mahjoubfar, L.-c. Tai, I. K. Blaby, A. Huang, K. R. Niazi, and B. Jalali, "Deep learning in label-free cell classification," *Scientific Reports* vol. 6, p. 21471, Mar. 15, 2016.

[55] S. H. Karandikar, C. Zhang, A. Meirappan, I. Barman, C. Finck, P. K. Srivastava, and R. Pandey, "Reagent-free and rapid assessment of T cell activation state using diffraction phase microscopy and deep learning," *Anal. Chem.,* vol. 91, pp. 3405-3411, Mar. 2019.

[56] F. Yi, I. Moon, and B. Javidi, "Automated red blood cells extraction from holographic images using fully convolutional neural networks," *Biomed. Opt. Express,* vol. 8, pp. 4466-4479, Oct. 1, 2017.

[57] T. O'Connor, S. Rawat, A. Markman, and B. Javidi, "Automatic cell identification and visualization using digital holographic microscopy with head mounted augmented reality devices," *Appl. Opt.,* vol. 57, pp. B197-B204, Mar. 1, 2018.

[58] S. Abad, M. Lopez-Amo, F. M. Araujo, L. A. Ferreira, and J. L. Santos, "Fiber Bragg grating-based self-referencing technique for wavelength-multiplexed intensity sensors," *Opt. Lett.,* vol. 27, pp. 222-4, 2002.

[59] G. P. Agrawal and S. Radic, "Phase-shifted fiber Bragg gratings and their application for wavelength demultiplexing," *IEEE Photonics Technology Letters,* vol. 6, pp. 995-997, 1994.

[60] P. Orr and P. Niewczas, "High-Speed, Solid State, Interferometric Interrogator and Multiplexer for Fiber Bragg Grating Sensors," *J. Lightwave Technol.,* vol. 29, pp. 3387-92, 2011.

[61] S.-L. Tsao, J. Wu, and B.-C. Yeh, "High-resolution neural temperature sensor using fiber Bragg gratings," *IEEE J. Quantum Electron.,* vol. 35, pp. 1590-1596, 1999.

[62] H. Z. Yang, X. G. Qiao, Y. P. Wang, M. M. Ali, M. H. Lai, K. S. Lim, and H. Ahmad, "In-fiber gratings for simultaneous monitoring temperature and strain in ultrahigh temperature," *IEEE Photonics Technol. Lett.,* vol. 27, pp. 58-61, 2015.

[63] A. D. Kersey, T. A. Berkoff, and W. W. Morey, "Fiber-grating based strain sensor with phase sensitive detection," in *First European Conference on Smart Structures and Materials, 12-14 May 1992*, Bristol, UK, 1992, pp. 61-7.

[64] M. Kanik, S. Orguc, G. Varnavides, J. Kim, T. Benavides, D. Gonzalez, T. Akintilo, C. C. Tasan, A. P. Chandrakasan, Y. Fink, and P. Anikeeva, "Strain-

programmable fiber-based artificial muscle," *Science,* vol. 365, pp. 145-150, Jul. 12, 2019.

[65] Y. Zhang, D. Feng, Z. Liu, Z. Guo, X. Dong, K. S. Chiang, and B. C. B. Chu, "High-sensitivity pressure sensor using a shielded polymer-coated fiber Bragg grating," *IEEE Photonics Technol. Lett.,* vol. 13, pp. 618-619, 2001.

[66] A. Fender, E. J. Rigg, R. R. J. Maier, W. N. MacPherson, J. S. Barton, A. J. Moore, J. D. C. Jones, D. Zhao, L. Zhang, I. Bennion, S. McCulloch, and B. J. S. Jones, "Dynamic two-axis curvature measurement using multicore fiber Bragg gratings interrogated by arrayed waveguide gratings," *Appl. Opt.,* vol. 45, pp. 9041-9048, Dec. 20, 2006.

[67] R. Ahmadi, M. Packirisamy, J. Dargahi, and R. Cecere, "Discretely loaded beam-type optical fiber tactile sensor for tissue manipulation and palpation in minimally invasive robotic surgery," *IEEE Sensors Journal,* vol. 12, pp. 22-32, 2012.

