GPU-Accelerated Algorithm to Compute Bessel-Fourier Moments

by

Tianpeng Xia

A thesis submitted to the Faculty of Graduate Studies in partial fulfillment of the requirements for the Master of Science degree.

Department of Applied Computer Science

University of Winnipeg

Winnipeg, Manitoba, Canada

March 2020

Copyright © 2020 Tianpeng Xia

Abstract

Bessel-Fourier moments have been applied in image pattern reconstruction since their introduction in 2010. In this research, a scalable GPU-based algorithm is proposed to accelerate the computation of Bessel-Fourier moments of high orders while preserving accuracy. To analyze our new algorithm, image reconstructions from Bessel-Fourier moments of orders up to 1000 were tested on two systems. The experimental results prove the correctness and scalability of the algorithm. In addition, by investigating the precision-related performance, both 64-bit and 32-bit precisions were shown to provide the same level of computational accuracy for Bessel-Fourier moments of orders up to 1000. Nevertheless, reconstructions with 64-bit precision are computationally more costly. Furthermore, we applied filtering in Bessel-Fourier moments and Fourier Frequency domains and found that Bessel-Fourier moments share some similarities with the frequencies in Fourier Frequency domain, though more image power is distributed in the Bessel-Fourier moments of lower orders.

Acknowledgements

This research would never have been possible without the support and guidance of professors and my classmates from the University of Winnipeg. I am a very lucky person to have the help and advice from those wonderful people.

First, I would like to thank Dr. Simon Liao for giving me the precious opportunity to complete my Master thesis under your supervision. Thank you for all the valuable ideas, suggestions and patience in guiding me through this research. Your wealth of knowledge in the field of digital image processing and pattern recognition in particular is inspiring. Thank you for sharing your programming experience with me when I was having a hard time debugging my programs.

I also want to thank Dr. Christopher Henry for your excellent course that introduced GPU programming to me. Being a listener in your classes is both a rewarding and enjoyable experience.

My sincere gratitude also goes to the faculty at the Department of Applied Computer Science for creating a friendly and comfortable study environment.

Last but not the least, I want to thank my family for their consistent encouragement and support, especially my parents and wife - Xiangli Wang. Since I started working on my thesis, Xiangli has been doing most of the housework so that I could focus on the research.

I believe the past two years of study at the University of Winnipeg to be a milestone in my life, and I am confident that the skills and experience I gained here will continue to serve as precious assets in my future career.

Contents

1 Introduction **1**

2 Image Moments **3**

2.1 Introduction 3

2.2 Orthogonal Image Moments 4

2.3 Computation Efficiency 6

3 Bessel-Fourier Moments **8**

3.1 Definition 8

3.2 Computational Accuracy 11

3.2.1 Approximation Error 11

3.2.2 Improvement of Accuracy 12

3.3 Summary 14

4 GPU-Based Implementation for Computing Bessel-Fourier Moments **15**

4.1 Matrix Operation for Moment Computing and Reconstruction . . 15

4.2 Symmetric Algorithm for Computing Bessel-Fourier Moments . . . 18

4.3 Reordering of Image Data 20

4.4 Memory Optimization 23

4.5 Kernels 26

4.5.1 Kernels for Computing Bessel-Fourier Moments 26

4.5.2 Reconstruction Kernel 28

4.6 Summary 28

5 Image Reconstructions from Bessel-Fourier Moments 29
5.1 Experiment 31
5.2 Summary 33

6 Filtering of Color Images in the Domain of Bessel-Fourier Moments 36
6.1 Ideal Filters 40
6.2 Gaussian Filters 48
6.3 Summary 48

7 Concluding Remarks 53
7.1 Summary 53
7.2 Future Work 54

A Source Code for GPU Kernels 56

List of Figures

3.1 Bessel polynomial $J_1(\lambda_n r)$ with $n = 1, 2, \ldots, 5$ 9
3.2 Circular domain in a Cartesion plane 10
3.3 The distributions of $J_1(\lambda_n r)$ within one of the four central pixels for an image sized at 1024×1024. 12
3.4 The distributions of $\exp(-jm\theta)$ within one of the four central pixels for an image sized at 1024×1024. 13
3.5 Applying a 2×2 scheme to a 4×4 image (a) yields (b). 14

4.1 Eight octants in a unit disk area 18
4.2 Reordering of image data 22
4.3 Two cases for warp W_i 23
4.4 CUDA Memory hierarchy 24
4.5 A Minimum Working Example of tiled algorithm 25

5.1 The testing images is sized at $1,024 \times 1,024$, with 256 gray levels 29
5.2 Reconstructed Figure 5.1 from Bessel-Fourier moments from order 80 to 200 31
5.3 Reconstructed Figure 5.1 from Bessel-Fourier moments from order 400 to 1000 32

6.1 Filtering in (a) Fourier Frequency domain and (b) Bessel-Fourier moments domain. 37
6.2 The Filtering Workflow 38

6.3 The two testing images are sized at 256×256 with 256 RGB densities . . . 38
6.4 Reconstruction results of the two testing images . . . 39
6.5 Results of applying ILF in the Bessel-Fourier moments domain 41
6.6 Results of applying ILF in the Fourier Frequency domain . . . 42
6.7 Results of applying IHF in the Bessel-Fourier moments domain 43
6.8 Results of applying IHF in the Fourier Frequency domain . . . 44
6.9 The percentage of image power for Figure 6.3(a) in the Bessel-Fourier moments domain . . . 46
6.10 The percentage of image power for Figure 6.3(a) in the Fourier Frequency domain . . . 46
6.11 The percentage of image power for Figure 6.3(b) in the Bessel-Fourier moments domain . . . 47
6.12 The percentage of image power for Figure 6.3(b) in the Fourier Frequency domain . . . 47
6.13 Results of applying GLF in the Bessel-Fourier moments domain 49
6.14 Results of applying GLF in the Fourier Frequency domain . . 50
6.15 Results of applying GHF in the Bessel-Fourier moments domain 51
6.16 Results of applying GHF in the Fourier Frequency domain . . 52

List of Tables

4.1 The Radial and Fourier parts of the eight points in Figure 4.1 19

5.1 PSNR values of applying different numerical schemes with 32-bit precision 33
5.2 Reconstruction times of applying different numerical schemes with 64-bit precision 34
5.3 Reconstruction times of applying different numerical schemes with 32-bit precision 35
5.4 The breakdown of computing time on System II 35

6.1 The PSNR values for the red, green, blue, and combined channels for the two images in Figure 6.3. 39

Chapter 1

Introduction

In the past four decades, information technologies have rapidly developed, and today, we are in the age of an information explosion. Information is coming from various forms such as voice, text, video, and image. As a communication medium, images contain richer information than texts or voices. At the same time, since videos are comprised of frames that are basically consecutive images, the analysis of videos is closely related to the analysis of images. Image analysis is thus becoming indispensable for modern society. A digital image, depending on whether it is gray-level or color, can be represented as a matrix or as three matrices of pixels with different density values and analyzed mathematically with the help of computer programs. Image analysis has been widely applied in such areas as security, traffic monitoring and face recognition.

Since the introduction of geometric moments by Hu in 1962 [9], remarkable achievements have been made by researchers around the world. Density values and other mathematical properties of pixels are usually used to compute image moments. Moreover, an approximated version of original images can be retrieved by conducting reconstruction from image moments.

The next breakthrough was the introduction of orthogonal moments by Teague in 1980 [24]. Zernike and Legendre moments were proposed by Teague as a solution to the inherent high information redundancy of geometric moments. Based on the foundational work of orthogonal moments laid by Teague, researchers have proposed other orthogonal moments to describe image information.

Numerous researchers have explored such topics as accuracy and the ap-

plication of continuous moments in both rectangular and circular domains. As defined in a circular domain, Bessel-Fourier moments have been shown to be more suitable for image analysis and pattern recognition [29]. Nevertheless, the computation of continuous moments is time consuming, which tends to limit their potential for further application. In this research, a parallel algorithm is proposed to compute Bessel-Fourier moments on CUDA-enabled GPUs. Filtering in the domain of Bessel-Fourier moments is also used to study some of their properties.

The remaining content is organized into six chapters. In Chapter 2, a brief literature review of image moments is provided. Chapter 3 introduces Bessel-Fourier moments and discusses the approximation error. Chapter 4 provides the implementation details of our algorithm. In Chapter 5, the experimental results are shown and analyzed. In Chapter 6, filtering is applied to the domain of Bessel-Fourier moments and the results are compared to those in the Fourier Frequency domain. Finally, a summary is given and potential areas of future work discussed in Chapter 7.

Chapter 2

Image Moments

2.1 Introduction

Image moments were first introduced to the field of pattern recognition by Hu in 1962 [9]. Based on the foundational work on algebraic invariants by Cayley, which is a branch of abstract algebra, Hu defined the 2-D $(p+q)th$ order moments of an image function $f(x,y)$ as [9]

$$m_{pq} = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} x^p y^q f(x,y)dx\,dy, \tag{2.1}$$

where $p, q = 0, 1, 2, \cdots$.

According to the uniqueness theorem, proven by Hu, the moment sequence m_{pq} is uniquely determined by the image function $f(x,y)$ and vice versa, which allows image reconstruction and recognition to be performed with image moments.

Geometric moments of lower orders hold intuitive properties of an image. The "mass" of an image is m_{00}; and the centroid of the image is defined by m_{10} and m_{01}. If we think of an image as a probability density function (pdf), with m_{00} normalized to 1, m_{10} and m_{01} represent the mean values. Second-order moments m_{20} and m_{02} reveal the "distribution of mass" of an image with regards to the coordinate axes, which is also called inertia in mechanics [7].

Hu also discovered that the centroid moments μ_{pq} are invariants under

translation, which are defined as

$$\mu_{pq} = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty}(x-\bar{x})^p(y-\bar{y})^q f(x,y)d(x-\bar{x})\,d(y-\bar{y}), \tag{2.2}$$

where $\bar{x} = m_{10}/m_{00}$ and $\bar{y} = m_{01}/m_{00}$.

Built upon the definition in Equation (2.1), seven absolute orthogonal moment invariants are introduced to accomplish pattern identification, independently of position, size, and orientation, and also independently of parallel projection.

In any case, the major issue of geometric moments is the high information redundancy. To address this problem, orthogonal moments have been proposed with various features such as higher accuracy and better performance. The general definition of image moments is given as

$$\Phi_{pq} = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\Psi_{pq}(x,y)f(x,y)dx\,dy, \tag{2.3}$$

where Ψ_{pq} denotes the kernel function and $p, q = 0, 1, 2\cdots$. In the case of geometric moments, $x^p y^q$ is the kernel function.

2.2 Orthogonal Image Moments

To overcome some of the drawbacks in geometric moments, such as the high information redundancy and computing complexity, Teague, in 1980, studied the properties of Legendre moments and Zernike moments [24]. Due to the recursive relation of Legendre moments, the computing complexity is much lower than that of geometric moments. It was also found that Zernike moments can be computed from geometric moments. Compared with geometric moments, higher-order moment invariants based on Zernike moments are simpler to retrieve. In addition, since each Zernike moment merely acquires a phase factor on rotation, the Zernike moments have relatively simple rotational properties.

In 1988, Teh and Chin carried out a comprehensive research on Legendre moments, Zernike moments and pseudo-Zernike moments [25]. They investigated three fundamental issues regarding their usefulness in image analysis, which were sensitivity to noise, aspects of information redundancy, and image reconstruction capacity. Their experimental results showed that moments of higher orders are generally more sensitive to noise and they all have lower information redundancy than the regular moments. Furthermore, a relatively small set of moments were sufficient to describe an image.

Since then, a number of orthogonal image moments in rectangular and circular domains were introduced to image processing. In 1994, Orthogonal Fourier-Mellin moments were proposed by Sheng and Shen [22]. In 2001, Mukundan proposed discrete Chebyshev moments [18]. In 2005, Wu and Shen presented Gaussian-Hermite moments [21]. In 2007, Jacobi-Fourier moments were proposed by Ping as a type of generic orthogonal moments defined in a circular domain, from which Legendre-Fourier moments, Chebyshev-Fourier moments and Zernike moments could be derived [20]. More recently in 2016, Xuan proposed circularly semi-orthogonal moments that do not involve factorial terms and are more robust to numeric errors [11].

Computation accuracy is closely related to the usefulness of continuous orthogonal moments such as Zernike moments and Orthogonal Fourier-Mellin moments. In 1998, Liao analyzed the discretization error for Zernike moments in detail, using cubature formulas to reduce the numerical error. In 2013, Wang proposed a numerical scheme that divides a pixel into $k \times k$ subregions with the same weights, and significant improvement was gained for images reconstructed from higher orders [28]. Due to its straightforwardness and performance, the $k \times k$ scheme was also used to improve the computing accuracy of other orthogonal image moments [3, 19, 27].

With the maturing of the theoretical premise for orthogonal moments, increasing interest has been shown by researchers in terms of different applications. In 2007, Xin proposed a method to select "good" Zernike/pseudo-

Zernike moments. Based on the invariance property for watermarking, the embedded information can be decoded at low error rates, and is robustness to various image manipulations [30]. In 2018, Bo studied rotation invariants for vector fields images computed from Gaussian-Hermite moments and Zernike moments, and found that the former demonstrated more stability [33].

In 2015, Bo proposed Gaussian-Hermite moments in 3D space and derived rotation invariants [32]. In 2018, Mostafa proposed 3D radial Hahn moments and conducted experiments in image reconstruction, geometric transformations, and pattern recognition [6].

As the popularity of artificial intelligence continued to rise in recent years, orthogonal image moments have been used in the field of machine learning. In 2017, Vijayalakshmin derived initial trainable convolution kernel coefficients from Zernike moments for a convolutional neural network (CNN) and analyzed the accuracy in gender classification and facial expression recognition [15]. Compared to the CNN architecture initialized with random kernels, the new method yielded satisfactory accuracy with less computation time. In 2018, Zernike moments were used as feature descriptors for facial images and trained under supervision using a Bayesian, support vector machine, linear discriminant analysis, and neural network classifiers by Vijayalakshmin. A greater than 90% accuracy was obtained for the neural network classifiers [16].

2.3 Computation Efficiency

Most techniques involving continuous moments are computationally expensive due to the exponential and factorial calculations. Therefore, besides accuracy, computation efficiency is another factor that has been impeding their application, especially in areas with limited processing time, such as surveillance and image search engines. There haven been some optimized algorithms for image moments defined in both rectangular and circular do-

mains.

In 2003, Chong and Raveendran proposed a fast algorithm called q-recursive method to improve the computing efficiency of Zernike moments by deriving moments of higher orders from those of lower orders [4]. Hwang made use of symmetric and anti-symmetric properties of Zernike basis functions and got Zernike moments by computing an octant of basis functions [10]. Chandan made q-recursive method even faster by deriving recurrence relation for the computation of trigonometric functions in 2011 [23]. In 2016, Rahul accelerated the computation of Jacobi-Fourier moments by using a recursive algorithm [26]. In 2016, Wang proposed a matrix algorithm for exponent-Fourier moments and improved the computation efficiency significantly even with increasing $k \times k$ schemes [27].

In 2019, Marcel developed a new algorithm by applying the recurrent formulas, symmetry properties, and parallelized matrix operations to compute the moments defined in a rectangular region. The experimental results showed that the new algorithm improved the efficiency of computing Legendre, Gegenbauer, and Jacobi moments extensively with excellent accuracy [19].

With the recent development of GPU (graphics processing unit) parallel programming, some research is emerging on GPU-accelerated parallel algorithms for computing orthogonal image moments. In 2014, Requena carried out an in-depth study on GPU computations for Zernike moments and achieved a 5x speedup for the Geforce 8800 GTX against a Pentium 4 CPU [17]. In 2018, Xuan proposed an optimized algorithm for computing Zernike moments by reordering the pixels before loading the data into GPU kernels, and the computing time was reduced by approximately half with the use of two GPUs [31].

Chapter 3

Bessel-Fourier Moments

3.1 Definition

Bessel function of the first kind [1, 2] is defined as

$$J_v(x) = \sum_{p=0}^{\infty} \frac{(-1)^p}{p!\Gamma(v+p+1)} (\frac{x}{2})^{v+2p}, \tag{3.1}$$

where $J_v(x)$ represents Bessel polynomial, $\Gamma(v+p+1)$ is the Gamma function based on v, which is the order of Bessel function, and $p = 1, 2, 3, \ldots$.

Utilizing Bessel function of the first kind as the moment weighting kernel, Bessel-Fourier moments are defined in a polar coordinate system

$$B_{nm} = \frac{1}{2\pi a_n} \int_0^{2\pi} \int_0^1 f(r,\theta) J_v(\lambda_n r) exp(-jm\theta) r dr d\theta, \tag{3.2}$$

where $n = 0, 1, 2, 3, \ldots$ and $m = 0, \pm 1, \pm 2, \pm 3, \ldots$ as the orders of Bessel-Fourier moments, $f(r, \theta)$ is the image function, and

$$a_n = \frac{[J_{v+1}(\lambda_n)]^2}{2} \tag{3.3}$$

is the normalization constant. Please note that the orders of Bessel-Fourier moments n and m are different from the order of Bessel function v in Equation 3.1, and that it is conventional to use the term - "order" in the study of image moments.

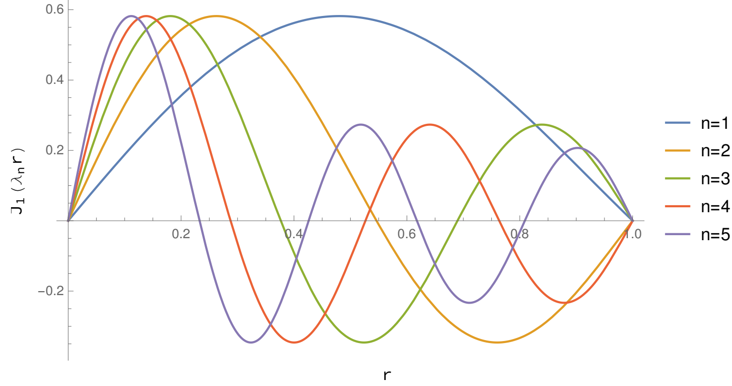


Figure 3.1: Bessel polynomial $J_1(\lambda_n r)$ with $n = 1, 2, \ldots, 5$

$J_v(\lambda_n r)$ is the Bessel polynomial in r of order n, λ_n is the n-th zero of $J_v(r)$ [29]. The 0-th zero of $J_v(r)$, λ_0, is defined as 0 when the order $v \neq 0$ [1, 2]. The plotting of $J_1(\lambda_n r)$ up to the fifth order are displayed in Figure 3.1, with $v = 1$. Since the value of v does not have an impact on the computation of image moments and it is a common practice to set v to 1, $J_1(\lambda_n r)$ is used in this research.

The Bessel polynomial set $J_1(\lambda_n r)$ satisfies the orthogonal property in range $0 \leq r \leq 1$

$$\int_0^1 r J_1(\lambda_n r) J_1(\lambda_m r) dr = a_n \delta_{nm}, \tag{3.4}$$

where δ_{nm} is the Kronecker symbol. Therefore, the kernel function of Bessel-Fourier moments, $J_1(\lambda_n r)\exp(-jm\theta)$, is also orthogonal on a unit circle disk

$$\int_0^1 \int_0^{2\pi} [J_1(\lambda_n r)\exp(-jp\theta)] J_1(\lambda_m r)\exp(-jq\theta) r dr d\theta = 2\pi a_n \delta_{nm} \delta_{pq}. \tag{3.5}$$

For the purpose of image processing and analysis, digital images are usually obtained by the means of Cartesian coordinate model [14]. Since Bessel-Fourier moments are defined in circular domain, only pixels located inside the unit disc area are involved in the computation, with the origin shifted to the center of the image (see Figure 3.2) and image dimensions scaled to be between 0 and 1.

