Accelerating Computation of Zernike and Pseudo-Zernike Moments with a GPU Algorithm

by

Shiying Li

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
The University of Winnipeg
Winnipeg, Manitoba, Canada
December 2020

Abstract

Although Zernike and pseudo-Zernike moments have some advanced properties, the computation process is generally very time-consuming, which has limited their practical applications. To improve the computational efficiency of Zernike and pseudo-Zernike moments, in this research, we have explored the use of GPU to accelerate moments computation, and proposed a GPU-accelerated algorithm. The newly developed algorithm is implemented in Python and CUDA C++ with optimizations based on symmetric properties and $k \times k$ sub-region scheme. The experimental results are encouraging and have shown that our GPU-accelerated algorithm is able to compute Zernike moments up to order 700 for an image sized at 512×512 in 1.7 seconds and compute pseudo-Zernike moments in 3.1 seconds. We have also verified the accuracy of our GPU algorithm by performing image reconstructions from the higher orders of Zernike and pseudo-Zernike moments. For an image sized at 512×512, with the maximum order of 700 and $k = 11$, the PSNR (Peak Signal to Noise Ratio) values of its reconstructed versions from Zernike and pseudo-Zernike moments are 44.52 and 46.29 separately. We have performed image reconstructions from partial sets of Zernike and pseudo-Zernike moments with various order n and different repetition m. Experimental results of both Zernike and pseudo-Zernike moments show that the images reconstructed from the moments of lower and higher orders preserve the principle contents and details of the original image respectively, while moments of positive and negative m result in identical images. Lastly, we have proposed a set of feature vectors based on pseudo-Zernike moments for Chinese character recognition. Three different feature vectors are composed of different parts of four selected lower pseudo-Zernike moments. Experiments on a set of 6,762 Chinese characters show that this method performs well to recognize similar-shaped Chinese characters.

Acknowledgements

I would like to express my deepest appreciation to Dr. Simon Liao. Your academic knowledge and experience have provided valuable guidance and suggestions during my research. During the process of writing this thesis, I was strongly influenced by your rigorous and diligent attitude to studying. Thank you for your patience and encouragement when I met difficulties and had doubts about myself. I cannot be more grateful for having you as my supervisor during my graduate study in the University of Winnipeg.

I am extremely grateful to my thesis committee members, Dr. Christopher Henry and Christopher Bidinosti, for their very valuable suggestions which have helped me to improve the thesis significantly.

I would also like to extend my thanks to those who have also studied under Dr. Liao. Discussions with them have provided me with a great deal of help in solving problems during research and programming.

Thanks also to the faculty of the Department of Applied Computer Science and the Graduate Studies for their assistance in students' daily campus life.

I could not have completed this thesis without the support of my family. I would particularly like to thank my husband, Jie Liang, who has taken most of the responsibility of raising our baby and ensured me more time on the research. Finally, thanks to my son, whose smile gives me the courage and power to overcome any obstacles.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Moments of digital images were first introduced in 1962 by Hu [1]. Since then, various moments with different kernel functions have been devised by researchers. Among them, orthogonal moments have several advanced properties, such as representing images with minimum redundancy information and being invariant to rotations and reflections [2]. Zernike moments and pseudo-Zernike moments are orthogonal moments defined on a circular domain. They are widely applied in many scientific areas, such as image analysis, pattern recognition, and watermarking [3].

Although Zernike and pseudo-Zernike moments have some more advanced properties, their practical applications, such as medical image watermarking [4] and image retrieval [5], are limited by the higher computational demands, especially when the applications are real-time or involve a large number of images. The Zernike and pseudo-Zernike kernel functions include both radial and exponential polynomials, and the former involves a series of factorials, which makes it a time-consuming process to calculate. The computational accuracy is also an important issue for developing applications based on Zernike or pseudo-Zernike moments, while some researchers have made a good effort to find solutions in the past a few years [2, 6–10].

Along with optimizing the CPU-based algorithms to improve moments computation, researchers have applied GPUs to accelerate the computing processes of Zernike moments in the past a few years. The research results in Refs. [11–14] show some notable successes in improving the computation efficiency of Zernike moments with orders lower than 40. However, there is a lack of research on the GPU accelerated computation for Zernike and pseudo-Zernike moments with higher orders. In this research, we have developed an

algorithm to accelerate Zernike and pseudo-Zernike moments computation with a CUDA-enabled GPU. To verify the performance of our GPU algorithm, we have conducted image reconstructions from Zernike and pseudo-Zernike moments with various maximum orders up to 700. Our experimental results have shown that the newly developed GPU algorithm could provide the efficient moments computation. To investigate the properties of Zernike and pseudo-Zernike moments, we have also performed image reconstructions from different sets of partial moments with diverse order n and different repetition m.

Chinese character recognition systems based on local structure features face difficulties to recognize characters with similar structures. Since moment-based features could capture the global statistic properties of an image rather than local structure features, they are introduced into Chinese character recognition systems. We have proposed a method based on pseudo-Zernike moments for Chinese character recognition. Experiments on a set of 6,762 Chinese characters show that this method performs well to recognize similar-shaped Chinese characters.