[68] P. Puangmali, L. Hongbin, K. Althoefer, and L. D. Seneviratne, "Optical fiber sensor for soft tissue investigation during minimally invasive surgery," in *IEEE Int. Conf. Robotics Automation*, 2008, pp. 2934-2939.

[69] G. Rajan, D. Callaghan, Y. Semenova, and G. Farrell, "Miniature temperature insensitive fiber optic sensors for minimally invasive surgical devices," *Proc. SPIE,* vol. 7753, p. 77536Z (4 pp.), 2011.

[70] J. Peirs, J. Clijnen, D. Reynaerts, H. Van Brussel, P. Herijgers, B. Corteville, and S. Boone, "A micro optical force sensor for force feedback during minimally invasive robotic surgery," *Sensor. Actuat. A-Phys.,* vol. A115, pp. 447-455, 2004.

[71] T. Li, C. Shi, and H. Ren, "A high-sensitivity tactile sensor array based on fiber Bragg grating sensing for tissue palpation in minimally invasive surgery," *IEEE/ASME Transactions on Mechatronics,* vol. 23, pp. 2306-2315, Oct. 2018.

[72] C. Ledermann, H. Alagi, H. Woern, R. Schirren, and S. Reiser, "Biomimetic tactile sensor based on Fiber Bragg Gratings for tumor detection — Prototype and results," in *2014 IEEE International Symposium on Medical Measurements and Applications (MeMeA)*, 2014, pp. 1-6.

[73] A. A. Abushagur, N. Arsad, M. I. Reaz, and A. A. Bakar, "Advances in bio-tactile sensors for minimally invasive surgery using the fibre Bragg grating force sensor technique: a survey," *Sensors (Basel),* vol. 14, pp. 6633-65, Apr. 9, 2014.

[74] S. Park, G. Loke, Y. Fink, and P. Anikeeva, "Flexible fiber-based optoelectronics for neural interfaces," *Chem. Soc. Rev.,* vol. 48, pp. 1826-1852, Mar 2019.

[75] D. L. Presti, C. Massaroni, C. S. J. Leitão, M. D. F. Domingues, M. Sypabekova, D. Barrera, I. Floris, L. Massari, C. M. Oddo, S. Sales, I. I. Iordachita, D. Tosi, and E. Schena, "Fiber Bragg gratings for medical applications and future challenges: a review," *IEEE Access,* vol. 8, pp. 156863-156888, 2020.

[76] G. M. Noah, Y. Bao, and T. K. Gaylord, "Cross-sectional refractive-index variations in fiber Bragg gratings measured by quantitative phase imaging," *Opt. Lett.,* vol. 45, pp. 53-56, Jan. 2020.

[77] M. H. Jenkins and T. K. Gaylord, "Three-dimensional quantitative phase imaging via tomographic deconvolution phase microscopy," *Appl. Opt.,* vol. 54, pp. 9213-9227, Oct. 27, 2015.

[78] F. Zernike, "Phase contrast, a new method for the microscopic observation of transparent objects Part I," *Physica Scripta,* vol. 9, pp. 686-698, Jul. 1, 1942.

[79] P. Marquet, B. Rappaz, P. J. Magistretti, E. Cuche, Y. Emery, T. Colomb, and C. Depeursinge, "Digital holographic microscopy: a noninvasive contrast imaging technique allowing quantitative visualization of living cells with subwavelength axial accuracy," *Opt. Lett.,* vol. 30, pp. 468-470, Mar. 1, 2005.

[80] E. Cuche, P. Marquet, and C. Depeursinge, "Simultaneous amplitude-contrast and quantitative phase-contrast microscopy by numerical reconstruction of Fresnel off-axis holograms," *Appl. Opt.,* vol. 38, pp. 6994-7001, Dec. 1, 1999.

[81] Z. Wang, L. Millet, M. Mir, H. Ding, S. Unarunotai, J. Rogers, M. U. Gillette, and G. Popescu, "Spatial light interference microscopy (SLIM)," *Opt. Express,* vol. 19, pp. 1016-1026, Jan. 17, 2011.