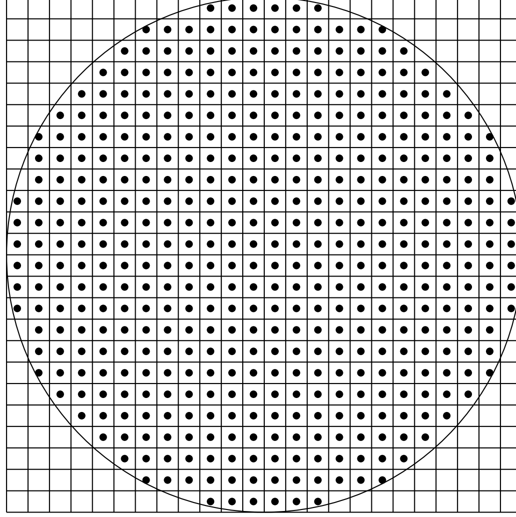


Figure 3.2: Implementing a circularly defined function in a Cartesian plane for a digital image

In a Cartesian coordinate system, Bessel-Fourier moments of a digital image function $f(x_i, y_j)$ sized at $M \times N$ can be computed as

$$\widehat{B}_{nm} = \frac{1}{2\pi a_n} \sum_{i=1}^{M} \sum_{j=1}^{N} f(x_i, y_j) J_1(\lambda_n r) \exp(-jm\theta) \triangle x \triangle y, \tag{3.6}$$

where $\triangle x$ and $\triangle y$ represent the sampling intervals in the x and y directions, respectively.

Due to the orthogonal property of Bessel-Fourier moments, an image function can be reconstructed by its infinite set of Bessel-Fourier moments [29]

$$f(r, \theta) = \sum_{n=0}^{\infty} \sum_{m=-\infty}^{\infty} B_{nm} J_1(\lambda_n r) \exp(-jm\theta), \tag{3.7}$$

where $f(r, \theta)$ represents the reconstructed image function, and B_{nm} is the set of Bessel-Fourier moments with order n from 0 to ∞ and order m from $-\infty$ to ∞.

Nevertheless, in practice, with a finite set of Bessel-Fourier moments, $0 \leq n \leq N$ and $-M \leq m \leq M$, we can only reconstruct an approximate

version of $f(r,\theta)$ by

$$\widehat{f}(r,\theta) = \sum_{n=0}^{N} \sum_{m=-M}^{M} B_{nm} J_1(\lambda_n r) \exp(-jm\theta). \tag{3.8}$$

If we replace B_{nm} with its version in a Cartesian coordinate system, $\widehat{B}_{nm}$ as expressed in Equation (3.6), the reconstructed image from a finite set of Bessel-Fourier moments, $\widehat{f}(r,\theta)$, can be expressed by

$$\widehat{f}(x_i, y_j) = \sum_{n=0}^{N} \sum_{m=-M}^{M} \widehat{B}_{nm} J_1(\lambda_n r) \exp(-jm\theta). \tag{3.9}$$

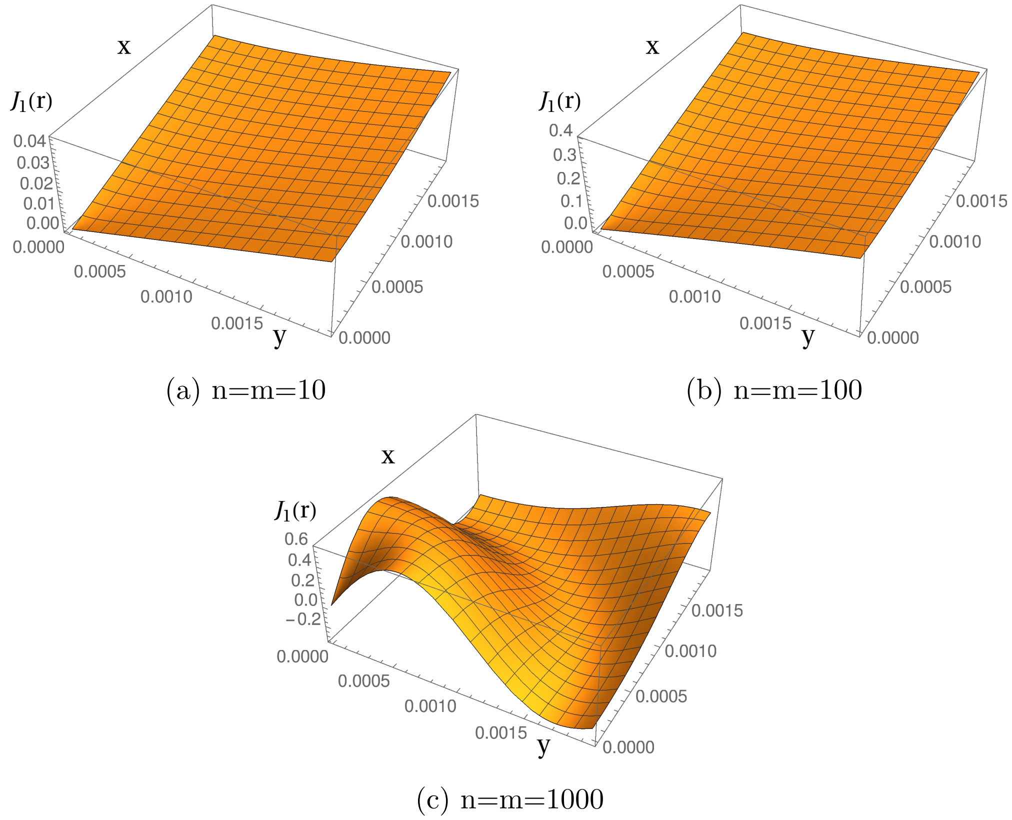
3.2 Computational Accuracy

3.2.1 Approximation Error

As one of the commonly used formulas to compute Bessel-Fourier moments of a digital image, Equation (3.6) is straightforward and relatively easy to implement. However, its computational accuracy is closely related to the distribution of the kernel function $J_1(\lambda_n r)\exp(-jm\theta)$.

If the distribution of the kernel function is smooth within each pixel, Equation (3.6) would provide a relatively justified approximation of Equation (3.2) [14]. Unfortunately, the distribution within a pixel varies significantly when the order increases.

For an image sized at 1024×1024, the distributions of $J_1(\lambda_n r)$ and $\exp(-jm\theta)$ within one of the four central pixels are displayed in Figure 3.3 and Figure 3.4 respectively. As order increases, it can be observed that the distribution of $J_1(\lambda_n r)$ varies more smoothly than that of $\exp(-jm\theta)$. When $n = m = 20$, the accuracy of approximated values for double integration in Equation (3.2) will suffer severe degradation.



(a) n=m=10

(b) n=m=100

(c) n=m=1000

Figure 3.3: The distributions of $J_1(\lambda_n r)$ within one of the four central pixels for an image sized at 1024×1024. Sub-figure (a) to (c) shows the distribution of $J_1(\lambda_n r)$ with $n = 10$, $n = 100$ and $n = 1000$, respectively

3.2.2 Improvement of Accuracy

To improve the computational accuracy of Bessel-Fourier moments, Equation (3.6) can be rewritten to

$$\widehat{B}_{nm} = \frac{1}{2\pi a_n} \sum_{i=1}^{M} \sum_{j=1}^{N} f(x_i, y_j) h_{nm}(x_i, y_j), \tag{3.10}$$

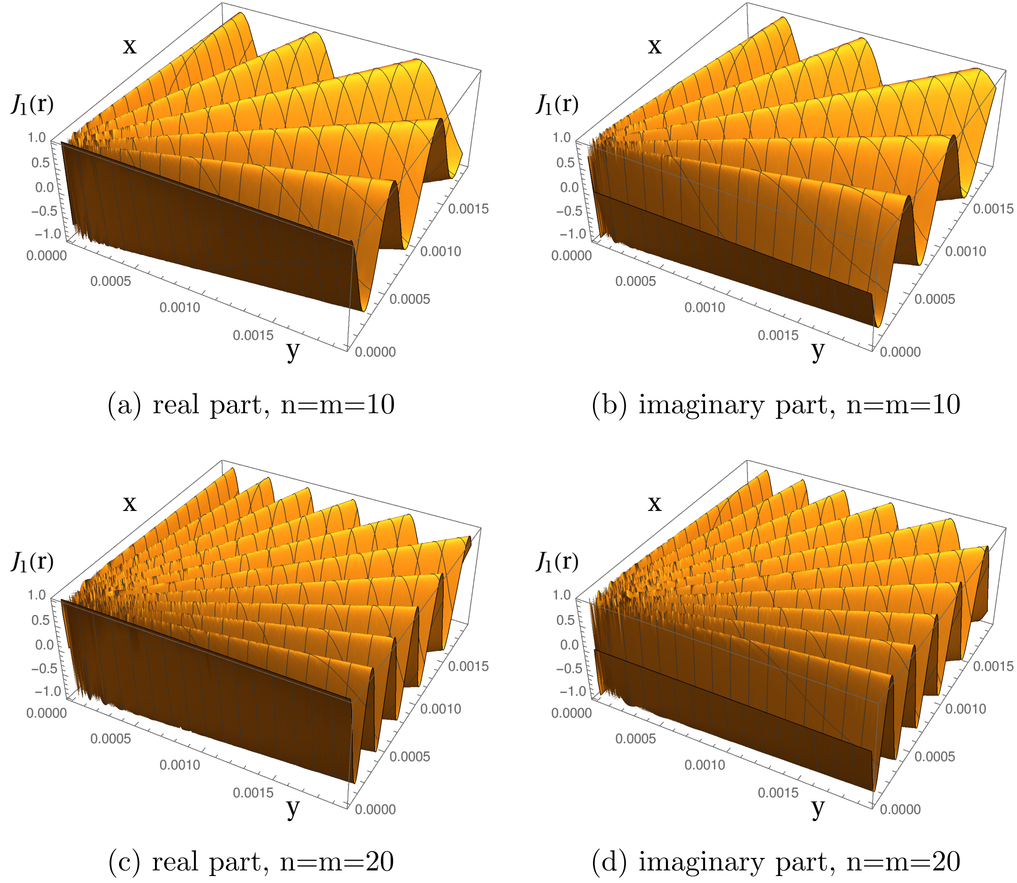


Figure 3.4: The distributions of $\exp(-jm\theta)$ within one of the four central pixels for an image sized at 1024×1024. Sub-figures (a) and (c) display the distributions of the real part of $\exp(-jm\theta)$ with $n = 10$ and $n = 20$, while (b) and (d) show those of the imaginary part

where

$$h_{mn}(x_i, y_j) = \int_{x_i - \frac{\triangle x}{2}}^{x_i + \frac{\triangle x}{2}} \int_{y_i - \frac{\triangle y}{2}}^{y_i + \frac{\triangle y}{2}} J_1(\lambda_n r) \exp(-jm\theta) dx\, dy. \tag{3.11}$$

It is obvious that the accuracy of $\widehat{B}_{nm}$ depends on the computation of Equation (3.11). In this research, we have adopted the method proposed in Reference [28], which divides each pixel into $k \times k$ sub-regions, the idea of

which is illustrated in Figure 3.5. By averaging the values of all sub-regions, the double integration in Equation (3.11) can be computed more accurately with a higher k due to increased sampling points within each pixel.

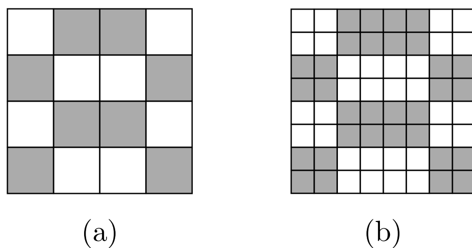


Figure 3.5: Applying a 2×2 scheme to a 4×4 image (a) yields (b).

3.3 Summary

In this chapter, we have analyzed the computational errors of Bessel-Fourier moments and concluded that the distributions of the Bessel kernel function and Fourier function are the major causes of the errors in computing. In order to improve the computational accuracy of Bessel-Fourier moments, the numerical $k \times k$ sub-region scheme is adopted in this research.

Chapter 4

GPU-Based Implementation for Computing Bessel-Fourier Moments

4.1 Matrix Operation for Moment Computing and Reconstruction

To implement the $k \times k$ numerical scheme in GPU computing, we have developed our algorithms based on the matrix operations.

Based on the matrix concept, we can rewrite the definition of Bessel-Fourier moments expressed in Equation (3.6) to

$$\boldsymbol{B_{nm}} = \frac{1}{2\pi \times \boldsymbol{A_n}} \circ [\boldsymbol{f(x,y)} \circ \boldsymbol{J_1(\lambda_n r)} \times \mathbf{exp}(\boldsymbol{-jm\theta})^T]dxdy, \tag{4.1}$$

where $\boldsymbol{A_n}$, $\boldsymbol{f(x,y)}$, $\boldsymbol{J_1(\lambda_n r)}$, and $\mathbf{exp}(\boldsymbol{-jm\theta})$ represent matrices. The symbol $\circ$ and $\times$ indicate the entrywise and matrix products between two matrices respectively.

$\boldsymbol{B_{nm}}$ is a matrix for the radial order n and Fourier order m of Bessel-Fourier moments with $(n+1)$ rows and $(2m+1)$ columns, which can be expressed by

$$\boldsymbol{B_{nm}} = \begin{bmatrix} B_{0,-m} & \cdots & B_{0,0} & \cdots & B_{0,m} \\ B_{1,-m} & \cdots & B_{1,0} & \cdots & B_{1,m} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ B_{n,-m} & \cdots & B_{n,0} & \cdots & B_{n,m} \end{bmatrix}. \tag{4.2}$$

In Equation (4.1), $\boldsymbol{A_n}$ is a matrix with $(n+1)$ rows and 1 column containing

normalization constants $J_2(\lambda_n)^2/2$ corresponding to different radial order n

$$\boldsymbol{A_n} = \begin{bmatrix} J_2(\lambda_0)^2/2 & J_2(\lambda_1)^2/2 & J_2(\lambda_2)^2/2 & \cdots & J_2(\lambda_n)^2/2 \end{bmatrix}^T, \tag{4.3}$$

where the symbol ^{T} denotes the transpose of a matrix. Assuming that the image function is sized at $M \times N$, the dimensions of $\boldsymbol{J_1(\lambda_n r)}$ are n rows by MN columns, where $\boldsymbol{r}$ represents the matrix dimensioned with M columns by N rows for the radius of sampling points

$$\boldsymbol{r} = \begin{bmatrix} r_{1,1} & r_{1,2} & \cdots & r_{1,N} \\ r_{2,1} & r_{2,2} & \cdots & r_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ r_{M,1} & r_{M,2} & \cdots & r_{M,N} \end{bmatrix}, \tag{4.4}$$

and $\boldsymbol{\lambda_n}$ is the matrix dimensioned with n rows and 1 column for the n-th zeros of Bessel function of the first kind

$$\boldsymbol{\lambda_n} = \begin{bmatrix} \lambda_0 & \lambda_1 & \lambda_2 & \cdots & \lambda_n \end{bmatrix}^T. \tag{4.5}$$

By combining the matrices $\boldsymbol{r}$ and $\boldsymbol{\lambda_n}$, $\boldsymbol{J_1(\lambda_n r)}$ can be expressed as

$$\boldsymbol{J_1(\lambda_n r)} = \begin{bmatrix} J_1(\lambda_0 r_{1,1}) & \cdots & J_1(\lambda_0 r_{1,N}) & \cdots & J_1(\lambda_0 r_{M,N}) \\ J_1(\lambda_1 r_{1,1}) & \cdots & J_1(\lambda_1 r_{1,N}) & \cdots & J_1(\lambda_1 r_{M,N}) \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ J_1(\lambda_n r_{1,1}) & \cdots & J_1(\lambda_n r_{1,N}) & \cdots & J_1(\lambda_n r_{M,N}) \end{bmatrix}. \tag{4.6}$$

On the other hand, $\boldsymbol{f(x, y)}$ is the data matrix of the image function

$$\boldsymbol{f(x, y)} = \begin{bmatrix} f(x_{1,1}, y_{1,1}) & \cdots & f(x_{1,N}, y_{1,N}) & \cdots & f(x_{M,N}, y_{M,N}) \end{bmatrix}, \tag{4.7}$$

therefore, the matrix $\boldsymbol{f(x, y)} \circ \boldsymbol{J_1(\lambda_n r)}$ can be expressed as

$$\begin{bmatrix} J_1(\lambda_0 r_{1,1}) f(x_{1,1}, y_{1,1}) & \cdots & J_1(\lambda_0 r_{1,N}) f(x_{1,N}, y_{1,N}) & \cdots & J_1(\lambda_0 r_{M,N}) f(x_{M,N}, y_{M,N}) \\ J_1(\lambda_1 r_{1,1}) f(x_{1,1}, y_{1,1}) & \cdots & J_1(\lambda_1 r_{1,N}) f(x_{1,N}, y_{1,N}) & \cdots & J_1(\lambda_1 r_{M,N}) f(x_{M,N}, y_{M,N}) \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ J_1(\lambda_n r_{1,1}) f(x_{1,1}, y_{1,1}) & \cdots & J_1(\lambda_n r_{1,N}) f(x_{1,N}, y_{1,N}) & \cdots & J_1(\lambda_n r_{M,N}) f(x_{M,N}, y_{M,N}) \end{bmatrix}. \tag{4.8}$$

Meanwhile, $\mathbf{exp(-\boldsymbol{jm\theta})}$ is the matrix of $e^{-jm\theta}$ that shows the e^{θ} of $(-jm)$-th order and the dimension of the matrix is $2 \times m + 1$ rows, which represents the orders from $-m$ to m, and $M \times N$ columns corresponding to every sampling point

$$\mathbf{exp}(-\boldsymbol{jm\theta}) = \begin{bmatrix} e^{jm\theta_{1,1}} & e^{jm\theta_{1,2}} & \cdots & e^{jm\theta_{M,N}} \\ \vdots & \vdots & \ddots & \vdots \\ e^{j\theta_{1,1}} & e^{j\theta_{1,2}} & \cdots & e^{j\theta_{M,N}} \\ e^{0} & e^{0} & \cdots & e^{0} \\ e^{-j\theta_{1,1}} & e^{-j\theta_{1,2}} & \cdots & e^{-j\theta_{M,N}} \\ \vdots & \vdots & \ddots & \vdots \\ e^{-jm\theta_{1,1}} & e^{-jm\theta_{1,2}} & \cdots & e^{-jm\theta_{M,N}} \end{bmatrix}. \quad (4.9)$$

Each element in matrix $\boldsymbol{B_{nm}}$ of Equation (4.2) represents the corresponding Bessel-Fourier moments of the image function $f(x, y)$ with the radial and Fourier orders, n and m, respectively.

To fully utilize the computing potential of GPUs, matrix operations are also used for image reconstruction, as shown in the equation below:

$$\boldsymbol{f(x,y)} = \boldsymbol{J_1^{*T}(\lambda_n r)} \times \boldsymbol{f(x,y)} \circ \mathbf{exp}(-\boldsymbol{jm\theta})^T \times \boldsymbol{ones}^T \, dx \, dy, \quad (4.10)$$

where $\boldsymbol{ones}$ is a 1×M matrix with all its elements equal to one. By reshaping the generated (MN)×1 matrix to a M×N one, the reconstructed image is retrieved.