The rest of the thesis is organized as follows. Chapter 2 demonstrates the definitions and properties of image moments, Zernike moments and pseudo-Zernike moments. In Chapter 3, three main computation methods of the moments and two optimization strategies are presented. Chapter 4 introduces the structures and optimizations of our implemented GPU algorithm. Our experimental results are shown in Chapter 5. Chapter 6 gives details of applying pseudo-Zernike moments in Chinese characters recognition. Finally, we will give our concluding remarks in Chapter 7.

Chapter 2

Zernike and Pseudo-Zernike Moments

2.1 Image Moments

Image moments are real or complex-valued scalar quantities that describe the pixels distribution of an image [3, 15]. Mathematically, moments can be seen as projections of a image function onto a polynomial basis. The general definition of the $(p+q)$-th order of moments with the image intensity function $f(x, y)$ and a moment weighting kernel $\psi_{pq}(x, y)$ is given as [16]

$$\Psi_{pq} = \int_x \int_y \psi_{pq}(x, y) f(x, y) dxdy, \quad p, q = 0, 1, 2, \ldots. \tag{2.1}$$

The moments are orthogonal if their kernel satisfy the condition of orthogonality

$$\int\int_{\Omega} \psi_{pq}(x, y)\psi_{st}(x, y)dxdy = w\delta_{ps}\delta_{qt} \tag{2.2}$$

where w is the normalization coefficient, δ_{ps} is the Kronecker delta function and Ω is the area of orthogonality [17].

Zernike moments and pseudo-Zernike moments are both orthogonal moments defined over a circular domain. To calculate those moments of an image in a Cartesian coordinate system, the image plane needs to be mapped over a unit disk, with the center of the image taken as the origin point of the disk, as shown in Fig. 2.1 [18]. Pixels who are entirely located in the circle will be used in the computation; otherwise, they will be discarded.

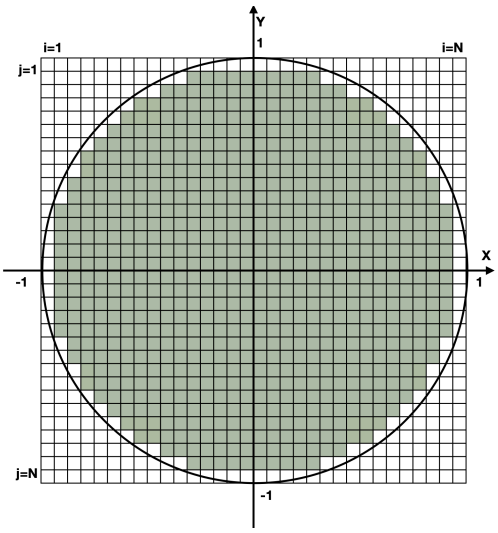


Figure 2.1: Mapping image plane to a unit circle. The center of the image is taken as the origin point of the disk. Pixels with any part out of the circle are discarded.

With such a mapping method, for an image with $N \times N$ pixels, the Cartesian coordinate of the pixel located in the i-th column and j-th row, notes as (x_i, y_j), is calculated by

$$x_i = \frac{2 \times i - N - 1}{N} \tag{2.3}$$

and

$$y_j = \frac{N - 2 \times j + 1}{N}. \tag{2.4}$$

2.2 Zernike Moments

Zernike moments are orthogonal moments defined over a circular domain. The Zernike function with order n and repetition m is defined on the unit disk as [19]

$$V_{nm}(x, y) = R_{nm}(\rho)e^{jm\theta}, \quad x^2 + y^2 \leq 1, \tag{2.5}$$

where $\rho = \sqrt{x^2 + y^2}$ is the length of the vector from the origin to the pixel (x, y), $\theta = \tan^{-1}(y/x)$ is the angle between the vector and the x axis, and $j = \sqrt{-1}$. V_{nm} is a complex number when $m \neq 0$. The real valued radial Zernike polynomial $R_{nm}(\rho)$ is defined as [20]

$$R_{nm}(\rho) = \sum_{s=0}^{(n-|m|)/2} (-1)^s \frac{(n-s)!}{s!(\frac{n+|m|}{2} - s)!(\frac{n-|m|}{2} - s)!} \rho^{n-2s}, \tag{2.6}$$

where $n - |m|$ is an even number, and m is an integer that can take positive, negative, or zero values while $|m| \leq n$.

The Zernike moments of order n with repetition m, A_{nm}, is defined as [21]

$$A_{nm} = \frac{n+1}{\pi} \iint_{x^2+y^2 \leq 1} f(x, y) V_{nm}^*(x, y) dx dy, \tag{2.7}$$

where * denotes complex conjugate.

The Zernike moments are rotational invariant. If an image $f(x, y)$ is rotated α degrees counter-clockwise, the Zernike moments of the rotated image are

$$A_{nm}^{(\alpha)} = A_{nm} e^{-jm\alpha}. \tag{2.8}$$

This leads to

$$|A_{nm}^{(\alpha)}| = |A_{nm}|, \tag{2.9}$$

which means that the magnitudes of the Zernike moments are rotational invariant [21].

To compute the Zernike moments A_{nm} in a Cartesian coordinate system, a commonly used format is

$$\hat{A}_{nm} = \frac{n+1}{\pi} \sum_{{x_i}^2+{y_j}^2 \leq 1} \sum f(x_i, y_j) V_{nm}^*(x_i, y_j) \Delta x \Delta y, \tag{2.10}$$

where Δx and Δy are the sampling intervals in the x and y directions [21], which are $\frac{2}{N}$.