[82] X. Ou, R. Horstmeyer, C. Yang, and G. Zheng, "Quantitative phase imaging via Fourier ptychographic microscopy," *Opt. Lett.,* vol. 38, pp. 4845-4848, Nov. 15, 2013.

[83] G. Zheng, R. Horstmeyer, and C. Yang, "Wide-field, high-resolution Fourier ptychographic microscopy," *Nat. Photon.,* vol. 7, pp. 739-745, Jul. 28, 2013.

[84] N. Streibl, "Phase imaging by the transport equation of intensity," *Opt. Commun.,* vol. 49, pp. 6-10, Feb. 1, 1984.

[85] M. R. Teague, "Deterministic phase retrieval: a Green's function solution," *J. Opt. Soc. Am. A,* vol. 73, pp. 1434-1441, Nov. 1, 1983.

[86] C. Zuo, J. Li, J. Sun, Y. Fan, J. Zhang, L. Lu, R. Zhang, B. Wang, L. Huang, and Q. Chen, "Transport of intensity equation: a tutorial," *Opt. Laser Eng.,* p. 106187 (89 pp.), Dec. 2020.

[87] D. Paganin and K. A. Nugent, "Noninterferometric phase imaging with partially coherent light," *Phys. Rev. Lett.,* vol. 80, pp. 2586-2589, Mar. 23, 1998.

[88] F. Charrière, A. Marian, F. Montfort, J. Kuehn, T. Colomb, E. Cuche, P. Marquet, and C. Depeursinge, "Cell refractive index tomography by digital holographic microscopy," *Opt. Lett.,* vol. 31, pp. 178-180, Jan. 15, 2006.

[89] W. Krauze, A. Kuś, and M. Kujawinska, "Limited-angle hybrid optical diffraction tomography system with total-variation-minimization-based reconstruction," *Opt. Eng.,* vol. 54, pp. 054104-054104, May 2015.

[90] Y. Sung, W. Choi, C. Fang-Yen, K. Badizadegan, R. R. Dasari, and M. S. Feld, "Optical diffraction tomography for high resolution live cell imaging," *Opt. Express,* vol. 17, pp. 266-277, Jan. 5, 2009.

[91] F. Macias-Garza, K. R. Diller, and A. C. Bovik, "Missing cone of frequencies and low-pass distortion in three-dimensional microscopic images," *Opt. Eng.,* vol. 27, pp. 461-465, Jun. 1988.

[92] F. Charrière, E. Cuche, P. Marquet, and C. Depeursinge, "Biological cell (pollen grain) refractive index tomography with digital holographic microscopy," *Proc. SPIE,* vol. 6090, pp. 609008-1--609008-8, Feb. 23, 2006.

[93] F. Charrière, J. Kühn, T. Colomb, E. Cuche, P. Marquet, and C. Depeursinge, "Sub-cellular quantitative optical diffraction tomography with digital holographic microscopy," *Proc. SPIE,* vol. 6441, pp. 64410K-1--64410K-6, Feb. 19, 2007.

[94] T. Kim, R. Zhou, M. Mir, S. D. Babacan, P. S. Carney, L. L. Goddard, and G. Popescu, "White-light diffraction tomography of unlabelled live cells," *Nat. Photon.,* vol. 8, pp. 256-263, Mar. 1, 2014.

[95] P. Bon, S. Aknoun, S. Monneret, and B. Wattellier, "Enhanced 3D spatial resolution in quantitative phase microscopy using spatially incoherent illumination," *Opt. Express,* vol. 22, pp. 8654-8671, Apr. 7, 2014.

[96] C. Zuo, Q. Chen, W. Qu, and A. Asundi, "High-speed transport-of-intensity phase microscopy with an electrically tunable lens," *Opt. Express,* vol. 21, pp. 24060-24075, Oct. 7, 2013.

[97] Y. Bao and T. K. Gaylord, "Iterative optimization in tomographic deconvolution phase microscopy," *J. Opt. Soc. Am. A,* vol. 35, pp. 652-660, Apr. 1, 2018.

[98] R. H. Dennard, F. H. Gaensslen, H. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *IEEE J. Solid-State Ckt.,* vol. 9, pp. 256-268, Oct. 1974.