4.2 Symmetric Algorithm for Computing Bessel-Fourier Moments

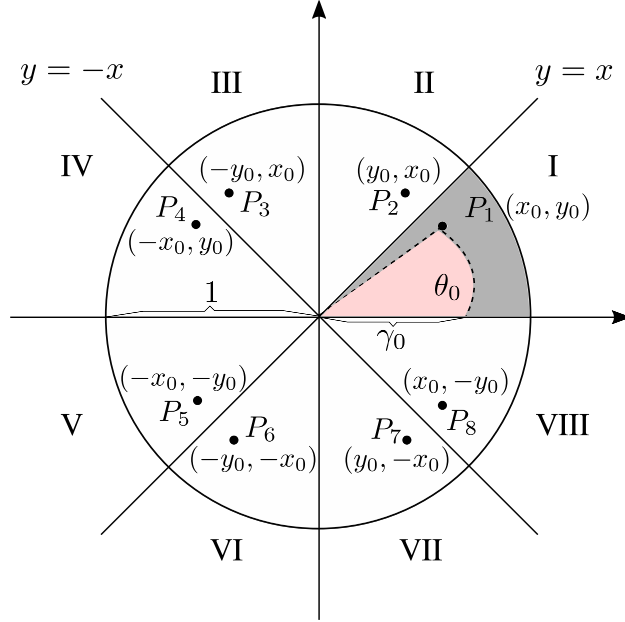


Figure 4.1: Eight octants in a unit disk area

In a Cartesian coordinate system, there are already four quadrants formed by the X and Y axes. The four quadrants can further by divided by two additional lines $y = x$ and $y = -x$. Due to the fact that Bessel polynomials are orthogonal within a unit disk, pixels outside the disk area do not participate in the calculation of Bessel-Fourier moments, the valid pixels eventually reside in eight octants of the circle shown in Figure 4.1.

For the point P_1 in octant I, there is one special point in each of the other seven octants, as shown in Figure 4.1. Let the polar coordinates of P_1 be (r_0, ρ_0), where $r_0 \leq 1$ and $\rho_0 < \pi/4$, the coordinates of the other seven points are derived in Table 4.1.

Octant #	Point	Radius	Angle	$J_1(\lambda_n\rho)$	$\exp(-jm\theta)$
I	P_1	ρ_0	θ_0	$J_1(\lambda_n\rho_0)$	$\exp(-jm\theta_0)$
II	P_2	ρ_0	$\frac{\pi}{2}-\theta_0$	$J_1(\lambda_n\rho_0)$	$\exp[-jm(\frac{\pi}{2}-\theta_0)]$
III	P_3	ρ_0	$\frac{\pi}{2}+\theta_0$	$J_1(\lambda_n\rho_0)$	$\exp[-jm(\frac{\pi}{2}+\theta_0)]$
IV	P_4	ρ_0	$\pi-\theta_0$	$J_1(\lambda_n\rho_0)$	$\exp[-jm(\pi-\theta_0)]$
V	P_5	ρ_0	$\pi+\theta_0$	$J_1(\lambda_n\rho_0)$	$\exp[-jm(\pi+\theta_0)]$
VI	P_6	ρ_0	$\frac{3\pi}{2}-\theta_0$	$J_1(\lambda_n\rho_0)$	$\exp[-jm(\frac{3\pi}{2}-\theta_0)]$
VII	P_7	ρ_0	$\frac{3\pi}{2}+\theta_0$	$J_1(\lambda_n\rho_0)$	$\exp[-jm(\frac{3\pi}{2}+\theta_0)]$
VIII	P_8	ρ_0	$2\pi-\theta_0$	$J_1(\lambda_n\rho_0)$	$\exp[-jm(2\pi-\theta_0)]$

Table 4.1: The evaluation of Radial $J_1(\lambda_n\rho)$ and Fourier $\exp(-jm\theta)$ parts for the eight points in Figure 4.1

According to Eula's formula, the Fourier part $\exp(-jm\theta)$ can also be expressed as:

$$\exp(-jm\theta) = \cos(m\theta) - j\,\sin(m\theta), \tag{4.11}$$

Therefore, Equation (3.6) can be rewritten to Equation (4.12) form using the similar notations in Reference [10]:

$$B_{nm} = BR_{nm} + BI_{nm},$$
$$\begin{cases} BR_{nm} = \frac{1}{2\pi a_n}\sum_{i=1}^{N}\sum_{j=1}^{M} f(x_i,y_i)J_1(\lambda_n r)\cos(m\theta), \\ BI_{nm} = -\frac{1}{2\pi a_n}\sum_{i=1}^{N}\sum_{j=1}^{M} f(x_i,y_i)J_1(\lambda_n r)j\,\sin(m\theta), \end{cases} \tag{4.12}$$

where BR_{nm} and BI_{nm} are the real and imaginary parts of B_{nm} respectively.

As explained in Reference [10], because the sinusoidal function is a periodic function with a regular waveform, $\cos(m\theta)$ and $i\sin(m\theta)$ have specific symmetric or anti-symmetric properties.

By the aforementioned symmetric and anti-symmetric properties, Equa-

tion (3.2) can be rewritten with two new notations g_m^i and g_m^r:

$$B_{nm} = \frac{1}{2\pi a_n} \iint\limits_{x^2+y^2 \leq 1} J_1(\lambda_n r)[g_m^r(x,y) - jg_m^i(x,y)]\, dxdy, \tag{4.13}$$

where $g_m^r(x,y)$ and $g_m^i(x,y)$ are defined in Equation (4.14). Let $h_i(i = 1, 2, ..., 8)$ be the value of the image function $f(r,\theta)$ at point P_i, g_m^i and g_m^r can be grouped into four cases [10]:

$$\begin{aligned}
&\boldsymbol{m = 4k}: \\
&g_m^r = [h_1 + h_2 + h_3 + h_4 + h_5 + h_6 + h_7 + h_8] \cos(m\theta_0), \\
&g_m^i = [h_1 - h_2 + h_3 - h_4 + h_5 - h_6 + h_7 - h_8] \sin(m\theta_0), \\
&\boldsymbol{m = 4k + 1}: \\
&g_m^r = [h_1 - h_4 - h_5 + h_8] \cos(m\theta_0) + [h_2 - h_3 - h_6 + h_7] \sin(m\theta_0), \\
&g_m^i = [h_1 + h_4 - h_5 - h_8] \sin(m\theta_0) + [h_2 + h_3 - h_6 - h_7] \cos(m\theta_0), \\
&\boldsymbol{m = 4k + 2}: \\
&g_m^r = [h_1 - h_2 - h_3 + h_4 + h_5 - h_6 - h_7 + h_8] \cos(m\theta_0), \\
&g_m^i = [h_1 + h_2 - h_3 - h_4 + h_5 + h_6 - h_7 - h_8] \cos(m\theta_0), \\
&\boldsymbol{m = 4k + 3}: \\
&g_m^r = [h_1 - h_4 - h_5 + h_8] \cos(m\theta_0) + [-h_2 + h_3 + h_6 - h_7] \sin(m\theta_0), \\
&g_m^i = [h_1 + h_4 - h_5 - h_8] \sin(m\theta_0) + [-h_2 - h_3 + h_6 + h_7] \cos(m\theta_0),
\end{aligned} \tag{4.14}$$

where $k \in \mathbb{Z}$ and $k \geqslant 0$.

According to the CUDA Programming Guide by Nvidia, "applications should strive to minimize data transfer between the host and the devices" [5]. In our case, only 1/8 of the data is needed to compute the $\mathbf{exp(-\boldsymbol{jm\theta})}$ matrix, which reduces the data transfer overhead by 7/8.

4.3 Reordering of Image Data

As illustrated in Figure 4.2 (a), the discrete version of Equation (4.12) can be applied to all of the pixels except for the shaded ones because they

lie in the joint areas of two octants.

For pixel $\acute{P_1}$ on the diagonals in Figure 4.2 (a), g_m^i and g_m^r can be grouped into 2 cases:

$$\begin{aligned}
&\boldsymbol{m = 2k}: \\
&g_m^r = [h_1 + h_2 + h_3 + h_4]\cos(m\tfrac{\pi}{4}), \\
&g_m^i = [h_1 - h_2 + h_3 - h_4]\sin(m\tfrac{\pi}{4}), \\
&\boldsymbol{m = 2k + 1}: \\
&g_m^r = [h_1 - h_2 - h_3 + h_4]\cos(m\tfrac{\pi}{4}), \\
&g_m^i = [h_1 + h_2 - h_3 - h_4]\sin(m\tfrac{\pi}{4}),
\end{aligned} \tag{4.15}$$

where $k \in \mathbb{Z}$ and $k \geqslant 0$. Therefore, those diagonal pixels need special treatment.

A straightforward approach is to make each thread check if a pixel belongs to the diagonal group before processing it. However, this method has a major flaw of creating branch divergence, which arises if some threads in a warp take the if path while the others take the $else$ path. It will cost the extra pass for the hardware to allow the threads in the same warp to make their own decisions and hinders the performance [13].

To address the divergence issue, Xuan et al. used eight arrays of the same dimensions while setting four of the slots of a diagonal pixel to 0 for parallel computation of Zernike moments [31]. To avoid adding the unnecessary zeroes used as dummy values to the GPU kernel, in this research, we have proposed a method for reordering image.

As shown in Figure 4.2(b), except for the diagonal pixels, all the pixels in octant I has seven corresponding pixels in the other octants which share the same colour. Pixels with a radius greater than 1 are not used for the computation of Bessel-Fourier moments and are coloured by white. In Figure 4.2(c), we have categorized the valid pixels into two groups - diagonal and non-diagonal.

The values of pixels in non-diagonal group are concatenated together from a_1 to a_8, and those in diagonal group from b_1 to b_4. Then the concatenated

values from diagonal and non-diagonal groups are further joined together to a flat array $\boldsymbol{I}$, with the non-diagonal group followed by the diagonal group.

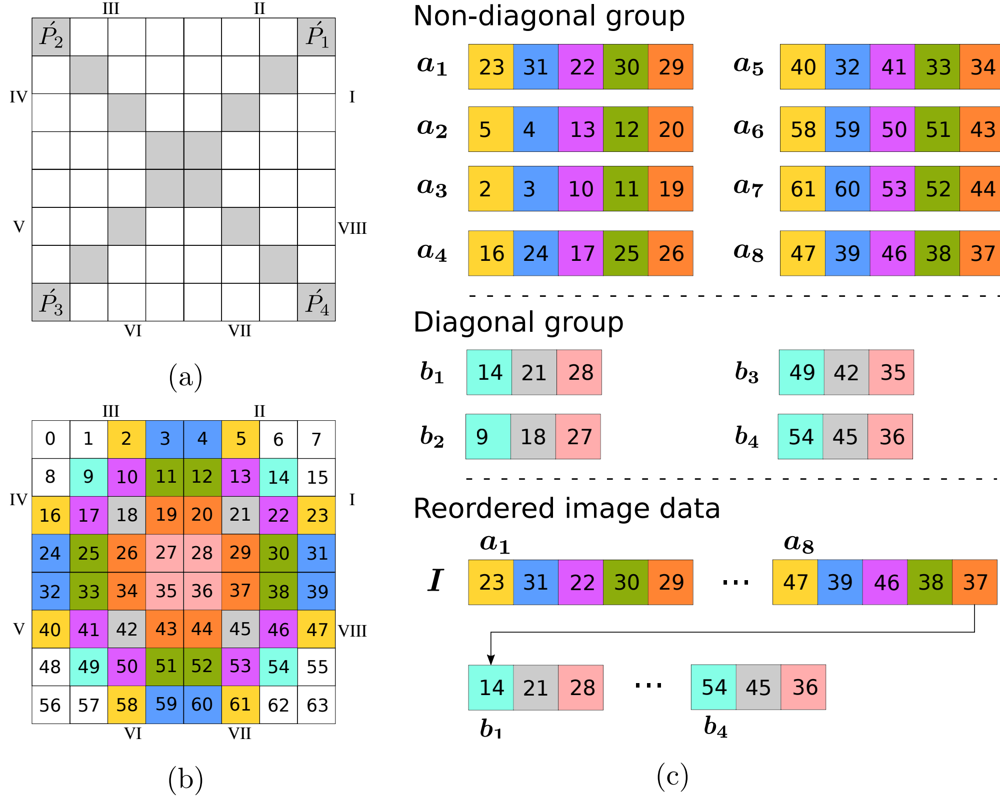


Figure 4.2: Reordering of image data for an 8×8 image. (a)Diagonal pixels, (b)pixels of the image with corresponding groups of a pixel filled the same color, (c)reordering of image data from diagonal and non-diagonal groups

According to Reference [5], threads are executed in groups of 32 parallel threads called warps and branch divergence happens only within a warp. Let W_i be the last warp (shaded), whose first thread is responsible for the calculation of pixels from the diagonal group in the reordered image data. There are two cases shown in Figure 4.3. In Case 1, all 32 threads in W_i will only do the computation for pixels from non-diagonal group and no divergence will occur. As for Case 2, W_i is spanning the pixels from both

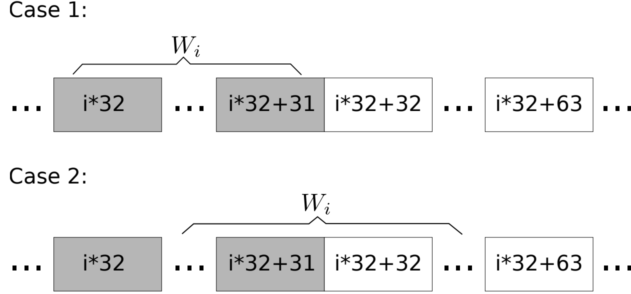


Figure 4.3: Two cases for warp W_i

groups and there will be one divergence. Therefore, using the reordering method proposed in this research, we can limit the occurrence of branch divergences to only once without extra zeroes. Compared with the method used in Reference [31], our reordering could reduce the consumption of device memory especially when there are a large number of small images to be processed.

4.4 Memory Optimization

To compute $\boldsymbol{Jv(\lambda_n r)}$ and $\mathbf{exp(-}\boldsymbol{jm\theta}\mathbf{)}$, two arrays of r and θ are copied to the GPU global memory using the same ordering as $\boldsymbol{I}$ in Figure 4.2(c), which will allow the computation of $\boldsymbol{B_{nm}}$ to use the same indexing method to find image values, corresponding radial polynomials and exponential values.

We store zeros crossings in constant memory as shown in Figure 4.4, which will benefit from constant cache residing in a multiprocessor and are accessible for all threads in the running kernel [5].

The efficiency of computing $\boldsymbol{B_{nm}}$ is closely related to two issues: data locality and access pattern. If we read data directly from the global memory, the implementation will suffer from relatively high latency. According to Reference [13], the favourite access pattern to data in global memory is achieved when all threads in a warp access consecutive global memory loca-

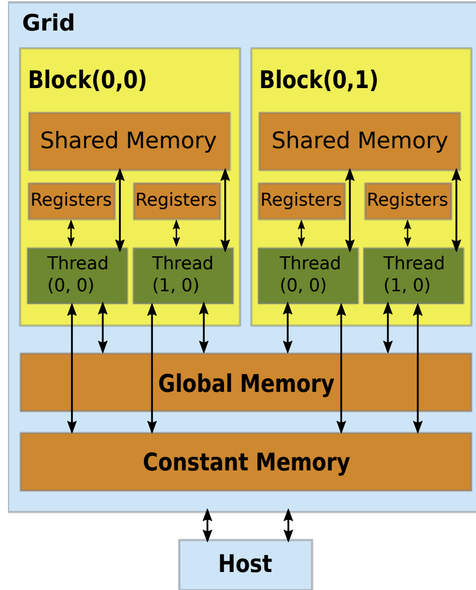


Figure 4.4: The hierarchical structure of CUDA Memory [13]

tions, which means that we have to do some extra work to make the threads read data in a coalesced pattern.

A tiled algorithm to utilize the shared memory is employed in this research to address both issues at the same time. The introduction of shared memory can address the former issue with its low latency because it is on-chip [5], which can be viewed in Figure 4.4. The latter issue is also addressed by loading partial data into their corresponding tiles so that a more intrinsic access pattern can be used without extra effort for a coalesced one.

However, due to the limited space of shared memory, we would have to do the loading multiple times to apply the tiled algorithm.

An Minimum Working Example (MWE) is shown in Figure 4.5 to display the computation process for the first thread block, where l and $\acute{l}$ denotes the number of pixels in one of the eight non-diagonal groups and those in one of the four diagonal groups. If we have a device that has only enough shared

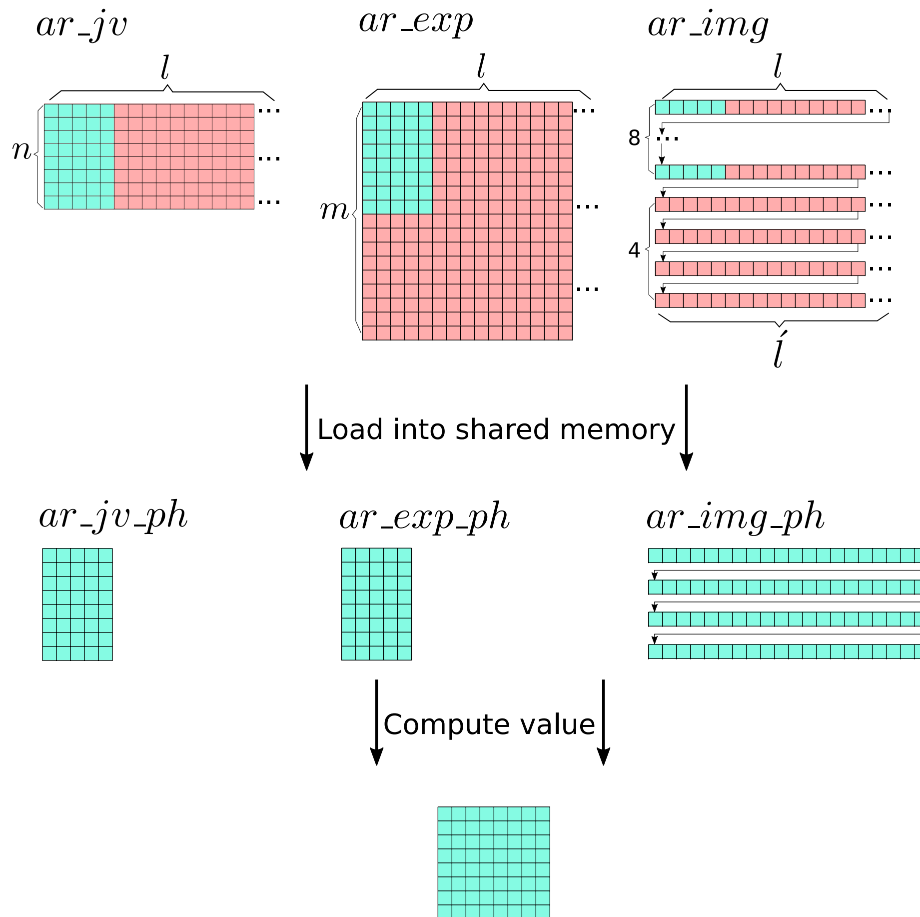


Figure 4.5: A Minimum Working Example of tiled algorithm

memory to store the data for 5 pixels in a block and each block has 64 threads, with variables $blockDim.y = blockDim.x = 8$, and $n = m = 8$. This will result in an 8×8 array as displayed in Figure 4.5. In Figure 4.5, *ar_jv*,*ar_exp* and *ar_img* denote the inputs of $\boldsymbol{Jv(\lambda_n r)}$, $\mathbf{exp}(\boldsymbol{-jm\theta})$ and image data respectively. Variables suffixed with "_ph" are used to store the values of the three input arrays in each iteration of the tiled algorithm.