Due to the orthogonality of the Zernike functions, we can reconstruct the image function $f(x, y)$ by its Zernike moments

$$f(x, y) = \sum_{n=0}^{\infty} \sum_{m=-n}^{n} A_{nm} V_{nm}(x, y). \tag{2.11}$$

To reconstruct the image function $f(x, y)$ with a finite set of its Zernike moments in a Cartesian plane, Eq. (2.11) can be approximated by

$$\hat{f}(x_i, y_j) = \sum_{n=0}^{T} \sum_{m=-n}^{n} \hat{A}_{nm} V_{nm}(x_i, y_j), \tag{2.12}$$

where T is the maximum order of Zernike moments taken into account in the image reconstruction.

2.3 Pseudo-Zernike Moments

Pseudo-Zernike moments are the modified version of Zernike moments, and were developed by Bhatia and Wolf [22]. The pseudo-Zernike polynomial of order n with repetition m is defined on the unit disk as [21]

$$V_{nm}(x, y) = R_{nm}(\rho) e^{jm\theta}, \quad x^2 + y^2 \leq 1, \tag{2.13}$$

where $R_{nm}(\rho)$ is defined as

$$R_{nm}(\rho) = \sum_{s=0}^{n-|m|} (-1)^s \frac{(2n + 1 - s)!}{s!(n + |m| + 1 - s)!(n - |m| - s)!} \rho^{n-s}, \tag{2.14}$$

where $n = 0, 1, 2, \ldots, \infty$, and m is an integer that can take positive, negative, or zero values while $|m| \leq n$.

Based on Eqs. (2.6) and (2.14), the number of all the Zernike moments whose order $\leq n$ is $\frac{1}{2}(n+1)(n+2)$, while this number for the pseudo-Zernike moments is $(n + 1)^2$. For example, as shown in Fig. 2.2, when $n = 0$, there

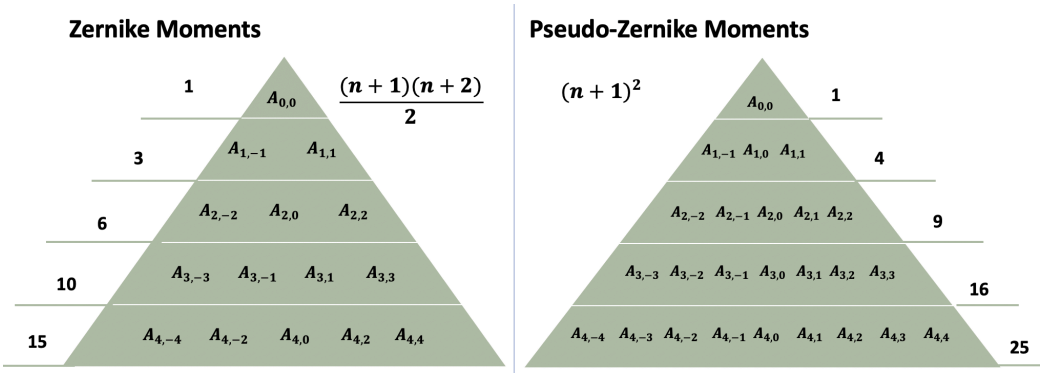


Figure 2.2: The number of Zernike and pseudo-Zernike moments with different order n. There are $\frac{1}{2}(n+1)(n+2)$ Zernike moments and $(n+1)^2$ pseudo-Zernike moments whose order is $\leq n$.

are only one Zernike moment and one pseudo-Zernike moment. When $n = 4$, the number of Zernike moments is 15 and that of pseudo-Zernike moments is 25.

The pseudo-Zernike moments of order n with repetition m, A_{nm}, is defined as [21]

$$A_{nm} = \frac{n+1}{\pi} \iint\limits_{x^2+y^2 \leq 1} f(x,y) V_{nm}^*(x,y) dxdy, \tag{2.15}$$

where * denotes complex conjugate.

To compute the pseudo-Zernike moments A_{nm} in a Cartesian coordinate system, a commonly used format is

$$\hat{A}_{nm} = \frac{n+1}{\pi} \sum\sum_{{x_i}^2+{y_j}^2 \leq 1} f(x_i, y_j) V_{nm}^*(x_i, y_j) \Delta x \Delta y, \tag{2.16}$$

where Δx and Δy are the sampling intervals in the x and y directions.

Due to the orthogonality of the pseudo-Zernike functions, we can reconstruct the image function $f(x,y)$ by its pseudo-Zernike moments

$$f(x,y) = \sum_{n=0}^{\infty} \sum_{m=-n}^{n} A_{nm} V_{nm}(x,y). \tag{2.17}$$

To reconstruct the image function $f(x, y)$ with a finite set of its pseudo-Zernike moments in a Cartesian plane, Eq. (2.17) can be approximated by

$$\hat{f}(x_i, y_j) = \sum_{n=0}^{T} \sum_{m=-n}^{n} \hat{A}_{nm} V_{nm}(x_i, y_j), \tag{2.18}$$

where T is the maximum order of pseudo-Zernike moments taken into account in the image reconstruction.

Same as for the Zernike moments, the magnitudes of the pseudo-Zernike moments are also rotational invariant.