[99] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics,* vol. 38, pp. 114-117, Apr. 19, 1965.

[100] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," *Proc. IEEE,* vol. 96, pp. 879-899, May 2008.

[101] A. Biguri, R. Lindroos, R. Bryll, H. Towsyfyan, H. Deyhle, I. E. k. Harrane, R. Boardman, M. Mavrogordato, M. Dosanjh, S. Hancock, and T. Blumensath, "Arbitrarily large tomography with iterative algorithms on multiple GPUs using the TIGRE toolbox," *J. Parallel Distr. Comput.,* vol. 146, pp. 52-63, Dec. 1, 2020.

[102] H. Scherl, B. Keck, M. Kowarschik, and J. Hornegger, "Fast GPU-based CT reconstruction using the Common Unified Device Architecture (CUDA)," in *IEEE Nuclear Science Symp. Conf. Record*, 2007, pp. 4464-4466.

[103] X. Jia, H. Yan, L. Cerviño, M. Folkerts, and S. B. Jiang, "A GPU tool for efficient, accurate, and realistic simulation of cone beam CT projections," *Med. Phys.,* vol. 39, pp. 7368-7378, Nov. 27, 2012.

[104] O. Inam, M. Qureshi, H. Akram, H. Omer, and Z. Laraib, "Accelerating parallel magnetic resonance image reconstruction on graphics processing units using CUDA," in *IEEE 2nd IEEE Int. Conf. Inf. Comput. Technol., ICICT*, 2019, pp. 109-113.

[105] M. Sabbagh, M. Uecker, A. J. Powell, M. Leeser, and M. H. Moghari, "Cardiac MRI compressed sensing image reconstruction with a graphics processing unit," in *10th Int. Sym. Med. Inform.*, 2016, pp. 1-5.

[106] Y. Zhuo, X. Wu, J. P. Haldar, W. Hwu, Z. Liang, and B. P. Sutton, "Accelerating iterative field-compensated MR image reconstruction on GPUs," in *IEEE Int. Symp. Biomed. Imaging: Nano Macro*, 2010, pp. 820-823.

[107] M. Doulgerakis-Kontoudis, A. Eggebrecht, S. Wojtkiewicz, J. Culver, and H. Dehghani, "Toward real-time diffuse optical tomography: accelerating light propagation modeling employing parallel computing on GPU and CPU," *J. Biomed. Opt.,* vol. 22, p. 125001, Dec. 1, 2017.

[108] Y. Dai, J. Tian, D. Dong, G. Yan, and H. Zheng, "Real-Time Visualized Freehand 3D Ultrasound Reconstruction Based on GPU," *IEEE Trans. Inf. Technol. Biomed.,* vol. 14, pp. 1338-1345, Nov. 2010.

[109] T. Reichl, J. Passenger, O. Acosta, and O. Salvado, "Ultrasound goes GPU: real-time simulation using CUDA," *Proc. SPIE,* vol. 7261, p. 726116 (10 pp.), Mar. 13, 2009.

[110] H. Pham, H. F. Ding, N. Sobh, M. Do, S. Patel, and G. Popescu, "Off-axis quantitative phase imaging processing using CUDA: toward real-time applications," *Biomed. Opt. Express,* vol. 2, pp. 1781-1793, Jul. 1, 2011.

[111] K. Kim, K. S. Kim, H. Park, J. C. Ye, and Y. Park, "Real-time visualization of 3-D dynamic microscopic objects using optical diffraction tomography," *Opt. Express,* vol. 21, pp. 32269-32278, Dec. 30, 2013.

[112] J. Frank, G. Wernicke, J. Matrisch, S. Wette, J. Beneke, and S. Altmeyer, "Quantitative determination of the optical properties of phase objects by using a real-time phase retrieval technique," *Proc. SPIE,* vol. 8082, p. 80820N (9 pp.), May 26, 2011.

[113] O. Fialka and M. Cadik, "FFT and Convolution Performance in Image Filtering on GPU," in *10th IEEE Int. Conf. Inf. Vi.*, 2006, pp. 609-614.

[114] M. A. Bruce and M. J. Butte, "Real-time GPU-based 3D Deconvolution," *Opt. Express,* vol. 21, pp. 4766-4773, Feb. 25, 2013.