4.5 Kernels

4.5.1 Kernels for Computing Bessel-Fourier Moments

Five kernels are used for computing $\boldsymbol{B_{nm}}$:

- RADIUS_KERNEL
- ARCTAN_KERNEL
- JV_KERNEL
- EXP_KERNEL
- BNM_KERNEL

The first two kernels compute the radius and angle values for pixels of radius greater than 1. Results computed from the first two kernels are then copied to JV_KERNEL and EXP_KERNEL to get $\boldsymbol{Jv(\lambda_n r)}$ and $\mathbf{exp}(\boldsymbol{-jm\theta})$. Since the first four kernels are straightforward, pseudocode of their algorithms is not given here.

Lastly, with $\boldsymbol{Jv(\lambda_n r)}$ and $\mathbf{exp}(\boldsymbol{-jm\theta})$ at hand, we could compute $\boldsymbol{B_{nm}}$ in BNM_KERNEL. The pseudocode for BNM_KERNEL is given in Algorithm 1, where **t**, *ar_bnm_out*, j, *ph_len* represent the current thread, the output array, imaginary unit and the number of pixels for each iteration respectively.

Algorithm 1 BNM_KERNEL

```
1: function BNM_KERNEL(ar_bnm_out, ar_jv, ar_exp, ar_img)
2:     __shared__ ar_jv_ph[ph_len]
3:     __shared__ ar_exp_ph[ph_len]
4:     __shared__ ar_img_ph[ph_len]
5:     total ← 0
6:     for each iteration do
7:         if t should load jv into shared memory then
8:             load values from ar_jv to ar_jv_ph
9:         end if
10:        if t should load exp into shared memory then
11:            load values from ar_exp to ar_exp_ph
12:        end if
13:        if t should load image value into shared memory then
14:            if t maps to a pixel from non-diagonal group then
15:                load values of 8 pixels into ar_img_k_ph[ph_len]
16:            else
17:                load values of 4 pixels into ar_img_k_ph[ph_len]
18:            end if
19:        end if
20:        __syncthreads()
21:        if t should compute a value then
22:            if t maps to a pixel from non-diagonal group then
23:                compute g_m^r and g_m^i according to Equation (4.14)
24:            else
25:                compute g_m^r and g_m^i according to Equation (4.15)
26:            end if
27:            total ← total + (g_m^r − j × g_m^i)
28:        end if
29:        __syncthreads()
30:    end for
31:    if t should compute a value then
32:        ar_bnm_out[pos] ← total
33:    end if
34: end function
```

The CUDA C++ source code for the five kernels are listed in Appendix A.

4.5.2 Reconstruction Kernel

After the reconstruction of Bessel-Fourier moments, they are then copied to RECONSTRUCTION_KERNEL for reconstruction. Since the logic for RECONSTRUCTION_KERNEL is very similar to that of BNM_KERNEL, its pseudocode is not listed here. The source code for RECONSTRUCTION_KERNEL is listed in Appendix A as well.

4.6 Summary

Based on matrix operations, a scalable GPU-based algorithm to compute Bessel-Fourier moments is proposed for the higher computational performance in this chapter. To minimize the data transfer between host and kernels, we employed the symmetric algorithm which reduced the data transfer overhead by nearly 7/8. At the same time, a pixel reordering method is also proposed to reduce the branch divergences without introducing extra dummy data.

Furthermore, constant and shared memories in the CUDA memory hierarchy are leveraged for lower accessing latency.

Chapter 5

Image Reconstructions from Bessel-Fourier Moments

To verify the new approaches to improve the accuracy and the efficiency in Bessel-Fourier moments computing, we have performed the image reconstructions from different maximum orders of Bessel-Fourier moments. Figure 5.1 shows the testing image used in this research, which is sized at $1,024 \times 1,024$ with 256 gray levels.

Figure 5.1: The testing images is sized at $1,024 \times 1,024$, with 256 gray levels

To evaluate the performances of image reconstructions, we have employed the Peak Signal to Noise Ratio (PSNR) to measure the qualities of the reconstructed images. The PSNR is defined as

$$PSNR = 10 \log_{10}(\frac{Max^2}{MSE}), \tag{5.1}$$

where Max is the maximum gray level of the evaluated image, and MSE is the Mean Square Error between the original image $f(x_i, y_j)$ and its reconstructed

version $\widehat{f}(x_i, y_j)$, both are sized at $M \times N$

$$MSE = \frac{1}{MN} \sum_{i=1}^{M} \sum_{j=1}^{N} [f(x_i, y_j) - \widehat{f}(x_i, y_j)]^2. \tag{5.2}$$

In general, a higher PSNR value indicates the less difference between the reconstructed image and the original one.

We have performed the image reconstructions from Bessel-Fourier moments on two different computer systems for comparison, with precision set to 32-bit.

- System I: a laptop equipped with an Nvidia GTX 1050Ti, 16GB RAM and an 4-core Intel I7-8750H of 2.20GHz,
- System II: a Google Cloud Instance equipped with an Nvidia Tesla T4, 32GB RAM and an 8-core Intel Xeon of 2.30GHz.

Figure 5.2 and 5.3 show the reconstructed Figure 5.1 from Bessel-Fourier moments of orders 80 to 200 and 400 to 1,000, respectively. The numerical schemes ranges from 3 to 15.

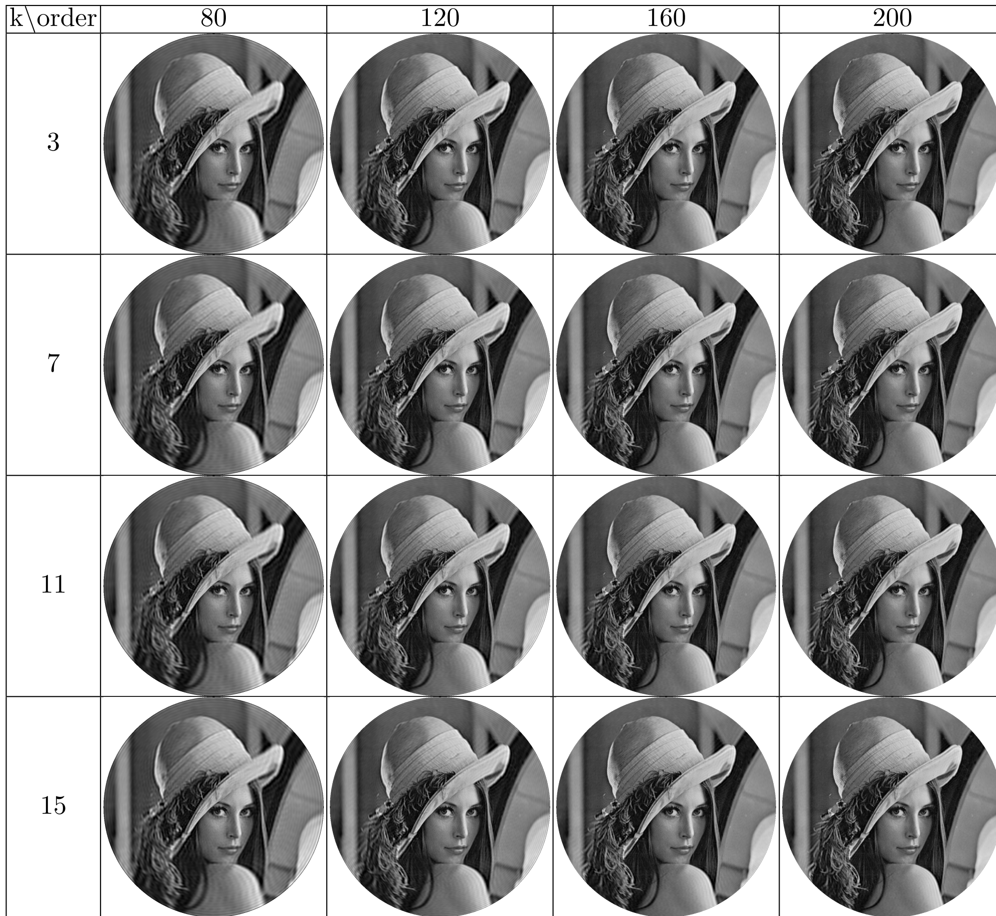


Figure 5.2: Reconstructed Figure 5.1 using different numerical schemes with 32-bit precision from Bessel-Fourier moments of order 80 to 200

5.1 Experiment

Shown in Table 5.1 are the PSNR values of the reconstructed images, with 2 digits preserved after the decimal point. As order increases, higher k is needed for better reconstruction qualities.

We have also done the reconstructions with 64-bit precision for compar-

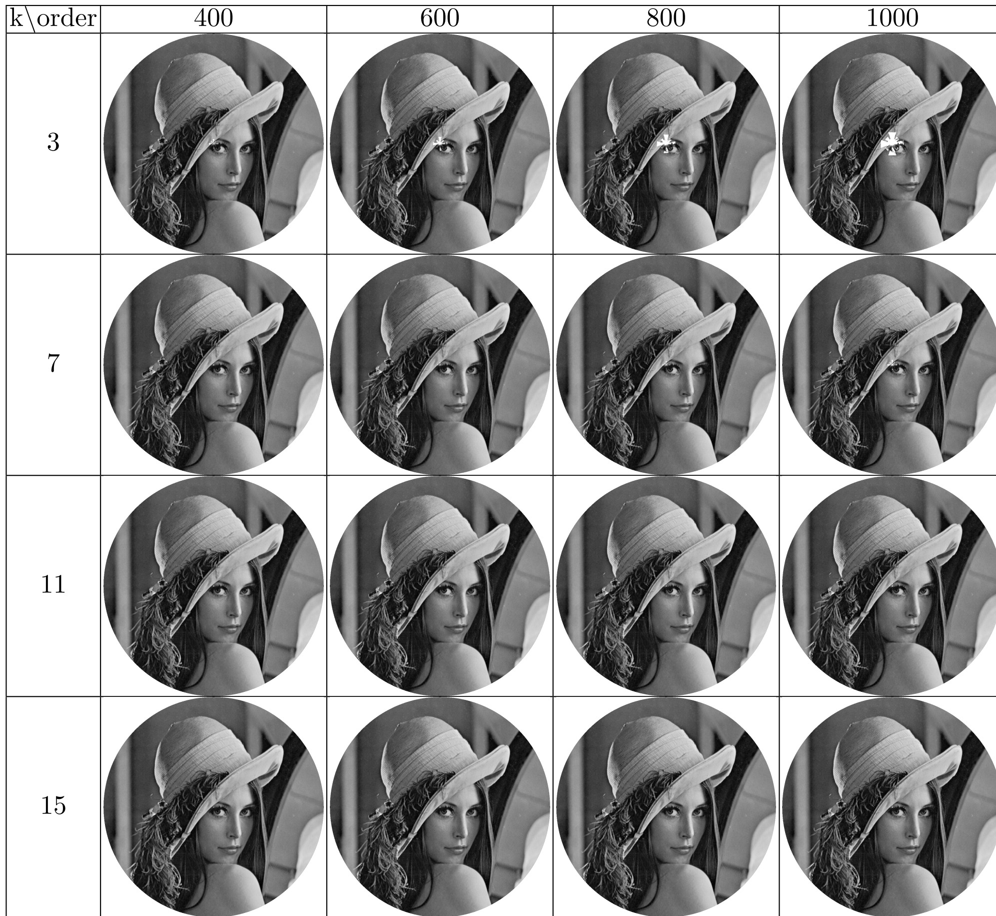


Figure 5.3: Reconstructed Figure 5.1 using different numerical schemes with 32-bit precision from Bessel-Fourier moments of order 400 to 1,000

ison. The results in Table 5.2 show that the 64-bit precision takes much longer computing time than those of 32-bit precision in Table 5.3 on every task. For example, when $k = 15$ and order $= 1,000$, the computing time for 64-bit precision is about 10 times longer even on System II, while there is no improvement on the corresponding PSNR values.

Table 5.4 shows a breakdown of the computing time for JV_KERNEL, EXP_KERNEL, BNM_KERNEL and the reconstruction, which reveals that the computation of $\boldsymbol{B}_{\mathbf{nm}}$ is significantly more time consuming. It also can

Table 5.1: PSNR values of applying different numerical schemes with 32-bit precision

	80	120	160	200	240	400	600	800	1000
1	26.16	28.33	29.75	30.48	30.40	26.21	21.61	18.10	11.31
3	26.67	29.30	31.57	33.60	35.18	38.22	35.50	32.34	29.38
5	26.75	29.43	31.78	33.95	35.72	40.33	40.19	38.20	35.73
7	26.79	29.49	31.86	34.07	35.91	40.97	42.37	41.48	39.33
9	26.82	29.53	31.93	34.17	36.03	41.13	43.32	43.42	41.72
11	26.83	29.56	31.96	34.22	36.10	41.24	43.97	44.20	43.34
13	26.84	29.57	31.98	34.24	36.13	41.37	44.13	45.14	44.01
15	26.85	29.58	31.99	34.26	36.16	41.45	44.27	45.79	44.55

be observed that the time of kernel computing increases more considerably than that of data transfer as precision becomes higher.

Theoretically, the number of computing operations equals to k^2 and the time complexity of our algorithm is $O(k^2)$. According to Table 5.2 and 5.3, the computing time grows slower than the number of sub-pixels in a pixel. For instance, in Table 5.2, with order equal to 1000, the computing time increases from 209.30 to 14773.64 seconds while a 225 time increase is seen for k^2 when k grows from 1 to 15. The computing time grows every slower with 32-bit precision in Table 5.3.

Since there has not been any report about the results of Bessel-Fourier moments computing and image reconstructions of high orders. It is hard to carry out a comparative study on the performance of image reconstruction.

5.2 Summary

In this chapter, we first demonstrated the scalability of our parallel algorithm by performing image reconstruction on two systems. As the compu-

tational complexity increases, the system with higher computing resources can reduce the computing time by larger percentage. On both systems, the computing time grows much slower than the number of sub-regions in a pixel does.

The precision-related computing performance is also investigated. Our experimental results suggest that both 32-bit and 64-bit precision provide the same computational accuracy in computing Bessel-Fourier moments of orders from 80 to 1000, while the computing time is longer for 64-bit precision.

Table 5.2: Reconstruction times (in seconds) of applying different numerical schemes with 64-bit precision from order 80 to 1000 on System II

k\order	80	120	160	200
1	6.72	10.26	14.67	19.13
3	13.81	20.93	33.62	45.42
5	28.12	42.10	71.51	97.82
7	49.76	74.47	128.88	176.72
9	79.30	117.48	205.11	281.58
11	114.98	171.12	300.13	413.42
13	158.68	235.28	414.97	571.55
15	209.89	311.28	548.83	756.75
k\order	400	600	800	1000
1	50.01	91.49	145.44	209.30
3	141.22	286.39	483.23	729.70
5	322.27	675.38	1160.11	1769.07
7	594.69	1261.17	2175.30	3328.16
9	958.28	2040.38	3529.70	5409.28
11	1404.66	3015.76	5225.11	8008.10
13	1945.71	4187.63	7257.12	11131.78
15	2580.52	5551.94	9628.64	14773.64

Table 5.3: Reconstruction times (in seconds) of applying different numerical schemes from order 80 to 1000 on both systems with 32-bit precision on both systems

k\order	80		120		160		200	
	I	II	I	II	I	II	I	II
1	3.06	3.40	4.85	4.88	7.09	6.72	10.36	8.68
3	4.56	4.63	7.88	7.07	11.72	9.76	16.88	12.71
5	7.85	7.48	13.90	11.51	22.43	15.80	32.60	20.83
7	13.98	11.82	25.34	18.18	34.69	25.06	50.76	33.20
9	19.34	17.70	36.27	27.27	55.15	37.56	80.50	50.41
11	27.51	25.13	53.26	38.91	85.40	53.06	116.94	71.26
13	41.08	34.46	70.28	52.32	107.10	71.90	160.61	96.74
15	51.33	44.35	93.11	68.37	153.42	93.26	213.79	126.08
k\order	400		600		800		1000	
	I	II	I	II	I	II	I	II
1	31.07	20.78	63.41	37.01	110.41	57.28	168.38	81.64
3	54.67	31.82	112.74	58.84	194.74	92.71	298.77	134.95
5	106.40	53.66	226.24	102.55	414.07	160.23	567.99	234.25
7	171.16	86.11	364.06	168.44	631.78	263.55	959.19	386.46
9	273.30	130.03	582.32	256.38	965.04	399.85	1494.69	590.58
11	395.26	185.86	808.45	366.66	1388.19	574.62	2130.56	845.97
13	535.27	252.10	1156.07	499.41	1898.63	780.58	2916.74	1150.95
15	682.04	330.05	1453.13	654.02	2633.60	1023.67	3828.35	1508.32

Table 5.4: The breakdown of computing time on System II (in seconds), when $k = 15$ and $n = m = 1000$

	32bit		64bit	
step	kernel	data transfer	kernel	data transfer
JV_KERNEL	2.52	105.7	42.45	208.44
EXP_KERNEL	1.41	212.05	7.09	424.41
BNM_KERNEL	1139.36	56.24	13940.2	113.95
RECONSTRUCTION-_KERNEL	44.8	30.18	84.26	57.84

Chapter 6

Filtering of Color Images in the Domain of Bessel-Fourier Moments

Filtering is a technique commonly used in the Fourier Frequency domain, where certain image information gets filtered, weakened or strengthened depending on the type of filters applied. As a tool for pattern recognition, Bessel-Fourier moments could be filtered so that only desirable image information gets kept for further analysis.

In daily life, most images contain color information. A digit color image consists of three channels: Red, Green and Blue, each of which can be represented by a 2-D matrix of intensity values from 0 to 255.

Similar to the filtering in the Fourier Frequency domain, a filter function $H(m,n)$ in the Bessel-Fourier moments domain is defined as

$$\widehat{f'}(x_i, y_j) = H(m,n) \sum_{n=0}^{N} \sum_{m=-M}^{M} \widehat{B}_{mn} J_1(\lambda_n r) \exp(-jm\theta). \tag{6.1}$$

In this research, four filters [8] are proposed and applied to the sample images in Figure 6.3:

- Ideal Lowpass Filter (ILF)

$$H(u,v) = \begin{cases} 1 & \text{if } D(u,v) \leqslant D_0 \\ 0 & \text{otherwise,} \end{cases} \tag{6.2}$$

- Ideal Highpass Filter (IHF)

$$H(u,v) = \begin{cases} 1 & \text{if } D(u,v) > D_0 \\ 0 & \text{otherwise,} \end{cases} \tag{6.3}$$

- Gaussian Lowpass Filter (GLF)

$$H(u,v) = e^{-D^2(u,v)/2D_0^2}, \tag{6.4}$$

- Gaussian Highpass Filter (GHF)

$$H(u,v) = 1 - e^{-D^2(u,v)/2D_0^2}, \tag{6.5}$$

where $D(u,v)$ is used to denote the distance of point (u,v) from the origin in the Bessel-Fourier moments domain, and D_0 is the "cutoff" distance. For better computational accuracy, a 7×7 numerical schema is used.