2.4 Summary

In this chapter, we have given the definitions and properties of image moments, Zernike moments and pseudo-Zernike moments. The Zernike and pseudo-Zerniek moments are different in the domain of their repetition m and the definition of the radial polynomials $R_{nm}(\rho)$. The number of all the Zernike moments whose order is less than or equal to n is $\frac{1}{2}(n+1)(n+2)$, while that of the pseudo-Zernike moments is $(n+1)^2$.

Chapter 3

Zernike and Pseudo-Zernike Moments Computation

3.1 Computation Methods

Computational efficiency and numerical accuracy are two major issues involved in the research of Zernike moments and pseudo-Zernike moments [3, 21]. Different methods have been proposed to calculate the two kinds of moments more efficiently and accurately [2, 7–10, 20, 23–26].

Direct Methods

The direct methods use the definition formulas to calculate moments directly. Equations (2.5), (2.6) and (2.10) are used to calculate Zernike moments, and Eqs. (2.13), (2.14) and (2.16) are used to compute pseudo-Zernike moments. Due to the radial polynomial and exponential component in the formulas, the calculation is time-consuming, especially when the size of the image is larger and the moment order is higher.

Geometric Moments Methods

The relationship between Zernike moments and geometric moments are given in Ref. [20] as

$$\hat{A}_{nm} = \frac{n+1}{\pi} \sum_{\substack{k=|m| \\ n-k=even}}^{n} \sum_{p=0}^{S} \sum_{q=0}^{|m|} (w)^q C_p^S C_q^{|m|} B_{n,|m|,k} G_{k-2p-q,2p+q}, \tag{3.1}$$

where $S = (k - |m|)/2$, and

$$w = \begin{cases} -j, & m > 0 \\ j, & m \leq 0, \end{cases} \tag{3.2}$$

$$B_{n,m,k} = \frac{(-1)^{\frac{n-k}{2}}(\frac{n+k}{2})!}{(\frac{n-k}{2})!\frac{k+|m|}{2})!\frac{k-|m|}{2})!}. \tag{3.3}$$

Hosny [2] expressed pseudo-Zernike moments in terms of geometric and radial geometric moments as follows:

$$\hat{A}_{nm} = \frac{n+1}{\pi}[\{A_1\}_{k-m=even} + \{A_2\}_{k-m=odd}], \tag{3.4}$$

with

$$A_1 = \sum_{k=m}^{n} \sum_{p=0}^{S_1} \sum_{q=0}^{m} (-j)^q C_p^{s_1} C_q^m B_{n,m,k} G_{k-2p-q,2p+q}, \tag{3.5}$$

$$A_2 = \sum_{k=m+1}^{n} \sum_{p=0}^{S_2} \sum_{q=0}^{m} (-j)^q C_p^{s_2} C_q^m B_{n,m,k} H_{k-2p-q-1,2p+q}, \tag{3.6}$$

where $S_1 = (k-m)/2$, $S_2 = (k-m-1)/2$, and

$$B_{n,m,k} = \frac{(-1)^{n-k}(n+k+1)!}{(n-k)!(m+k+1)!(k-m)!}. \tag{3.7}$$

The geometric moments $G_{n,m}$ and radial geometric moments $H_{n,m}$ are defined as

$$G_{n,m} = \int_x \int_y x^n y^m f(x,y) dxdy, \tag{3.8}$$

$$H_{n,m} = \int_x \int_y x^n y^m (x^2+y^2)^{\frac{1}{2}} f(x,y) dxdy. \tag{3.9}$$

This method transforms the computation of Zernike and pseudo-Zernike moments into that of the geometric moments, which can be computed more accurately. As a result, it removes the numerical errors produced in the direct method and outperforms the direct method in computational efficiency [2].

Recursive Methods

The recursive methods set some initial conditions and derive moments of different orders and repetitions recursively. They are more efficient and more stable than the direct method because they do not involve any factorial terms.

Inspired by Prata's [23] and Kintner's [24] recursive methods, Chong et al. proposed the q-recursive method to compute Zernike moments [7] and pseudo-Zernike moments [8]. Based on Chong's work, several other recursive methods were proposed [25, 26].

More recently, Deng et al. proposed a recursive algorithm that outperforms other algorithms of computing Zernike moments both in efficiency and accuracy [9]. The recurrence relations used to compute the radial polynomials $R_{n,|m|}(\rho)$ in this method are

$$R_{nn}(\rho) = \rho^n, \tag{3.10}$$

where $0 \leq n \leq T$, and

$$R_{n|m|}(\rho) = r(R_{n-1,|m-1|}(\rho) + R_{n-1,|m+1|}(\rho)) - R_{n-2,|m|}(\rho), \tag{3.11}$$

where $n \geq 2$.