[115] W. Wolf, B. Ozer, and T. Lv, "Smart cameras as embedded systems," *Computer,* vol. 35, pp. 48-53, Sep. 2002.

[116] T. Hussain, "ViPS: A novel visual processing system architecture for medical imaging," *Biomed. Signal Process.,* vol. 38, pp. 293-301, Sep. 1 2017.

[117] A. E. Desjardins, B. J. Vakoc, M. J. Suter, S. Yun, G. J. Tearney, and B. E. Bouma, "Real-time FPGA processing for high-dpeed optical frequency domain imaging," *IEEE Trans. Med. Imaging,* vol. 28, pp. 1468-1472, Mar. 24, 2009.

[118] J. Y. Xie, X. Y. Niu, A. K. S. Lau, K. K. Tsia, and H. K. H. So, "c," in *Int. Conf. Field Program. Technol.*, 2015, pp. 1-8.

[119] T. R. Savarimuthu, A. Kjaer-Nielsen, and A. S. Sorensen, "Real-time medical video processing, enabled by hardware accelerated correlations," *J. Real-Time Image Process.,* vol. 6, pp. 187-197, Sep. 2011.

[120] C. Moler. (2000). *MATLAB incorporates LAPACK*. Available: https://www.mathworks.com/company/newsletters/articles/matlab-incorporates-lapack.html

[121] MathWorks. (2021). *Parallel computing toolbox*. Available: https://www.mathworks.com/help/parallel-computing/

[122] OpenMP. (2018). *OpenMP application programming interface*. Available: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf

[123] H. Jin and N. Manjikian, "Embedded memory in system-on-chip design: Architecture and prototype implementation," in *CCECE Proc. Toward a Caring and Humane Technology*, New York, 2003, pp. 141-146.

[124] D. Franklin. (2018). *NVIDIA Jetson AGX Xavier delivers 32 TeraOps for new era of AI in robotics*. Available: https://developer.nvidia.com/blog/nvidia-jetson-agx-xavier-32-teraops-ai-robotics/

[125] M. Harris. (2013). *Unified memory in CUDA 6*. Available: https://developer.nvidia.com/blog/unified-memory-in-cuda-6/

[126] NVIDIA. (2014). *NVIDIA CUDA Toolkit V6.0*. Available: http://developer.download.nvidia.com/compute/cuda/6_0/rel/docs/CUDA_Toolkit_Release_Notes.pdf

[127] N. Sakharnykh. (2017). *Unified memory on Pascal and Volta*. Available: https://on-demand.gputechconf.com/gtc/2017/presentation/s7285-nikolay-sakharnykh-unified-memory-on-pascal-and-volta.pdf

[128] W. Q. Li, G. H. Jin, X. W. Cui, and S. See, "An evaluation of unified memory technology on NVIDIA GPUs," in *15th IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing*, 2015, pp. 1092-1098.

[129] R. Cavicchioli, N. Capodieci, and M. Bertogna, "Memory interference characterization between CPU cores and integrated GPUs in mixed-criticality platforms," in *22nd IEEE Int. Conf. on Emerging Technologies and Factory Automation (ETFA)*, 2017, pp. 1-10.

[130] Y. Ukidave, D. Kaeli, U. Gupta, and K. Keville, "Performance of the NVIDIA Jetson TK1 in HPC," in *IEEE Int. Conf. on Cluster Computing*, 2015, pp. 533-534.

[131] J. Choi, H. You, C. Kim, H. Y. Yeom, and Y. Kim, "Comparing unified, pinned, and host/device memory allocations for memory-intensive workloads on Tegra SoC," *Concurr. Comp.-Pract. E.,* vol. 33, p. e6018 (10 pp.), Feb. 2021.

[132] R. Cavicchioli, N. Capodieci, and M. Bertogna, "Memory interference characterization between CPU cores and integrated GPUs in mixed-criticality platforms," in *22nd IEEE Int. C. Emerg.*, 2017, pp. 1-10.