Figure 6.1 illustrates filtering in the Fourier Frequency and Bessel-Fourier moments domains. In Figure 6.1, the radius of purple transparent circles represent the cutoff range. According to Equation (3.2), m can only be a non-negative integer. Therefore, by setting $m = n$, only half of the area in Figure 6.1 (b) has Bessel-Fourier moment data which is highlighted in blue.

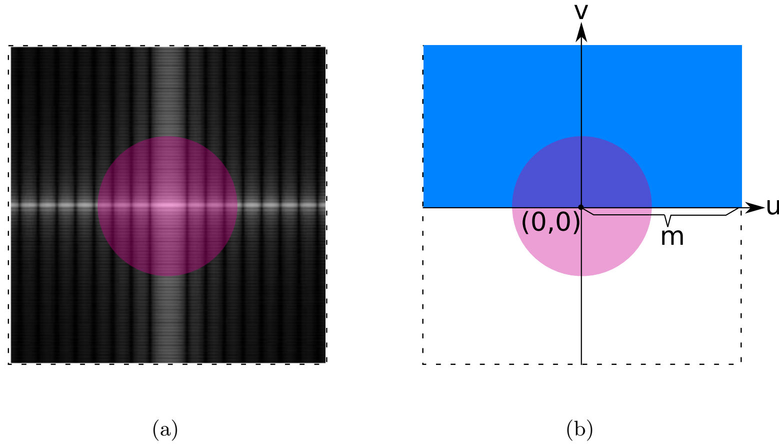


Figure 6.1: Filtering in (a) Fourier Frequency domain and (b) Bessel-Fourier moments domain.

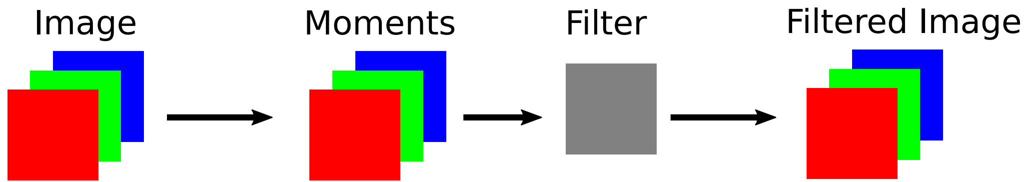
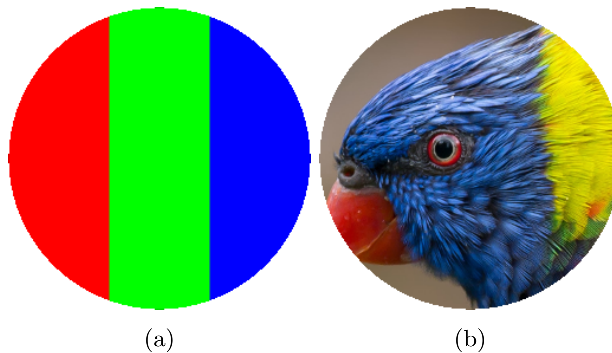


Figure 6.2: The Filtering Workflow

Figure 6.2 illustrates the flowchart of color image filtering. The Bessel-Fourier moments for each channel of a color image are firstly calculated separately. Then, the Bessel-Fourier moments of three channels are filtered. Finally, the filtered Bessel-Fourier moments of three channels are reconstructed individually and merged to yield a filtered color image.

Figure 6.3 displays the two testing images used in our experiment. Figure 6.3 (a) contains three colors of red, blue and green without any intersection area, and Figure 6.3 (b) is the image of a bird with more details.



(a) (b)

Figure 6.3: The two testing images are sized at 256×256 with 256 RGB densities

Figure 6.4 shows the reconstructed images from different channels of Bessel-Fourier moments with the maximum order of 150, where $m = n = 150$, and the combined color image.

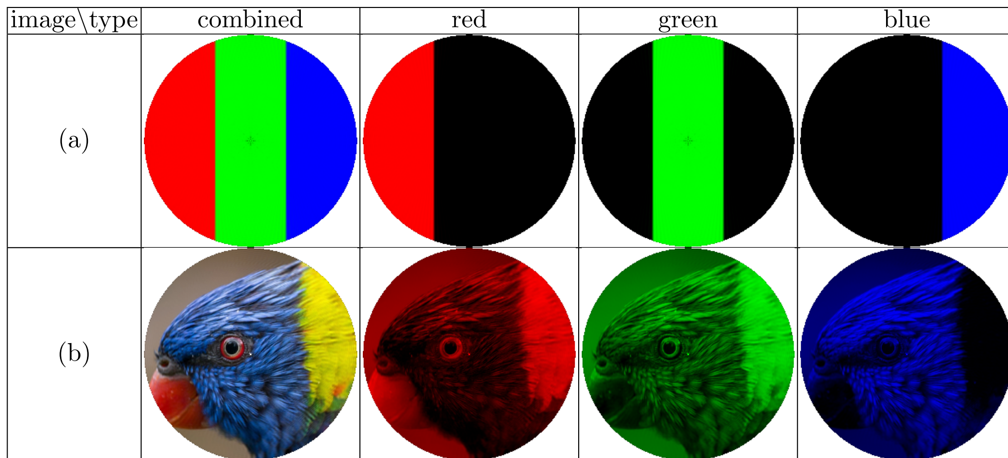


Figure 6.4: Reconstruction results of the two testing images

Table 6.1 lists the corresponding PSNR values of the reconstructed images shown in Figure 6.4. It can be observed that the reconstructions are satisfactory with all the PSNR values above 30. For the same original image, there is no significant difference among the PSNR values of its three channels and the combined one. Since Figure 6.3 (b) contains more details, its reconstructed images have lower PSNR values.

Table 6.1: The PSNR values for the red, green, blue, and combined channels for the two images in Figure 6.3.

image\type	combined	red	green	blue
(a)	37.25	37.17	37.40	37.17
(b)	31.60	31.66	31.33	31.81

For comparison, we have performed the filtering on the testing images in both the Bessel-Fourier moments and Fourier Frequency domains. To perform filtering in the Fourier Frequency domain, it is common to set $0 < D_0 < N$, where N is the width of image. However, the dimension of Bessel-Fourier moment matrix is dependent on the specified order, which is 150×301 when $n = 150$. Therefore, it is more feasible to use a percentage value α to

calculate D_0 dynamically as in

$$D_0 = \alpha\, r', \tag{6.6}$$

where

$$r' = \begin{cases} n & \text{if in Bessel-Fourier moments domain,} \\ N & \text{otherwise.} \end{cases}$$

6.1 Ideal Filters

The results of applying ILF to sample images in the Bessel-Fourier moments domain and the Fourier Frequency domain are shown in Figures 6.5 and 6.6. It can be observed that Bessel-Fourier moments and frequencies close to the origin in the Fourier Frequency domain correspond to the slowly varying components of the image.

In Figure 6.5, a small area in the central part is missing for all the 3 channels when $\alpha = 0.1$. As α increases, the central area and more details start to show up. The phenomenon is not observed in Figure 6.6, which shows the results of applying ILFs in Fourier Frequency domain. We can also see that the results for ILFs in Bessel-Fourier domain have more ringing effects than those in Fourier Frequency domain.

Figure 6.7 and Figure 6.8 display the results of applying the Ideal Highpass Filters in both of Bessel-Fourier moments and Fourier Frequency domains.

From Figure 6.7 and Figure 6.8, it can be seen that both the higher orders in the Bessel-Fourier moments domain and the higher frequencies close to the origin in Fourier Frequency domain correspond to changes in the intensity such as borders and textures. The central missing area in Figure 6.5 is present in Figure 6.7.