Deng and Gwo also proposed a recursive algorithm to compute pseudo-Zernike moments in 2018 [10]. In their work, the ultimate recursive relations among the radial polynomials $R_{nm}(\rho)$ and the intermediate values noted as $K_{nm}(\rho)$ are

$$R_{nn}(\rho) = K_{nn}(\rho) = \rho^n, \tag{3.12}$$

where $n = 0, 1, 2, \ldots, T$.

$$K_{nm}(\rho) = \rho(R_{n-1,m-1}(\rho) + R_{n-1,m}(\rho)) - K_{n-1,m}(\rho) \tag{3.13}$$

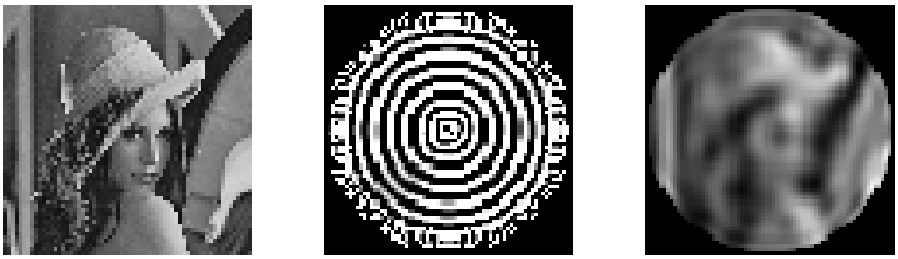
and

$$K_{n0}(\rho) = 2\rho R_{n-1,0}(\rho) - K_{n-1,0}(\rho), \tag{3.14}$$

where $n = 1, 2, \ldots, T$ and $m = n-1, n-2, \ldots, 1$.

$$R_{nm}(\rho) = K_{nm}(\rho) + K_{n,m+1}(\rho) - R_{n-1,m}(\rho), \tag{3.15}$$

where $n = 1, 2, \ldots, T$ and $m = n-1, n-2, \ldots, 1, 0$.



(a) (b) (c)

Figure 3.1: The images reconstructed from the Zernike moments up to order 23 of an image sized at 64×64. (a) The original image. (b) The image reconstructed from Zernike moments computed through the geometric moments method. It is totally broken because the moments are too inaccurate. (c) The image reconstructed from Zernike moments computed through Deng's recursive moments method. It is much better than the image in (b).

We have done some preliminary experiments to compare the performances of those computation methods. The moments computed through the direct methods and the geometric moments methods are inaccurate even when the order is not too high. For example, when the Zernike moments of an image sized at 64×64 are computed with the geometric moments methods and the maximum number T is 23, the image reconstructed from them is totally broken, as shown in Fig. 3.1(b). With the q-recursive methods and Deng's recursive methods, the moments are more accurate and the reconstructed images are like the image shown in Fig. 3.1(c). Deng's recursive methods are recently proposed and perform better than the q-recursive method [9, 10]. Therefore we have chosen them and applied the two optimization methods shown in Section 3.2 in our GPU implementation.

3.2 Optimization Strategies

In addition to the above moments calculation methods, there are also some strategies that can be combined with these methods to improve their performance.

Reducing Computation by Symmetric Property

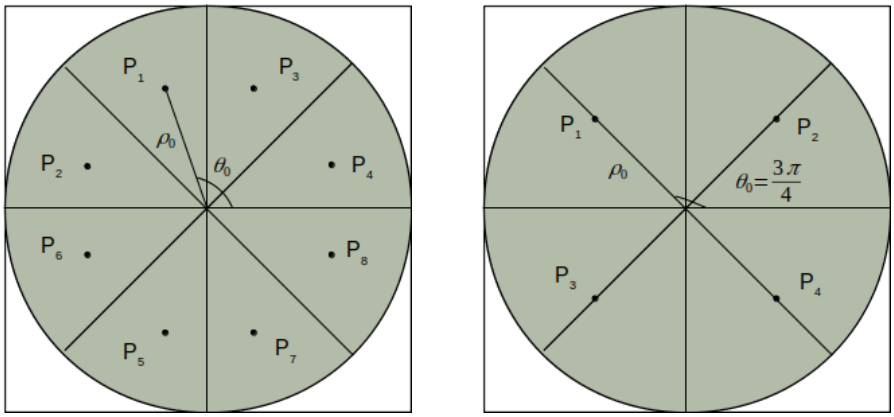


Figure 3.2: Symmetric pixels. (a) Pixels P_1 to P_8 are 'eight-symmetric'. The polar coordinates of P_1 is (ρ_0, θ_0). P_2 to P_8 have the same radial coordinate ρ_0 with P_1, and their angular coordinates can be expressed with θ_0. (b) Pixels P_1 to P_4 are 'four-symmetric'.

The symmetric property of moments defined on a circular domain was widely used by researchers to reduce moments computation time [9, 14, 27]. As shown in Fig. 3.2 (a), the unit disk is separated into eight parts by the x-axis, y-axis, and lines of $y = x$ and $y = -x$. For a pixel named P_1 with polar coordinates (ρ_0, θ_0), where $0 < \rho_0 \leq 1$ and $\frac{\pi}{2} \leq \theta_0 < \frac{3\pi}{4}$, there are seven corresponding pixels in other seven parts. Those pixels have the same radial coordinate ρ_0 with P_1, and their angular coordinates can be expressed

with θ_0, as shown in Table 3.1. We use 'eight-symmetric' to describe the relationships among these pixels.

Due to the property of $e^{jm\theta}$ shown in Table 3.1, we can obtain the Zernike or pseudo-Zernike polynomials $V_{nm}(x, y)$ of P_2 to P_8 through that of P_1. Therefore, the redundant computation of $R_{nm}(\rho)$ and $e^{jm\theta}$ is eliminated, and the computing time would be reduced significantly.

Table 3.1: Values of $e^{jm\theta}$ for eight-symmetric pixels with different m.