[133] NVIDIA. (2020). *CUDA for Tegra*. Available: https://docs.nvidia.com/cuda/archive/10.2/cuda-for-tegra-appnote/index.html

[134] J. Guan, S. Yan, and J. M. Jin, "An openMP-CUDA implementation of multilevel fast multipole algorithm for electromagnetic simulation on multi-GPU computing systems," *IEEE Transactions on Antennas and Propagation,* vol. 61, pp. 3607-3616, Jul. 2013.

[135] S. Rosenberger and G. Haase, "Pragma Based GPU Parallelizations for Cardiovascular Simulations," in *Int. Conf. on High Performance Computing & Simulation*, New York, 2018, pp. 1022-1027.

[136] F. J. Hernandez-Lopez, R. Legarda-Saenz, and C. Brito-Loeza, "Parallel algorithm for fringe pattern demodulation," *J. of Real-Time Image Process.,* p. (11 pp.), Jun. 8, 2021.

[137] NVIDIA. (2019). *CUDA C++ programming guide*. Available: https://docs.nvidia.com/cuda/archive/10.2/cuda-c-programming-guide/index.html

[138] M. Schabel, "3D Shepp-Logan phantom," ed. MATLAB Cent. File Exch., 2006.

[139] TheHDFGroup. (2006). *HDF5*. Available: https://portal.hdfgroup.org/display/HDF5/HDF5

[140] J. Lim, K. Lee, K. H. Jin, S. Shin, S. Lee, Y. Park, and J. C. Ye, "Comparative study of iterative reconstruction algorithms for missing cone problems in optical diffraction tomography," *Opt. Express,* vol. 23, pp. 16933-16948, Jun. 29, 2015.

[141] J. Huang, H. Jin, Q. Ye, and G. Meng, "Iterative phase retrieval by combining modulus constraints and angle relationships," *Inverse Probl.,* vol. 35, p. 014002 (23 pp.), 2019.

[142] A. Doblas, C. Buitrago-Duque, A. Robinson, and J. Garcia-Sucerquia, "Phase-shifting digital holographic microscopy with an iterative blind reconstruction algorithm," *Appl. Opt.,* vol. 58, pp. G311-G317, Dec. 1, 2019.

[143] K. Liu, H. Cheng, C. Zhang, C. Shen, F. Zhang, and S. Wei, "Iterative feedback algorithm for phase retrieval based on transport of intensity equation," *Proc. SPIE,* vol. 9817, pp. 98171F-1--98171F-5, Dec. 9, 2015.

[144] S. Fan, S. Smith-Dryden, J. Zhao, S. Gausmann, A. Schülzgen, G. Li, and B. E. A. Saleh, "Optical fiber refractive index profiling by iterative optical diffraction tomography," *J. Lightwave Technol.,* vol. 36, pp. 5754-5763, Dec. 15, 2018.

[145] T. Latychevskaia, "Iterative phase retrieval for digital holography: tutorial," *J. Opt. Soc. Am. A,* vol. 36, pp. D31-D40, Dec, 01 2019.

[146] A. H. Delaney and Y. Bresler, "Globally convergent edge-preserving regularized reconstruction: an application to limited-angle tomography," *IEEE Trans. Image Process.,* vol. 7, pp. 204-221, Feb. 1998.

[147] P. Charbonnier, L. Blanc-Feraud, G. Aubert, and M. Barlaud, "Deterministic edge-preserving regularization in computed imaging," *IEEE Trans. Image Process.,* vol. 6, pp. 298-311, Feb. 1997.

[148] M. Guo, L. Chen, X. Shen, H. Iwai, Y. Chen, and H. Liu, "System model enabling fast tomographic phase microscopy with total variation regularisation," *Phys. Med. Biol.,* vol. 60, pp. 9059-9077, 2015.

[149] M. Persson, D. Bone, and H. Elmqvist, "Total variation norm for three-dimensional iterative reconstruction in limited view angle tomography," *Phys. Med. Biol.,* vol. 46, pp. 853-866, 2001.

[150] E. Y. Sidky and X. Pan, "Image reconstruction in circular cone-beam computed tomography by constrained, total-variation minimization," *Phys. Med. Biol.,* vol. 53, pp. 4777-4807, 2008.