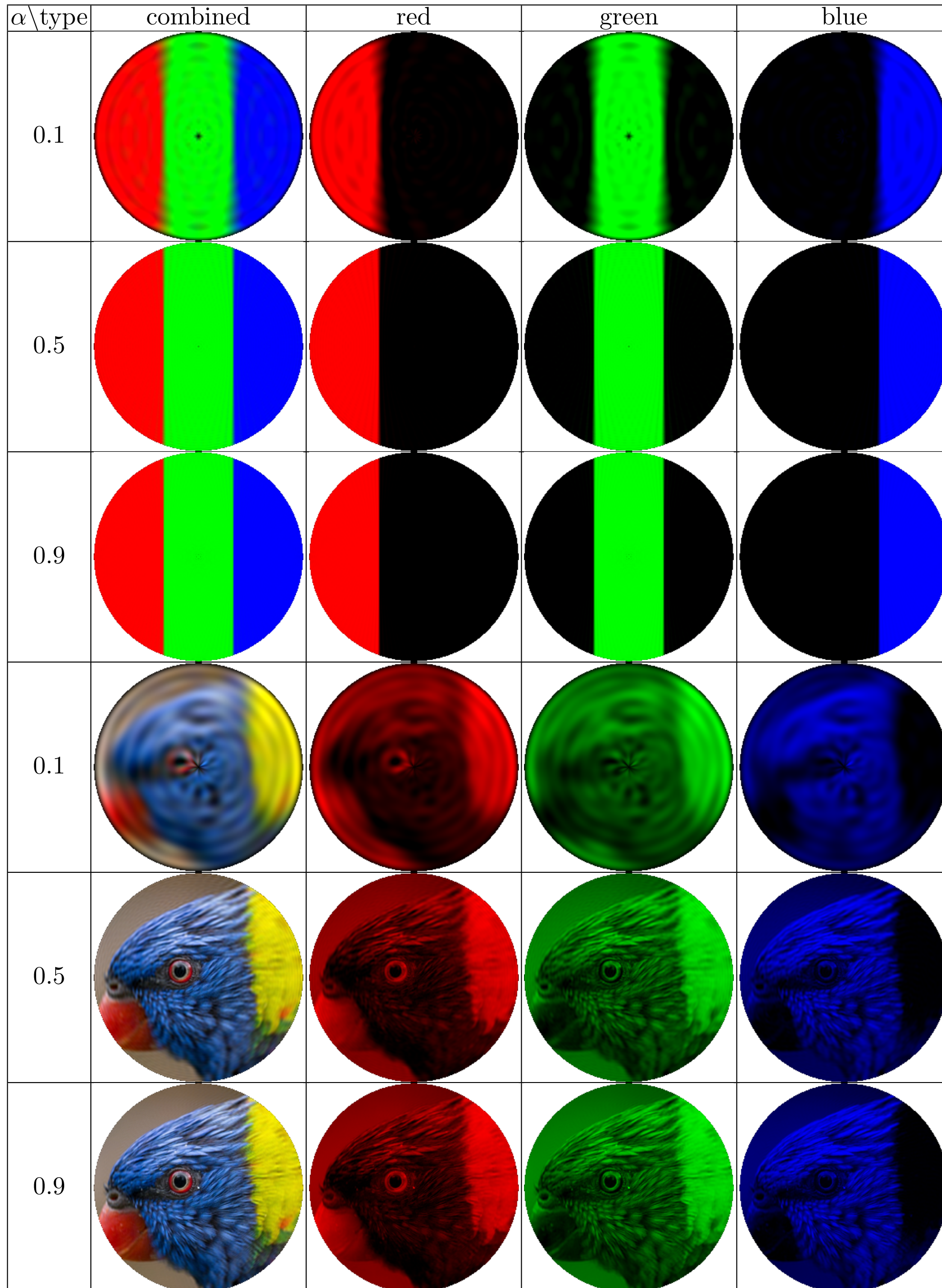


Figure 6.5: Results of applying ILF in the Bessel-Fourier moments domain

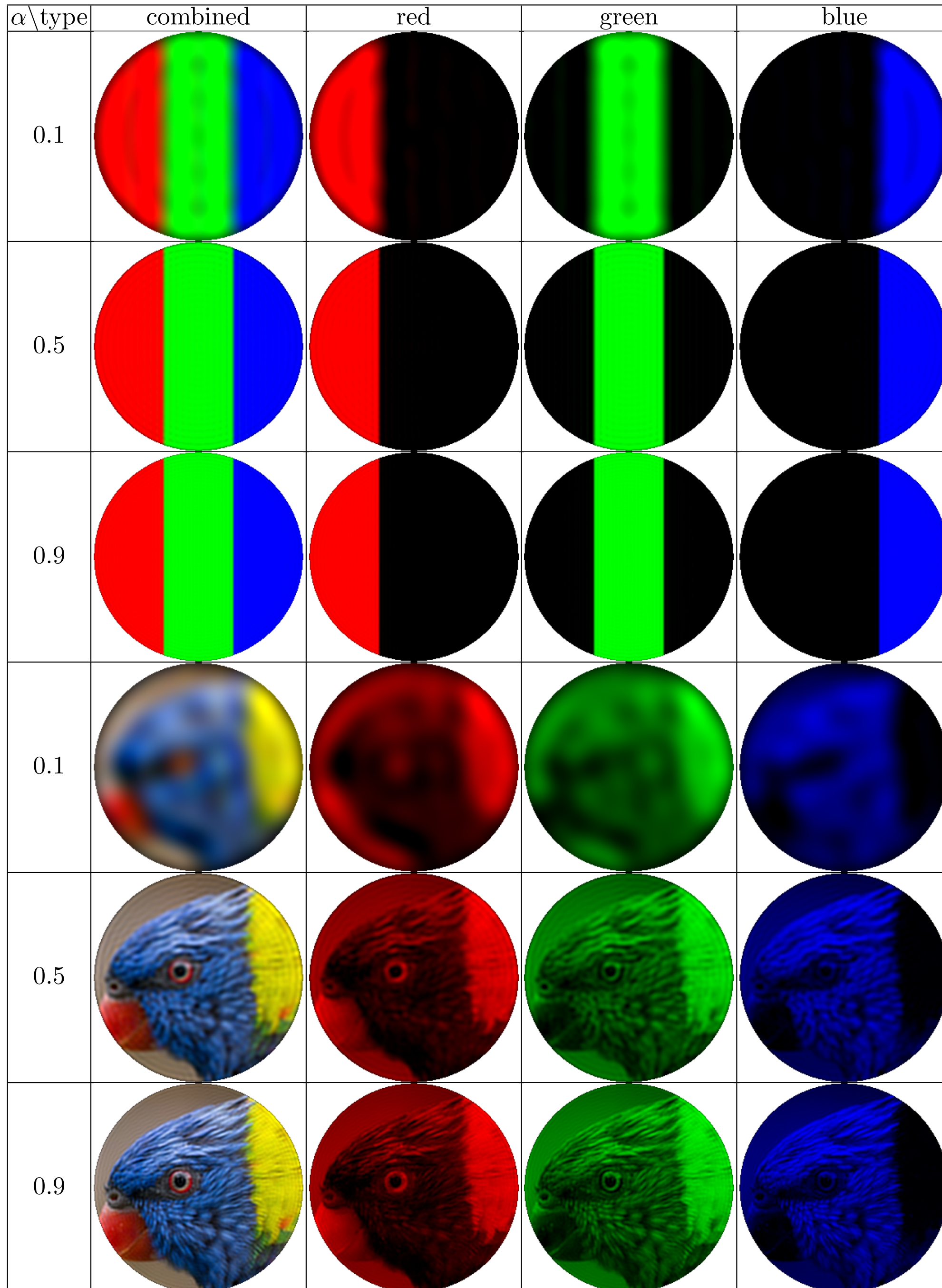


Figure 6.6: Results of applying ILF in the Fourier Frequency domain

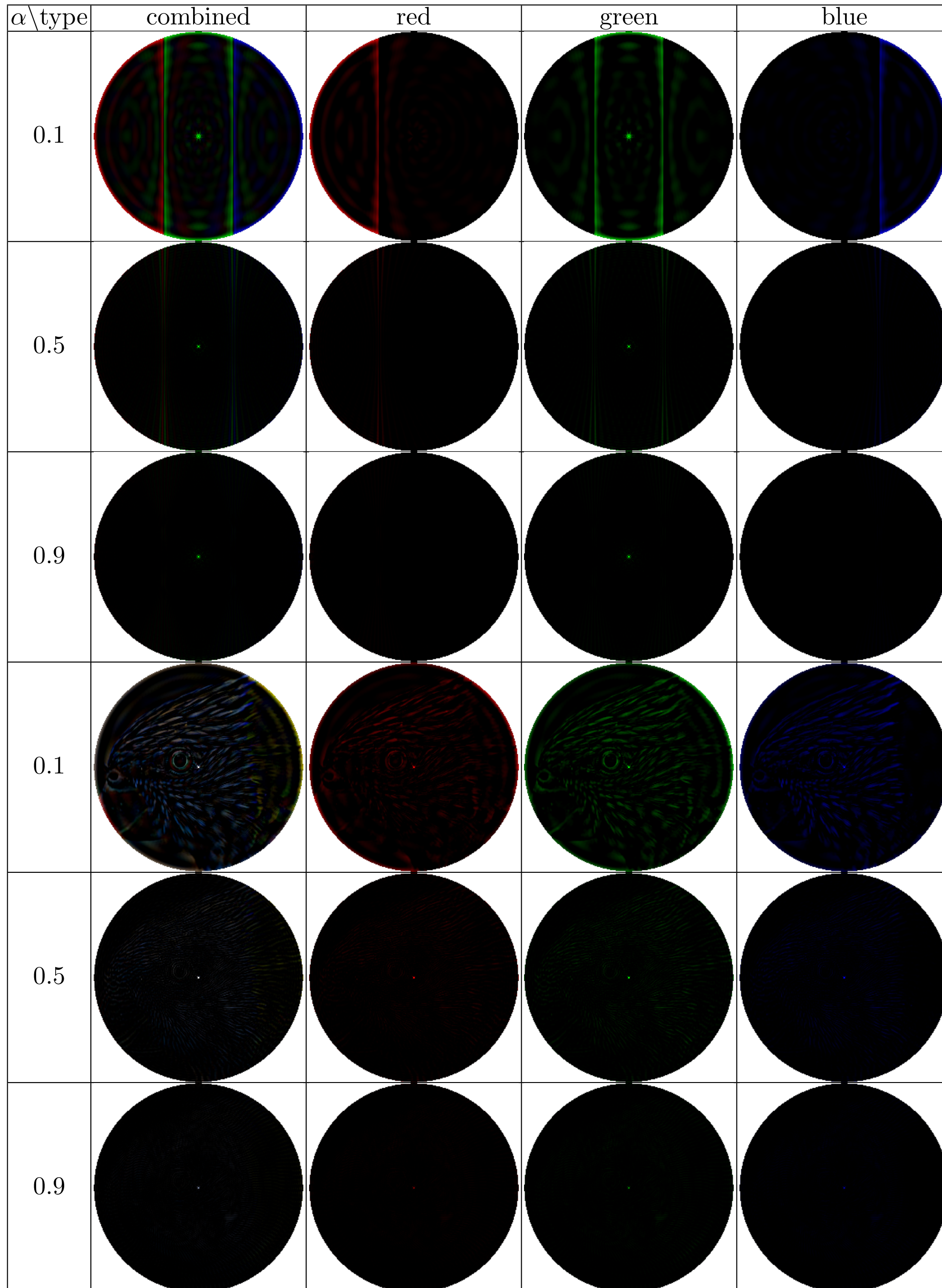


Figure 6.7: Results of applying IHF in the Bessel-Fourier moments domain

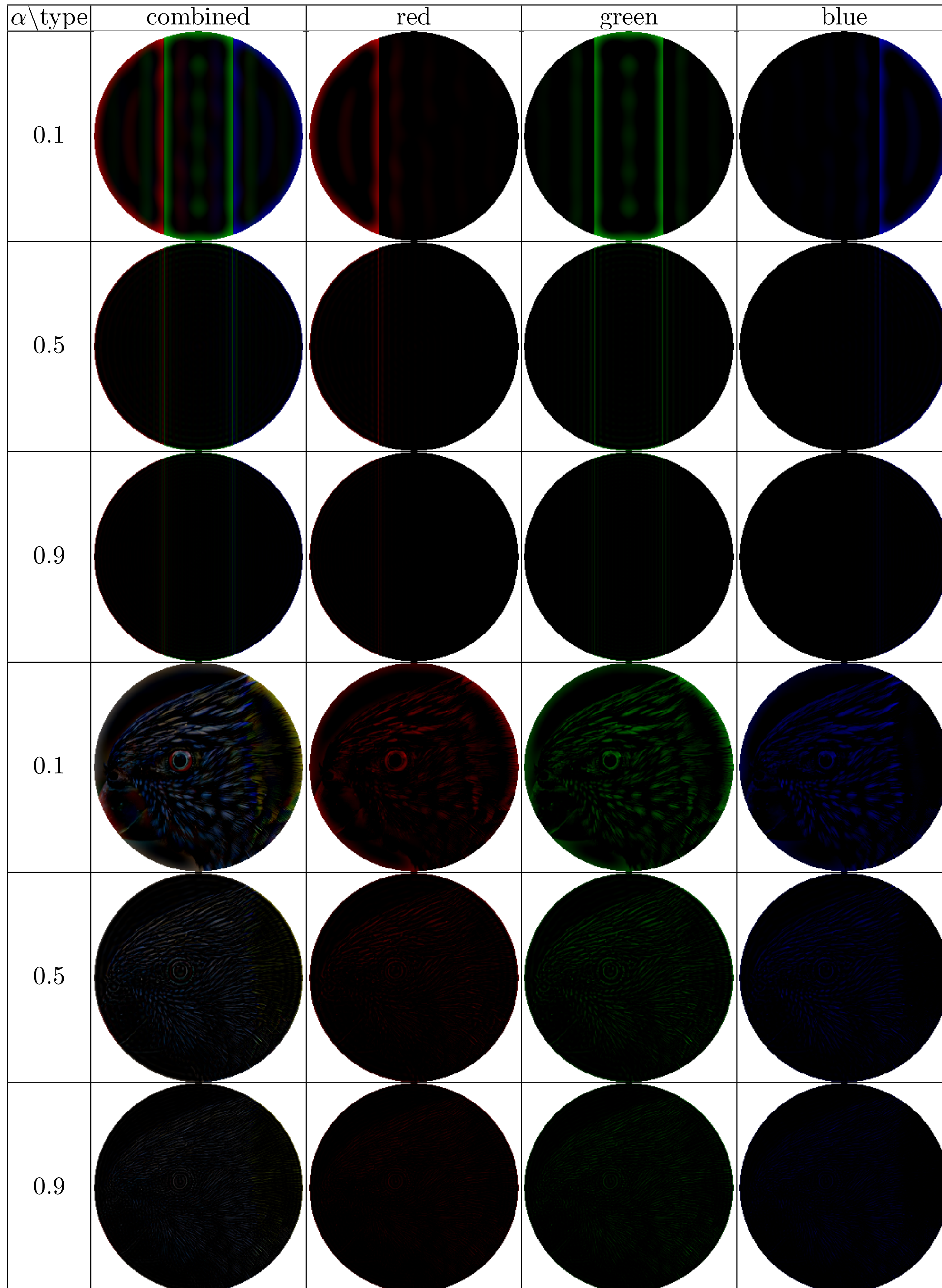


Figure 6.8: Results of applying IHF in the Fourier Frequency domain

To quantitatively study the effects of ideal filters, we used the following equations to compute the percentages of image power for IHF in the respective domain:

- Fourier Frequency domain

$$P_f = \frac{\sum_u^N \sum_v^N |F(u,v)|^2 H(u,v)}{\sum_u^N \sum_v^N |F(u,v)|^2}, \tag{6.7}$$

- Bessel-Fourier moments domain

$$P_b = \frac{\sum_{u=0}^n \sum_{v=-m}^m |B_{nm}|^2 H(u,v)}{\sum_{u=0}^n \sum_{v=-m}^m |B_{nm}|^2}, \tag{6.8}$$

where P_f and P_b represents the percentages of image power in the Fourier Frequency domain and Bessel-Fourier moments domain respectively, and $F(u,v)$ is the Fourier component.

Figure 6.9 and Figure 6.10 demonstrate the percentages of image power for Figure 6.3 (a) in Bessel-Fourier moments domain and Fourier Frequency domain, respectively. For all of the three channels and the combine color image, there is more power in the Bessel-Fourier moments domain than that in Fourier Frequency domain with the same α value.

As α increases, the percentages of image power starts to drop. Drastic decreases can be seen for both images when α is raised to 0.5. Since the percentages of image power for IHF can be computed by subtracting those for ILF from 1, they are not listed here.

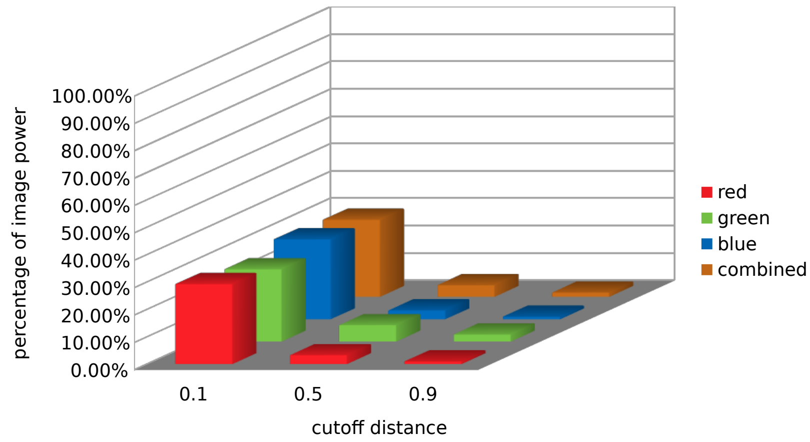


Figure 6.9: The percentage of image power for Figure 6.3(a) in the Bessel-Fourier moments domain with cutoff distances set to 0.1, 0.5 and 0.9

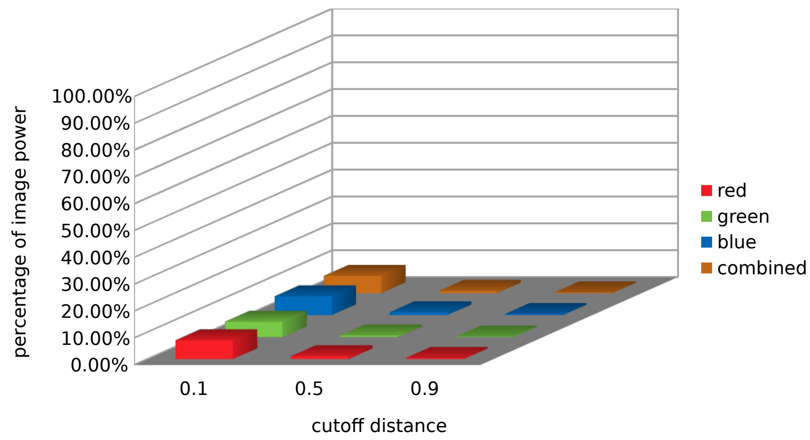


Figure 6.10: The percentage of image power for Figure 6.3(a) in the Fourier Frequency domain with cutoff distances set to 0.1, 0.5 and 0.9

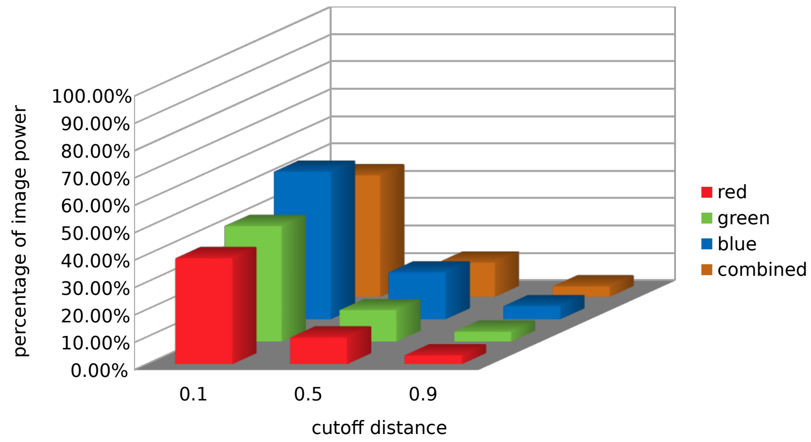


Figure 6.11: The percentage of image power for Figure 6.3(b) in the Bessel-Fourier moments domain with cutoff distances set to 0.1, 0.5 and 0.9

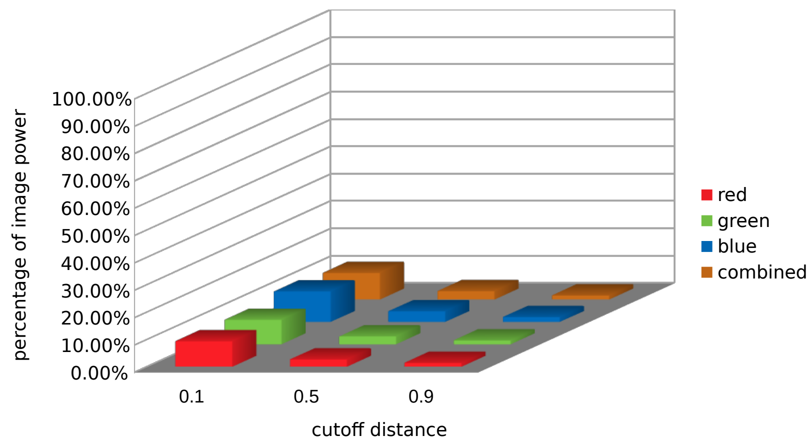


Figure 6.12: The percentage of image power for Figure 6.3(b) in the Fourier Frequency domain with the cutoff distance set to 0.1, 0.5 and 0.9

6.2 Gaussian Filters

Results of applying Gaussian Lowpass Filters in Bessel-Fourier moments and Fourier Frequency domains are shown in Figure 6.13 and Figure 6.14, while Figure 6.15 and Figure 6.14 demonstrate the results of applying Gaussian Highpass Filters in Bessel-Fourier moments domain and Fourier Frequency domain.

Compared with the results of applying ILFs, the ringing effect has been substantially reduced in both Bessel-Fourier moments and Fourier Frequency domains. When α equals to 0.1, the ringing effect is nearly invisible in Figure 6.13 and Figure 6.14. With most of the ringing removed, we can see that the results for GLFs in Bessel-Fourier moments domain are less blurry than those in Fourier Frequency domain, especially around the bird eye.

6.3 Summary

In this chapter, we proposed Ideal and Gaussian filters in the Bessel-Fourier moments domain and performed image filtering with two sample images. To compare the results, filtering was also done in the Fourier Frequency domain. It is observed that Bessel-Fourier moments appear to share some similarities with frequencies in the Fourier transform. Bessel-Fourier moments of lower orders corresponds to smooth intensity parts of images, while those of higher orders are more related to details. In addition, Ideal Lowpass Filter creates a ringing effect, while Gaussian Lowpass Filter can be used to reduce the ringing.

We also found specific behaviors of Bessel-Fourier moments. Most of the information in the central small area is missing for lower orders. Bessel-Fourier moments have a higher percentage of image power than the frequencies in the Fourier Frequency domain, particularly for lower cutoff distances.