Pixel	Angle	$e^{jm\theta}$ m $mod\ 4 = 0$	$e^{jm\theta}$ m $mod\ 4 = 1$	$e^{jm\theta}$ m $mod\ 4 = 2$	$e^{jm\theta}$ m $mod\ 4 = 3$
P_1	θ_0	$e^{jm\theta_0}$	$e^{jm\theta_0}$	$e^{jm\theta_0}$	$e^{jm\theta_0}$
P_2	$\frac{3\pi}{2} - \theta_0$	$e^{-jm\theta_0}$	$je^{-jm\theta_0}$	$-e^{-jm\theta_0}$	$-je^{-jm\theta_0}$
P_3	$\pi - \theta_0$	$e^{-jm\theta_0}$	$-e^{-jm\theta_0}$	$e^{-jm\theta_0}$	$-e^{-jm\theta_0}$
P_4	$\theta_0 - \frac{\pi}{2}$	$e^{jm\theta_0}$	$je^{jm\theta_0}$	$-e^{jm\theta_0}$	$-je^{jm\theta_0}$
P_5	$-\theta_0$	$e^{-jm\theta_0}$	$e^{-jm\theta_0}$	$e^{-jm\theta_0}$	$e^{-jm\theta_0}$
P_6	$\theta_0 - \frac{3\pi}{2}$	$e^{jm\theta_0}$	$-je^{jm\theta_0}$	$-e^{jm\theta_0}$	$je^{jm\theta_0}$
P_7	$\theta_0 - \pi$	$e^{jm\theta_0}$	$-e^{jm\theta_0}$	$e^{jm\theta_0}$	$-e^{jm\theta_0}$
P_8	$\frac{\pi}{2} - \theta_0$	$e^{-jm\theta_0}$	$-je^{-jm\theta_0}$	$-e^{-jm\theta_0}$	$je^{-jm\theta_0}$

Table 3.2: Values of $e^{jm\theta}$ for four-symmetric pixels with different m.

Pixel	Angle	$e^{jm\theta}$ m $mod\ 4 = 0$	$e^{jm\theta}$ m $mod\ 4 = 1$	$e^{jm\theta}$ m $mod\ 4 = 2$	$e^{jm\theta}$ m $mod\ 4 = 3$
P_1	$\frac{3\pi}{4}$	-1	$e^{jm\theta_0}$	$e^{jm\theta_0}$	$e^{jm\theta_0}$
P_2	$\frac{\pi}{4}$	-1	$je^{jm\theta_0}$	$-e^{jm\theta_0}$	$-je^{jm\theta_0}$
P_3	$\frac{-3\pi}{4}$	-1	$e^{-jm\theta_0}$	$e^{-jm\theta_0}$	$e^{-jm\theta_0}$
P_4	$\frac{-\pi}{4}$	-1	$je^{-jm\theta_0}$	$-e^{-jm\theta_0}$	$-je^{-jm\theta_0}$

Similarly, Fig. 3.2 (b) shows that for a pixel P_1 on the line $y = -x$ where $x < 0$, there are three symmetric pixels. We use 'four-symmetric' to describe such pixels. Their relationships are displayed in Table 3.2.

Let the gray values of pixels P_1 to P_8 be g_1 to g_8, then the Zernike (or pseudo-Zernike) moments with order n and repetition m can be obtained by

$$A_{nm} = \frac{n+1}{\pi}(E_{nm} + F_{nm})\Delta x \Delta y, \tag{3.16}$$

where

$$E_{nm} = \sum_{\substack{y>-x \\ x^2+y^2\leq 1}} \sum_{-1<x<0} R_{nm}(\rho) \sum_{i=1}^{8} g_i e^{-jm\theta_i} \tag{3.17}$$

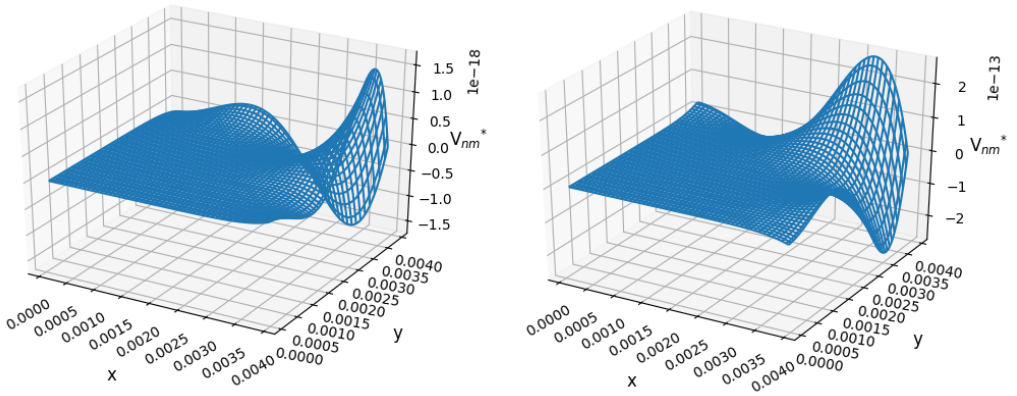
with $e^{jm\theta_i}$ as the value of $e^{jm\theta}$ for P_i in Table 3.1, and

$$F_{nm} = \sum_{\substack{y=-x \\ x^2+y^2\leq 1}} \sum_{-1<x<0} R_{nm}(\rho) \sum_{i=1}^{4} g_i e^{-jm\theta_i} \tag{3.18}$$

with $e^{jm\theta_i}$ as the value of $e^{jm\theta}$ for P_i in Table 3.2.