[151] M. V. W. Zibetti, C. Lin, and G. T. Herman, "Total variation superiorized conjugate gradient method for image reconstruction," *Inverse Problems,* vol. 34, p. 034001 (28 pp.), Jan. 2018.

[152] R. Kasai, Y. Yamaguchi, T. Kojima, and T. Yoshinaga, "Hybrid algorithm of maximum-likelihood expectation-maximization and multiplicative algebraic reconstruction technique for iterative tomographic image reconstruction," *Proc. SPIE,* vol. 11049, p. 110491F (4 pp.), Mar. 22, 2019.

[153] Y. Saad, *Iterative Methods for Sparse Linear Systems: Second Edition*: Society for Industrial and Applied Mathematics (SIAM, 3600 Market Street, Floor 6, Philadelphia, PA 19104), 2003.

[154] Z. Luo, J. Ma, P. Su, and L. Cao, "Digital holographic phase imaging based on phase iteratively enhanced compressive sensing," *Opt. Lett.,* vol. 44, pp. 1395-1398, Mar. 15, 2019.

[155] K. C. Zhou, K. C. Zhou, R. Horstmeyer, and R. Horstmeyer, "Diffraction tomography with a deep image prior," *Opt. Express,* vol. 28, pp. 12872-12896, Apr. 27, 2020.

[156] U. S. Kamilov, I. N. Papadopoulos, M. H. Shoreh, A. Goy, C. Vonesch, M. Unser, and D. Psaltis, "Learning approach to optical tomography," *Optica,* vol. 2, pp. 517-522, Jun. 2015.

[157] A. Sinha, J. Lee, S. Li, and G. Barbastathis, "Lensless computational imaging through deep learning," *Optica,* vol. 4, pp. 1117-1125, Sep. 20, 2017.

[158] S. H. Chan, R. Khoshabeh, K. B. Gibson, P. E. Gill, and T. Q. Nguyen, "An augmented Lagrangian method for total variation video restoration," *IEEE Trans. Image Process.,* vol. 20, pp. 3097-3111, Nov. 2011.

[159] H. Ikoma, M. Broxton, T. Kudo, and G. Wetzstein, "A convex 3D deconvolution algorithm for low photon count fluorescence imaging," *Sci. Rep.,* vol. 8, pp. 1-12, Jul. 2018.

[160] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, "Distributed optimization and statistical learning via the alternating direction method of multipliers," *Foundations and Trends in Machine Learning,* vol. 3, pp. 1-122, 2011.

[161] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge: Cambridge University Press, 2014.

[162] J. M. Long, J. Y. Chun, and T. K. Gaylord, "ADMM approach for efficient iterative tomographic deconvolution reconstruction of 3D quantitative phase images," *Appl. Opt.,* vol. 60, 2021 (submitted).

[163] D. Ryu, D. Ryu, Y. Baek, H. Cho, G. Kim, Y. S. Kim, Y. Lee, Y. Kim, J. C. Ye, H.-S. Min, and Y. Park, "DeepRegularizer: rapid resolution enhancement of tomographic imaging using deep learning," *IEEE Trans. Med. Imaging,* vol. 40, pp. 1508-1518, May 2021.

[164] J. M. Soto, J. A. Rodrigo, and T. Alieva, "Label-free quantitative 3D tomographic imaging for partially coherent light microscopy," *Opt. Express,* vol. 25, pp. 15699-15712, Jul. 10, 2017.

[165] M. H. Jenkins, "New Quantitative Phase Imaging Modalities on Standard Microscope Platforms," Ph.D. Thesis, School of Electrical and Computer Engineering, Georgia Institute of Technology, 2015.

[166] Y. Bao, "Theory, Development, and Application of Quantitative Phase Imaging Modalities on Standard Microscope Platforms," Ph. D. Dissertation, Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA, 2019.

[167] G. M. Noah, "Quantitative Phase Imaging of Fiber Bragg Grating," M. S. thesis, Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA, 2019.

[168] NVIDIA. (2021). *cuBLAS*. Available: https://docs.nvidia.com/cuda/cublas/index.html

[169] NVIDIA. (2021). *cuFFT*. Available: https://docs.nvidia.com/cuda/cufft/index.html