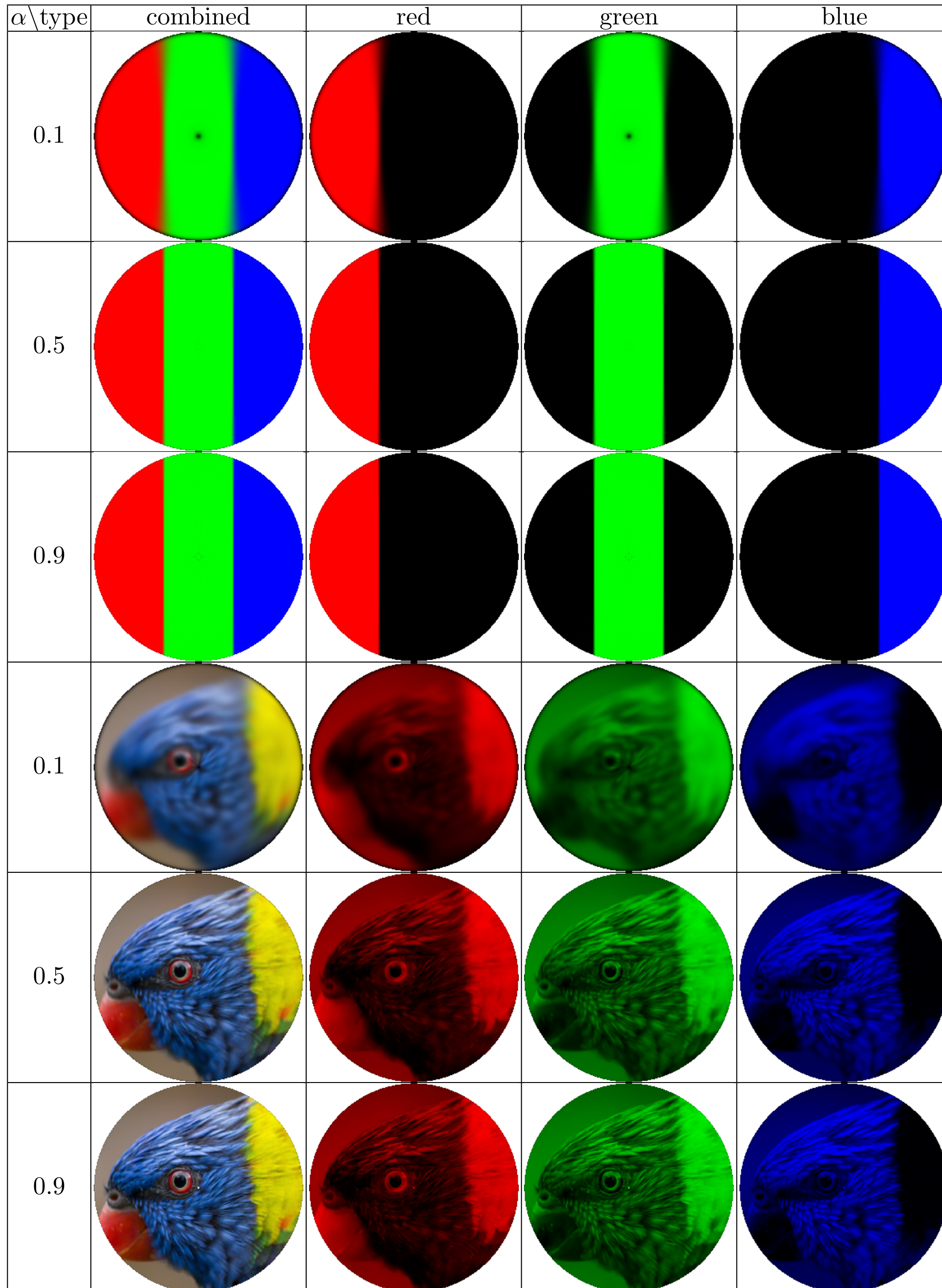


Figure 6.13: Results of applying GLF in the Bessel-Fourier moments domain

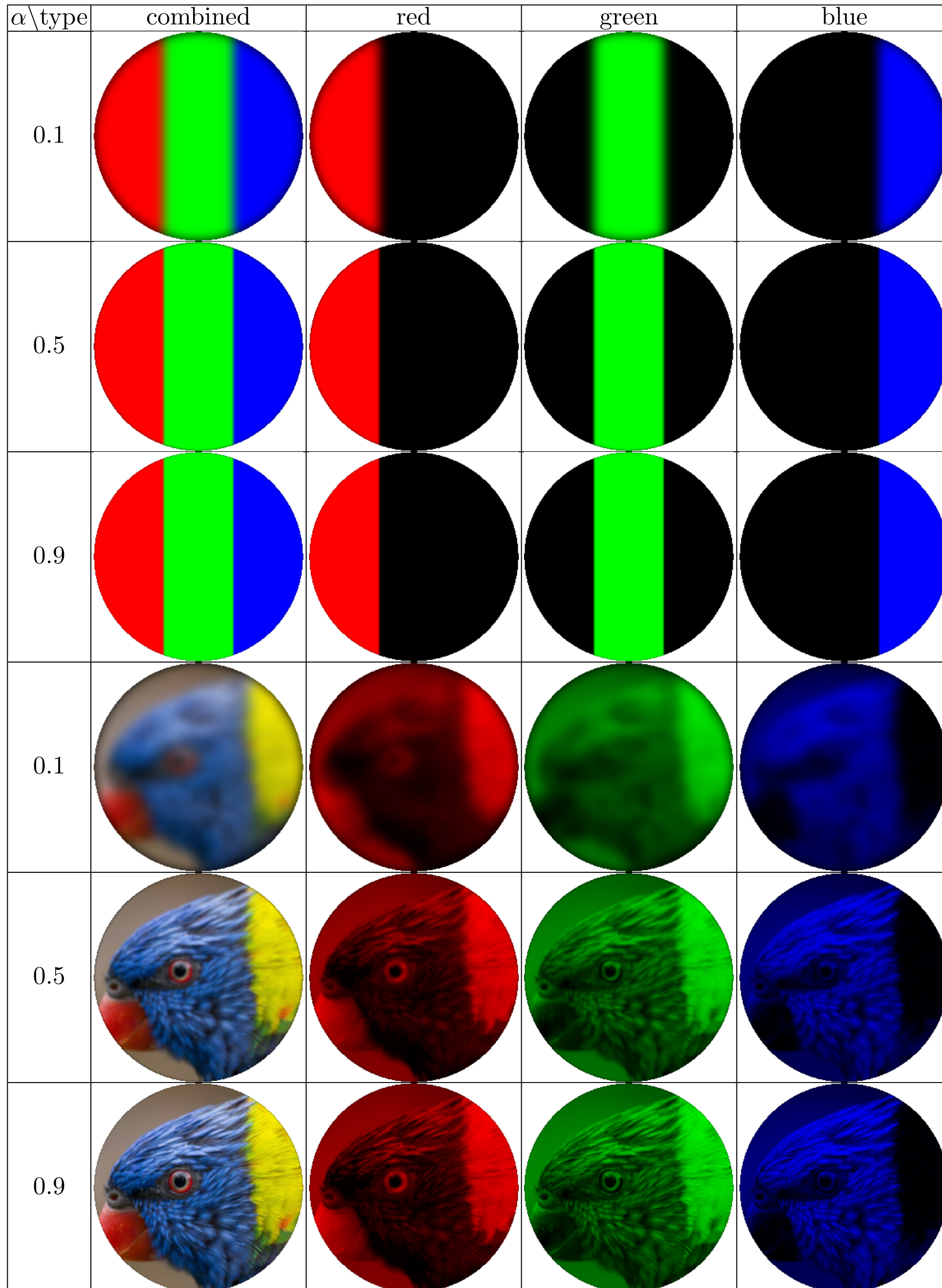


Figure 6.14: Results of applying GLF in the Fourier Frequency domain

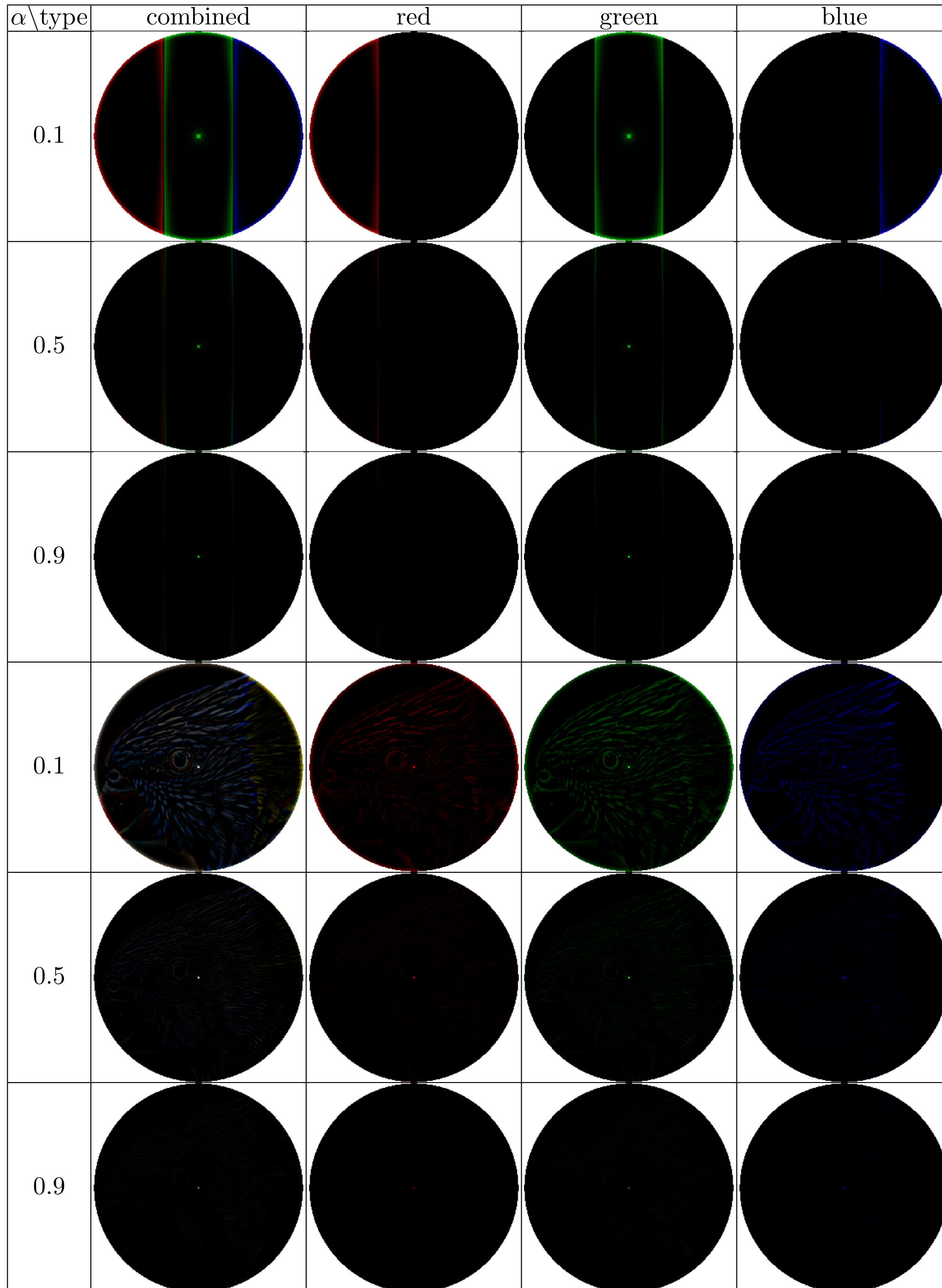


Figure 6.15: Results of applying GHF in the Bessel-Fourier moments domain

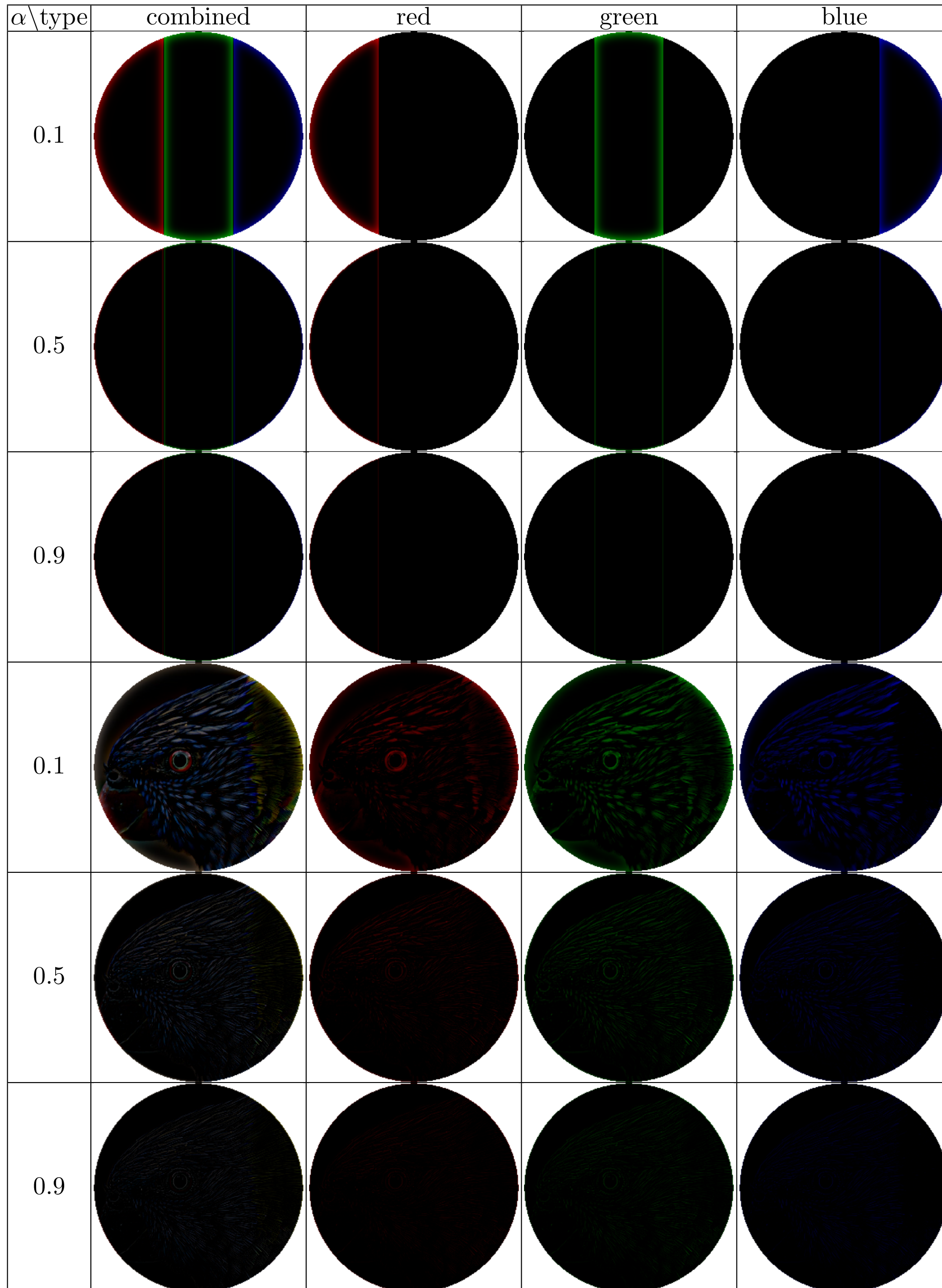


Figure 6.16: Results of applying GHF in the Fourier Frequency domain

Chapter 7

Concluding Remarks

7.1 Summary

In this research, we have proposed a GPU-based algorithm to compute Bessel-Fourier moments more efficiently with little compromise in accuracy. The symmetric algorithm is used to reduce the data transfer overhead by almost 7/8. Image pixels are also reordered to reduce the branch divergences without introducing extra data. By leveraging the constant and shared memories with the tiled algorithm, further optimization is achieved.

To benchmark the performance of the algorithm, image reconstructions were performed on two systems. On the system with better hardware, less computing time is required to perform reconstructions, which demonstrates the scalability of our algorithm. On both systems, the computing time increases more slowly than the number of sub-regions. From the breakdown of time consumption, the computation of the moment matrix takes up the largest part of the computing time.

We also investigated the performance in relation to precision. The results lead to the discovery that reconstructions with 64-bit precision yielded no noticeable quality improvement in comparison to 32-bit precision. Thus, since 64-bit precision is significantly more time-consuming, 32-bit precision would be recommended for most cases.

Furthermore, Ideal and Gaussian filters were introduced in the Bessel-Fourier moments domain. Filtering was also performed in the Fourier domain for comparison. The results show that Bessel-Fourier moments of lower orders

correspond to the slowly varying parts of images, while the moments of higher orders are more related to details such as borders and textures. In both the Bessel-Fourier moments domain and the Fourier domain, the Ideal Lowpass Filter can give rise to a ringing effect, while the Gaussian Lowpass Filter can reduce the ringing.

Lastly, we studied the image power of the Bessel-Fourier moments and found that the percentage of image for the Bessel-Fourier moments is greater than that of the frequencies in the Fourier domain for lower cutoff distances.

7.2 Future Work

Our results demonstrate a satisfactory performance of our newly proposed algorithm with little loss in the computational accuracy. Based on our work, some further areas should be explored in the future.

Extending our algorithm to other orthogonal moments defined in a circular domain

The computation of most orthogonal moments defined in a circular domain can benefit from the symmetric properties, because their kernel functions take radius and angle as input parameters. In addition, since the matrix operations can be used for computing other orthogonal moments, like the Exponent-Fourier moments, a similar parallel algorithm could lead to significant performance gains.

Use of multi-GPU architecture

Although several optimizations were made in this research, we did not take advantage of multiple GPUs, which would improve the system through-

put by increasing the available memory. At the same time, if the algorithm was implemented in CUDA-powered GPUs, the use of NVLink could lead to even higher bandwidths and lower latencies.

Appendix A

Source Code for GPU Kernels

The CUDA C++ source code snippets are given in this section. Since some of the variable types remain unknown until the initialization of kernels, we used Jinja as the template language for kernel codes [12]. Jinja variables are enclosed by {{}}.

Below is a list of Jinja variables present in the code snippets:

- float_type - the type of floating numbers,
- ph_len - the number of pixels to be processed in each iteration of the tiled algorithm,
- bw - the width of CUDA thread blocks,
- group_size - the group size by which the summation is done, which is set to k^2.

```
__global__ void radius_kernel(
    {{float_type}} * ar_radius_k,
    {{float_type}} * ar_radius,
    {{float_type}} * ar_radius_special,
    {{float_type}} * ar_radius_ndarray_mask,
    {{float_type}} * ar_radius_ndarray_mask_for_k,
    long long * width,
    long long * k,
    long long * width_special_scaled,
    long long * middle,
```

```
    long long * middle_k,
    long long * middle_special_scaled,
    float * special_to_full_ratio,
    long long * special_start,
    long long * special_end
){
    const long long tx = threadIdx.x;
    const long long ty = threadIdx.y;
    const long long bw = blockDim.x;
    const long long bx = blockIdx.x;
    const long long by = blockIdx.y;
    const long long col = bx*bw + tx;
    const long long row = by*bw + ty;
    const long long special_start_local = *special_start;
    const long long special_end_local = *special_end;

    const long long middle_local = *middle;
    const long long middle_k_local = *middle_k;
    const long long middle_special_scaled_local =
        *middle_special_scaled;
    const float special_to_full_ratio_local =
        *special_to_full_ratio;
    long long pos_in_area;
    {{float_type}} radius;

    const long long width_k = *width * *k;

    // load ar_radius
    if(col < *width && row < *width){
        pos_in_area = row * *width + col;
        ar_radius_ndarray_mask[pos_in_area] = 1;
```

```
        ar_radius_ndarray_mask_for_k[pos_in_area] = 1;
        ar_radius[pos_in_area] = sqrt(
            powf(-row-1+ middle_local+(({{float_type}})1)/2, 2)
            +
            powf(col+1- middle_local-(({{float_type}})1)/2, 2)
        ) / middle_local;
    }
    __syncthreads();

    // load ar_radius_k
    if(col < width_k && row < width_k){
        const long long new_row = row / (*k);
        const long long new_col = col / (*k);

        radius = sqrt(
            powf(-row-1+ middle_k_local+1./2, 2) +
            powf(col+1- middle_k_local-1./2, 2)
        ) / middle_k_local;

        const long long pos_in_origin = new_row * *width
            + new_col;
        if(radius > 1 ){
            ar_radius_ndarray_mask[pos_in_origin] = 0.;
            ar_radius_ndarray_mask_for_k[pos_in_origin] = 0.;
        }
        if(
            new_row >= special_start_local &&
            new_row < special_end_local &&
            new_col >= special_start_local &&
            new_col < special_end_local
        ){
```

```
            ar_radius_ndarray_mask_for_k[pos_in_origin] = 0.;
        }
        pos_in_area = row * width_k + col;
        ar_radius_k[pos_in_area] = sqrt(
            powf(-row-1+ middle_k_local+1./2, 2) +
            powf(col+1- middle_k_local-1./2, 2))
                / middle_k_local;
    }

    // load ar_radius_special
    if(col < *width_special_scaled && row < *
   width_special_scaled){
        pos_in_area = row * *width_special_scaled + col;
        ar_radius_special[pos_in_area] = sqrt(
            powf(-row-1+ middle_special_scaled_local+1./2, 2) +
            powf(col+1- middle_special_scaled_local-1./2, 2)
        ) / middle_special_scaled_local *
            special_to_full_ratio_local;
    }
}
```

Kernel Code for RADIUS_KERNEL

```
__global__ void arctan_kernel(
    {{float_type}} * ar_arctan_k,
    {{float_type}} * ar_arctan,
    {{float_type}} * ar_arctan_special,
    long long * width,
    long long * k,
    long long * width_special_scaled,
    long long * middle,
    long long * middle_k,
    long long * middle_special_scaled
){
    const long long tx = threadIdx.x;
    const long long ty = threadIdx.y;
    const long long bw = blockDim.x;
    const long long bx = blockIdx.x;
    const long long by = blockIdx.y;
    const long long col = bx*bw + tx;
    const long long row = by*bw + ty;

    long long pos_in_area;
    {{float_type}} angle;

    const long long width_k = *width * *k;
    const {{float_type}} pi = atan(1.)*4;

    // load ar_arctan_k
    if(col < width_k && row < width_k){
        pos_in_area = row * width_k + col;
        angle = atan2(-row-1+ *middle_k+1./2, col+1-
            *middle_k-1./2);
```

```
        ar_arctan_k[pos_in_area] = angle < 0 ? angle +
            2*pi : angle;
    }

    // load ar_arctan
    if(col < *width && row < *width){
        pos_in_area = row * *width + col;
        angle = atan2(-row-1+ *middle+1./2, col+1-
            *middle-1./2);
        ar_arctan[pos_in_area] = angle < 0
            ? angle + 2*pi : angle;
    }


    // load ar_arctan_special
    if(col < *width_special_scaled && row < *
   width_special_scaled){
        pos_in_area = row * *width_special_scaled + col;
        angle = atan2(-row-1+ *middle_special_scaled+
            1./2, col+1- *middle_special_scaled-1./2);
        ar_arctan_special[pos_in_area] = angle < 0
            ? angle + 2*pi : angle;
    }
}
```

Kernel Code for ARCTAN_KERNEL

```
__constant__ {{float_type}} zero_crossings[2048];
__global__ void jv_kernel({{float_type}} * ar_c, {{float_type}}
    * ar_a, long long * n, long long * pixels_in_part
){
        const long long tx = threadIdx.x;
        const long long ty = threadIdx.y;
        const long long bw = blockDim.x;
        const long long bh = blockDim.y;
        const long long bx = blockIdx.x;
        const long long pos = tx + ty*bw + bx*bw*bh;
        const long long n_local = *n;
        const long long pixels_in_part_local = * pixels_in_part
   ;

        if(pos < n_local * pixels_in_part_local){
        long long idx_origin_y = pos / pixels_in_part_local;
        long long idx_origin_x = pos - idx_origin_y*
   pixels_in_part_local;
        ar_c[pos] = j1(ar_a[idx_origin_x] * zero_crossings[
   idx_origin_y]) / \
        (powf(jn(2, zero_crossings[idx_origin_y]), 2.)/2);
    }
}
```

Kernel Code for JV_KERNEL

```
__constant__ {{float_type}} zero_crossings[2048];
__global__ void exp_kernel({{float_type}} * ar_c, {{float_type
   }} * ar_a, long long * n, long long * pixels_in_part)
{
    const long long tx = threadIdx.x;
    const long long ty = threadIdx.y;
    const long long bw = blockDim.x;
    const long long bh = blockDim.y;
    const long long bx = blockIdx.x;
    const long long pos = tx + ty*bw + bx*bw*bh;
    const long long n_local = *n;
    const long long pixels_in_part_local = * pixels_in_part;

    if(pos < n_local * pixels_in_part_local){
        long long idx_origin_y = pos / pixels_in_part_local;
        long long idx_origin_x = pos -
            idx_origin_y*pixels_in_part_local;
        ar_c[pos] = j1(ar_a[idx_origin_x] *
            zero_crossings[idx_origin_y]);
    }
}
```