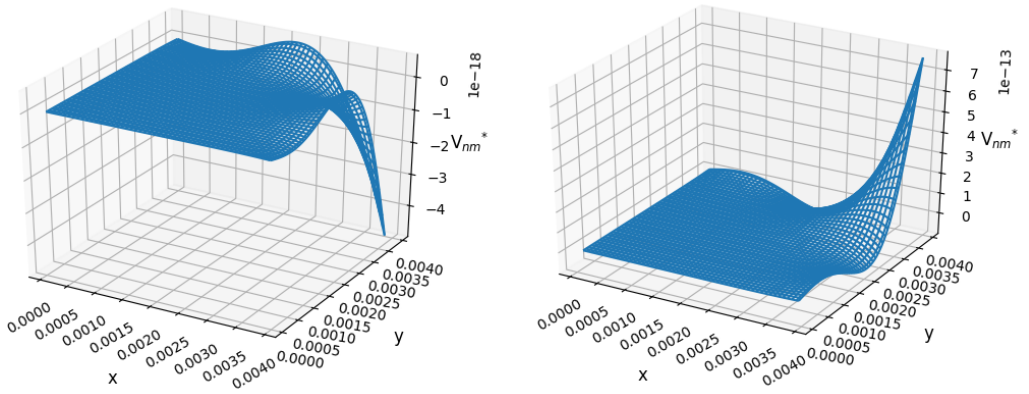
Improving Accuracy by Sub-region Scheme

In Eqs. (2.10) and (2.16), we use the area of a pixel, $\Delta x \Delta y$, to estimate the double integrals of the pixel in Eqs. (2.7) and (2.15). There is an assumption behind this approximation that the values of $f(x_i, y_j)V^*_{nm}(x_i, y_j)$ of all the points inside a image pixel are the same. This is not true because the distributions of Zernike and pseudo-Zernike polynomials $V^*_{nm}(x, y)$ within one pixel can vary significantly when the orders are high. Figures 3.3 and 3.4 show the distribution of $V^*_{nm}(x, y)$ of Zernike moments within one of the central pixels. Similarly, Figs. 3.5 and 3.6 show the distribution of $V^*_{nm}(x, y)$ of pseudo-Zernike moments. Within the four figures, $n = 30$ and $m = 10$ in sub-figure (a), while $n = 100$ and $m = 10$ in sub-figure (b). As a result, when



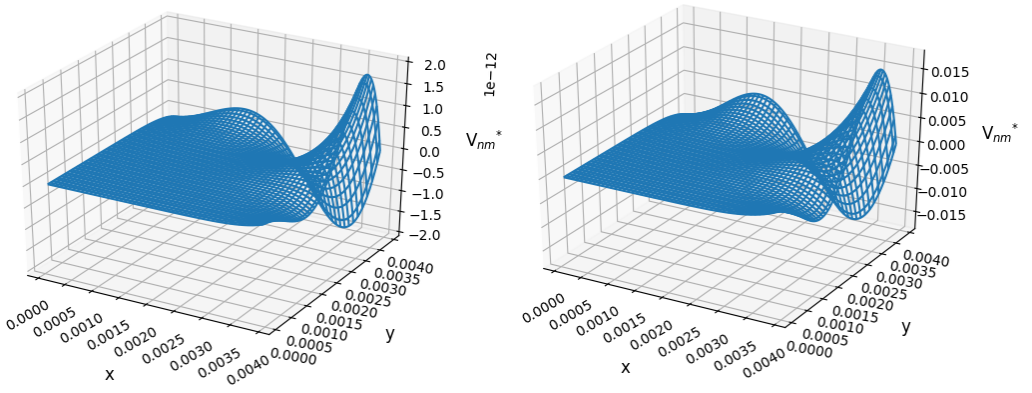
(a) $n = 30, m = 10$ (b) $n = 100, m = 10$

Figure 3.3: Distribution of the real part of $V_{nm}^*(x, y)$ in Zernike moments within one of the central pixels with location of (256, 257) in an image sized at 512×512. In sub-figure (a), $n = 30, m = 10$. In sub-figure (b), $n = 100, m = 10$. The distribution varies more significantly when n is larger.



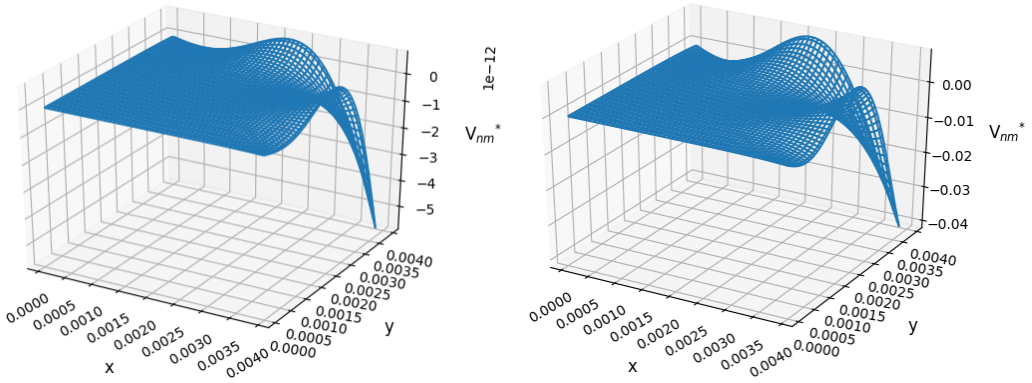
(a) $n = 30, m = 10$ (b) $n = 100, m = 10$