Kernel Code for EXP_KERNEL

```
#include <pycuda-complex.hpp>
typedef pycuda::complex<{{float_type}}> dcmplx;
__global__ void bnm_kernel(
    dcmplx *ar_bnm,
    {{float_type}} *ar_jv,
    {{float_type}} *ar_img_k,
    dcmplx *ar_exp,
    int *n,
    int *m,
    int *pixels_in_part,
    int *pixels_in_part_diagonal,
    int *pixels_in_part_no_diagonal
){
    const int bx = blockIdx.x;
    const int by = blockIdx.y;
    const int bw = blockDim.x;
    const int tx = threadIdx.x;
    const int ty = threadIdx.y;

    const int n_local = *n;
    const int m_local = *m;
    const int pixels_in_part_local = *pixels_in_part;
    const int pixels_in_part_no_diagonal_local =
        *pixels_in_part_no_diagonal;
    const int pixels_in_part_diagonal_local =
        *pixels_in_part_diagonal;
    const int ph_len = {{ph_len}};

    // identify the row and col num for the current thread
    const int row = by * bw + ty;
```

```
const int col = bx * bw + tx;
const long long pos = (long long)row * m_local + col;
const int col_as_m = col - m_local/2;

// for the imagine part of exp
const {{float_type}} exp_imagine_multiplier =
    col_as_m > 0 ? -1 : 1;

const int block_pos = tx + bw*ty;

dcmplx total = dcmplx(0,0);
dcmplx total_tmp = dcmplx(0,0);
int count = 0;

// allocate shared memory
__shared__ {{float_type}} ar_jv_ph[{{bw}}][ph_len];
__shared__ dcmplx ar_exp_ph[{{bw}}][ph_len];
__shared__ {{float_type}} ar_img_k_ph[8 * ph_len];

// loop over tiles in ar_jv and ar_exp to calculate
for (
    int ph = 0;
    ph <= (pixels_in_part_local-1)/ph_len;
    ph++
){
    // load data into shared momery && boundary checking
    const int iteLen = (ph+1)*ph_len > pixels_in_part_local
        ? pixels_in_part_local-ph*ph_len : ph_len;

    const int ph_start = ph * ph_len;
    const int ph_end = ph_start + iteLen;
```

```
int base_load_count = iteLen / bw +
    int(tx < iteLen % bw);

if(row < n_local){
    for(int i=0; i<base_load_count; i++){
        ar_jv_ph[ty][i*bw + tx] =
            ar_jv[(long long)row * pixels_in_part_local
            + ph*ph_len + i*bw + tx];
    }
}

base_load_count = iteLen / bw + int(ty < iteLen % bw);
if(col < m_local){
    dcmplx tmp_exp = dcmplx(0,0);
    for(int i=0; i<base_load_count; i++){
        tmp_exp = ar_exp[((long long)(m_local/2)-
            (int)fabs(({{float_type}})(col_as_m))) *
            pixels_in_part_local +
            ph*ph_len + i*bw + ty
        ];
        ar_exp_ph[tx][i*bw + ty] = dcmplx(
            tmp_exp.real(),
            exp_imagine_multiplier * tmp_exp.imag()
        );
    }
}

int pixels_diagonal_ph = fmaxf(fminf(ph_end -

    pixels_in_part_no_diagonal_local, iteLen), 0);
```

```
const int pixels_no_diagonal_ph = iteLen -
    pixels_diagonal_ph;

if(block_pos < iteLen){
    int pos_in_part = ph_start + block_pos;
    if(pos_in_part < pixels_in_part_no_diagonal_local){
        // load 8 octants if it's not diagonal
        for(int i=0; i<8; i++){
            ar_img_k_ph[
                i*pixels_no_diagonal_ph + block_pos
            ] =
            ar_img_k[(long long)pos_in_part +
            pixels_in_part_no_diagonal_local * i];
        }
    }else{
        // load 4 quarters
        for(int i=0; i<4; i++){
            ar_img_k_ph[
                7*pixels_no_diagonal_ph +
                pixels_diagonal_ph*i + block_pos
            ] = ar_img_k[
                (long long)pos_in_part +
                pixels_in_part_no_diagonal_local*7 +
                pixels_in_part_diagonal_local * i
            ];
        }
    }
}

__syncthreads();
```

```
    // do stuff if the coordinates of current thread is
supposed to calculate a value

    if(row < n_local && col < m_local ){
        // except for the upper-left quadrant, the other 3
need special treatment

        // if col_as_m is 0, the else part of odd-even test
 will handle the case correctly as 1 is a real number
        // and conjugate() won't change it
        const bool odd = col_as_m % 2;

        const {{float_type}} upper_right_real_multiplier =
            powf(-1, (int)odd);

        for (
            int vec_idx_in_ph = 0;
            vec_idx_in_ph < iteLen;
            vec_idx_in_ph++
        ){
            // for pixels in the the 8 octants
            if(vec_idx_in_ph < pixels_no_diagonal_ph){
                // for upper half in the upper left
                total_tmp += ar_jv_ph[ty][vec_idx_in_ph] *
                    ar_img_k_ph[vec_idx_in_ph]
                * ar_exp_ph[tx][vec_idx_in_ph];

                // upper right: conj(-1 * (real, imag))
                total_tmp += ar_jv_ph[ty][vec_idx_in_ph] *
                    ar_img_k_ph[
```

```
                2*pixels_no_diagonal_ph +
                vec_idx_in_ph
            ]
            * dcmplx(
                upper_right_real_multiplier*
                    ar_exp_ph[tx][
                        vec_idx_in_ph
                    ].real(),
                -upper_right_real_multiplier*
                    ar_exp_ph[tx][
                        vec_idx_in_ph
                    ].imag()
            );
        // lower right: -1 * (real, imag)
        total_tmp += ar_jv_ph[ty][vec_idx_in_ph] *
            ar_img_k_ph[
                6*pixels_no_diagonal_ph +
                vec_idx_in_ph
            ]
            * (upper_right_real_multiplier) *
            ar_exp_ph[tx][vec_idx_in_ph];

        // lower left: conj
        total_tmp += ar_jv_ph[ty][vec_idx_in_ph] *
            ar_img_k_ph[
                4*pixels_no_diagonal_ph +
                vec_idx_in_ph
            ]
            * conj(ar_exp_ph[tx][vec_idx_in_ph]);

        // for lower half in the upper left
```

```
dcmplx exp_lower = dcmplx(
    ar_exp_ph[tx][vec_idx_in_ph].imag(),
    ar_exp_ph[tx][vec_idx_in_ph].real()
);

if(odd){
    if(
        (
            col_as_m > 0 &&
            !((col_as_m+1) % 4 )
        )
        || ((col_as_m-1) % 4 )
    ){
        exp_lower *= -1;
    }
}else{
    if(col_as_m % 4){
        exp_lower = -conj(
            ar_exp_ph[tx][vec_idx_in_ph]
        );
    }else{
        exp_lower = conj(
            ar_exp_ph[tx][vec_idx_in_ph]
        );
    }
}

total_tmp += ar_jv_ph[ty][vec_idx_in_ph] *
    ar_img_k_ph[
        pixels_no_diagonal_ph +
        vec_idx_in_ph
```

```
                        ] * exp_lower;

                    // upper right: conj(-1 * (real, imag))
                    total_tmp += ar_jv_ph[ty][vec_idx_in_ph] *
                        ar_img_k_ph[
                            5*pixels_no_diagonal_ph +
                            vec_idx_in_ph
                        ] * dcmplx(
                            upper_right_real_multiplier *
                            exp_lower.real(),
                            -upper_right_real_multiplier *
                            exp_lower.imag()
                        );
                    // lower right: -1 * (real, imag)
                    total_tmp += ar_jv_ph[ty][vec_idx_in_ph] *
                        ar_img_k_ph[
                            7*pixels_no_diagonal_ph +
                            vec_idx_in_ph
                        ] * (upper_right_real_multiplier) *
                        exp_lower;

                    // lower left: conj
                    total_tmp += ar_jv_ph[ty][vec_idx_in_ph] *
                        ar_img_k_ph[
                            3*pixels_no_diagonal_ph +
                            vec_idx_in_ph
                        ] * conj(exp_lower);
                    count +=8;
                }else{
                    // for pixels along the diagonal axies in 4
    quarters
```

```
                    // for upper half in the upper left
                    total_tmp += ar_jv_ph[ty][vec_idx_in_ph] *
                        ar_img_k_ph[
                            7*pixels_no_diagonal_ph +
                            vec_idx_in_ph
                        ] * ar_exp_ph[tx][vec_idx_in_ph];

                    // upper right: conj(real, imag)
                    total_tmp += ar_jv_ph[ty][vec_idx_in_ph] *
                        ar_img_k_ph[
                            7*pixels_no_diagonal_ph +
                            vec_idx_in_ph +
                            pixels_diagonal_ph
                        ] * dcmplx(
                            upper_right_real_multiplier *
                            ar_exp_ph[tx][vec_idx_in_ph].real()
,

                            -upper_right_real_multiplier *
                            ar_exp_ph[tx][vec_idx_in_ph].imag()
                        );
                    // lower right: -1 * (real, imag)
                    total_tmp += ar_jv_ph[ty][vec_idx_in_ph] *
                        ar_img_k_ph[
                            7*pixels_no_diagonal_ph +
                            vec_idx_in_ph +
                            3*pixels_diagonal_ph
                        ] * (upper_right_real_multiplier) *
                        ar_exp_ph[tx][vec_idx_in_ph];

                    // lower left: conj
                    total_tmp += ar_jv_ph[ty][vec_idx_in_ph] *
```

```
                        ar_img_k_ph[
                            7*pixels_no_diagonal_ph +
                            vec_idx_in_ph +
                            2*pixels_diagonal_ph
                        ] * conj(ar_exp_ph[tx][vec_idx_in_ph]);
                    count +=4;
                }
                if(
                    count>={{group_size}} ||
                    vec_idx_in_ph == iteLen-1
                ){
                    count=0;
                    total += total_tmp;
                    total_tmp = dcmplx(0,0);
                }
            }
        }
        __syncthreads();
    }

    if(row < n_local && col < m_local )
        ar_bnm[pos] = total;
}
```

Kernel Code for BNM_KERNEL

```
# include <pycuda-complex.hpp>
typedef pycuda::complex<{{float_type}}> dcmplx;
__global__ void reconstruction_kernel(
    {{float_type}} *ar_jv,
    dcmplx *ar_bnm,
    dcmplx *ar_exp,
    long long *n,
    long long *m,
    long long *pixels_in_part
){
    const long long bx = blockIdx.x;
    const long long by = blockIdx.y;
    const long long bw = blockDim.x;
    const long long tx = threadIdx.x;
    const long long ty = threadIdx.y;
    const long long n_local = *n;
    const long long m_local = *m;
    const long long pixels_in_part_local = *pixels_in_part;
    const long long ph_len = {{ph_len}};

    // identify the row and col num for the current thread
    const long long row = by * bw + ty;
    const long long col = bx * bw + tx;
    const long long pos = (long long)row *
        (long long)pixels_in_part_local + (long long)col;
    // const long long block_pos = tx + bw*ty;

    dcmplx total = dcmplx(0,0);

    // allocate shared memory
```

```
__shared__ {{float_type}} ar_jv_ph[{{bw}}][ph_len];
__shared__ dcmplx ar_bnm_ph[{{bw}}][ph_len];

// loop over tiles in ar_jv and ar_bnm to calculate

for (int ph = 0; ph <= (n_local-1)/ph_len; ph++){

// load data into shared momery && boundary checking
const long long iteLen = (ph+1)*ph_len > n_local
    ? n_local-ph*ph_len : ph_len;

long long base_load_count = iteLen / bw +
    (int)(tx < iteLen % bw);

if(row < m_local){
    for(int i=0; i<base_load_count; i++){
        ar_bnm_ph[ty][i*bw + tx] = ar_bnm[
            (ph*ph_len + i*bw + tx)*m_local + row
        ];
    }
}

base_load_count = iteLen / bw + (int)(ty < iteLen % bw);
if(col < pixels_in_part_local){
    for(int i=0; i<base_load_count; i++){
        ar_jv_ph[tx][i*bw + ty] = ar_jv[
            (ph*ph_len + i*bw + ty)*pixels_in_part_local
            + col
        ];
    }
}
```

```
    __syncthreads();

    if(row < m_local && col < pixels_in_part_local ){
        for (
            int vec_idx_in_ph = 0;
            vec_idx_in_ph < iteLen;
            vec_idx_in_ph++
        ){
            total += ar_bnm_ph[ty][vec_idx_in_ph] *
                ar_jv_ph[tx][vec_idx_in_ph];
        }
    }

    __syncthreads();

    }

    if(row >= m_local || col >= pixels_in_part_local )
        return;

    ar_exp[pos] = total * (({{float_type}})1)/ar_exp[pos];
}
```

Kernel Code for RECONSTRUCTION_KERNEL

Bibliography

[1] Milton Abramowitz and Irene A Stegun. *Handbook of mathematical functions: with formulas, graphs, and mathematical tables*. Vol. 55. Courier Corporation, 1964.

[2] DE Amos. “A portable package for Bessel functions of a complex argument and nonnegative order, 1983”. In: *Trans. Math. Software* ().

[3] Amy Chiang and Simon Liao. “Image Analysis with Legendre Moment Descriptors”. In: Journal of Computer Science 11.1 (Jan. 2015), pp. 127–136. ISSN: 1549-3636. DOI: 10.3844/jcssp.2015.127.136.

[4] C W Chong, P. Raveendran, and R. Mukundan. “A comparative analysis of algorithms for fast computation of Zernike moments”. In: Pattern Recognition 36.3 (Mar. 2003), pp. 731–742. ISSN: 0031-3203. DOI: 10.1016/S0031-3203(02)00091-2.

[5] *CUDA C++ Programming Guide*. Aug. 2019. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#data-transfer-between-host-and-device.

[6] Mostafa El Mallahi et al. “Radial Hahn Moment Invariants for 2D and 3D Image Recognition”. In: International Journal of Automation and Computing 15.3 (June 2018), pp. 277–289. ISSN: 1476-8186. DOI: 10.1007/s11633-017-1071-1.

[7] Jan Flusser, Tomáš Suk, and Barbara Zitová. *Moments and moment invariants in pattern recognition*. Wiley, 2009. ISBN: 0470699876.

[8] Rafael C Gonzalez and Richard E Woods. *Digital image processing*. Prentice Hall, 2008. ISBN: 013505267X.

[9] Ming-Kuei Hu. *Moments and Moment Invariants - Theory and Applications*. Vol. 8. IEEE, Feb. 1962, pp. 179–187.

[10] Sun-Kyoo Hwang and Whoi-Yul Kim. “A novel approach to the fast computation of Zernike moments”. In: Pattern Recognition 39.11 (2006), pp. 2065–2076. ISSN: 0031-3203. DOI: //doi.org/10.1016/j.patcog.2006.03.004. URL: http://www.sciencedirect.com/science/article/pii/S0031320306001166.

[11] *Image analysis by circularly semi-orthogonal moments.* 2016. DOI: //doi.org/10.1016/j.patcog.2015.08.005. URL: http://www.sciencedirect.com/science/article/pii/S0031320315002939.

[12] *Jinja Document 2.11.x.* URL: https://jinja.palletsprojects.com/en/2.11.x/.

[13] David Kirk and Wen-mei Hwu. *Programming Massively Parallel Processors, 3rd Edition.* 3rd ed. Morgan Kaufmann, Nov. 2016. ISBN: 9780128119860. URL: https://proquestcombo.safaribooksonline.com/9780128119877.

[14] Simon Liao. “Accuracy Analysis of Moment Functions”. In: ed. by George A. Papakostas. Moments and Moment Invariants - Theory and Applications. Science Gate Publishing, 2014. Chap. 2, pp. 33–56.

[15] Vijayalakshmi G. V Mahesh. “Invariant moments based convolutional neural networks for image analysis”. In: International Journal of Computational Intelligence Systems (Jan. 2017). ISSN: 1875-6883. DOI: 10.2991/ijcis.2017.10.1.62.

[16] Vijayalakshmi G. V. Mahesh and Alex Noel Joseph Raj. “Zernike Moments and Machine Learning Based Gender Classification Using Facial Images”. In: ed. by Ajith Abraham et al. Springer International Publishing, 2018, pp. 398–408.

[17] Manuel Jesús Martín Requena, Pablo Moscato, and Manuel Ujaldón. “Efficient data partitioning for the GPU computation of moment functions”. In: Journal of Parallel and Distributed Computing 74.1 (Jan. 2014), pp. 1994–2004. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2013.

07.008. URL: https://www.sciencedirect.com/science/article/pii/S0743731513001342.

[18] R. Mukundan, S. H. Ong, and P. A. Lee. *Image analysis by Tchebichef moments*. 2001. DOI: 10.1109/83.941859.

[19] Marcel Nwali and Simon Liao. "A new fast algorithm to compute continuous moments defined in a rectangular region". In: Pattern Recognition 89 (May 2019), pp. 151–160. ISSN: 0031-3203. DOI: 10.1016/j.patcog.2019.01.001. URL: https://www.sciencedirect.com/science/article/pii/S0031320319300019.

[20] Ziliang Ping et al. "Generic orthogonal moments: Jacobi–Fourier moments for invariant image description". In: Pattern Recognition 40.4 (2007), p. 1245. ISSN: 0031-3203. DOI: //doi.org/10.1016/j.patcog.2006.07.016". URL: http://www.sciencedirect.com/science/article/pii/S0031320306003475.

[21] "Properties of Orthogonal Gaussian-Hermite Moments and Their Applications". In: EURASIP Journal on Advances in Signal Processing 2005 (2005), p. 439420. ISSN: 1687-6180. DOI: 10.1155/ASP.2005.588. URL: https://doi.org/10.1155/ASP.2005.588.

[22] Yunlong Sheng and Lixin Shen. "Orthogonal Fourier–Mellin moments for invariant pattern recognition". In: Journal of the Optical Society of America A 11.6 (June 1994), p. 1748. ISSN: 1084-7529. DOI: 10.1364/JOSAA.11.001748.

[23] Chandan Singh and Ekta Walia. "Algorithms for fast computation of Zernike moments and their numerical stability". In: Image and Vision Computing 29.4 (2011), pp. 251–259. ISSN: 0262-8856. DOI: 10.1016/j.imavis.2010.10.003. URL: https://www.sciencedirect.com/science/article/pii/S0262885610001423.

[24] M R TEAGUE. "Image-Analysis Via the General-Theory of Moments". In: Journal of the Optical Society of America 70.8 (1980), pp. 920–930. ISSN: 0030-3941. DOI: 10.1364/JOSA.70.000920.

[25] C. -. Teh and R. T. Chin. *On image analysis by the methods of moments*. 1988. DOI: 10.1109/34.3913.

[26] Rahul Upneja. "Accurate and fast Jacobi-Fourier moments for invariant image recognition". In: Optik - International Journal for Light and Electron Optics 127.19 (Oct. 2016), pp. 7925–7940. ISSN: 0030-4026. DOI: 10.1016/j.ijleo.2016.05.097. URL: https://www.sciencedirect.com/science/article/pii/S0030402616305472.

[27] Tiansheng Wang and Simon Liao. "Computational aspects of exponent-Fourier moments". In: Pattern Recognition Letters 84 (2016), pp. 35–42. ISSN: 0167-8655. DOI: //doi.org/10.1016/j.patrec.2016.08.004. URL: http://www.sciencedirect.com/science/article/pii/S0167865516301982.

[28] Xiaoyu Wang and Simon Liao. "Image Reconstruction from Orthogonal Fourier-Mellin Moments". In: Image Analysis and Recognition 7950 (2013), pp. 687–694. ISSN: 0302-9743.

[29] Bin Xiao, Jian-Feng Ma, and Xuan Wang. "Image analysis by Bessel-Fourier moments". In: Pattern Recognition 43.8 (Aug. 2010), pp. 2620–2629. ISSN: 0031-3203. DOI: 10.1016/j.patcog.2010.03.013.

[30] Yongqing Xin, Simon Liao, and Miroslaw Pawlak. "Circularly orthogonal moments for geometrically robust image watermarking". In: Pattern Recognition 40.12 (Dec. 2007), pp. 3740–3752. ISSN: 0031-3203. DOI: 10.1016/j.patcog.2007.05.004.

[31] Yubo Xuan, Dayu Li, and Wei Han. "Efficient optimization approach for fast GPU computation of Zernike moments". In: Journal of Parallel and Distributed Computing 111 (2018), pp. 104–114. ISSN: 0743-7315. DOI: //doi.org/10.1016/j.jpdc.2017.07.008. URL: http://www.sciencedirect.com/science/article/pii/S0743731517302204.

[32] Bo Yang, Jan Flusser, and Tomáš Suk. "3D rotation invariants of Gaussian–Hermite moments". In: Pattern Recognition Letters 54 (Mar. 2015), pp. 18–26. ISSN: 0167-8655. DOI: 10.1016/j.patrec.2014.11.014. URL: http://dx.doi.org/10.1016/j.patrec.2014.11.014.

[33] Bo Yang et al. "Rotation invariants of vector fields from orthogonal moments". In: Pattern Recognition 74.C (Feb. 2018), pp. 110–121. ISSN: 0031-3203. DOI: 10.1016/j.patcog.2017.09.004. URL: https://www.sciencedirect.com/science/article/pii/S0031320317303540.