Figure 3.4: Distribution of the imagine part of $V_{nm}^*(x, y)$ in Zernike moments within one of the central pixels with location of (256, 257) in an image sized at 512×512. In sub-figure (a), $n = 30, m = 10$. In sub-figure (b), $n = 100, m = 10$. The distribution varies more significantly when n is larger.



(a) $n = 30, m = 10$ (b) $n = 100, m = 10$

Figure 3.5: Distribution of the real part of $V^*_{nm}(x, y)$ in pseudo-Zernike moments within one of the central pixels with location of (256, 257) in an image sized at 512×512. In sub-figure (a), $n = 30, m = 10$. In sub-figure (b), $n = 100, m = 10$. The distribution varies more significantly when n is larger.



(a) $n = 30, m = 10$ (b) $n = 100, m = 10$

Figure 3.6: Distribution of the imagine part of $V^*_{nm}(x, y)$ in pseudo-Zernike moments within one of the central pixels with location of (256, 257) in an image sized at 512×512. In sub-figure (a), $n = 30, m = 10$. In sub-figure (b), $n = 100, m = 10$.

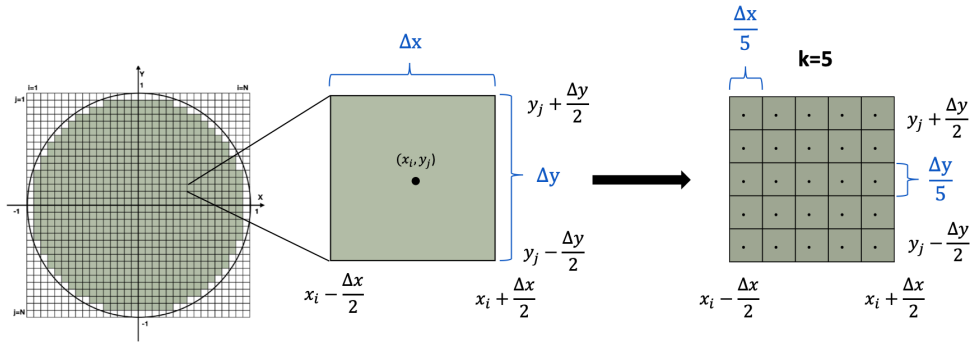


Figure 3.7: An example of the sub-region scheme with $k = 5$. Without the sub-region scheme, only the value of the central point (x_i, y_j) in each pixel is used to compute moments. The sampling intervals are Δx and Δy. With 5×5 sub-region scheme, the values of 25 points inside each pixel are used. The sampling intervals are $\frac{\Delta x}{5}$ and $\frac{\Delta y}{5}$.

using Eqs. (2.10) and (2.16) to calculate moments, a significant amount of approximation errors will occur.

In this research, to improve the computational accuracy of Zernike and pseudo-Zernike moment functions, we have divided each pixel into $k \times k$ sub-regions with the same weights. An example of the sub-region scheme with $k = 5$ is shown in Fig 3.7. Without the sub-region scheme, only the value of the central point (x_i, y_j) in each pixel is used to compute moments. The sampling intervals are Δx and Δy. With $k \times k$ sub-regions scheme, the values of k^2 points inside each pixel are used. The sampling intervals are $\frac{\Delta x}{k}$ and $\frac{\Delta y}{k}$. To compute the moments, We have rewritten Eqs. (2.10) and (2.16) as [21]

$$\hat{A}_{nm} = \frac{n+1}{\pi} \sum_{x_i{}^2 + y_j{}^2 \leq 1} \sum f(x_i, y_j) h_{nm}(x_i, y_j), \tag{3.19}$$

where

$$h_{nm}(x_i, y_j) = \sum_{s=1}^{k} \sum_{t=1}^{k} V^*_{nm}(x_{is}, y_{jt}) \frac{\Delta x}{k} \frac{\Delta y}{k} \tag{3.20}$$

and in Eq. (3.20),

$$x_{is} = x_i + (s - \frac{k+1}{2}) \times \frac{\Delta x}{k} \tag{3.21}$$

$$y_{jt} = y_j + (t - \frac{k+1}{2}) \times \frac{\Delta y}{k} \tag{3.22}$$

In practice, the program treats an image sized at $N \times N$ as an image with $N \times k \times N \times k$ pixels. Each sub-region is treated as a new pixel. All the new pixels inside the same original pixel have the same gray value. Then we use another format of Eq. (3.19), as

$$\hat{A}_{nm} = \frac{n+1}{\pi} \sum_{x_{is}{}^2 + y_{jt}{}^2 \leq 1} \sum \sum_{s=1}^{k} \sum_{t=1}^{k} f(x_{is}, y_{jt}) V^*_{nm}(x_{is}, y_{jt}) \frac{\Delta x}{k} \frac{\Delta y}{k} \tag{3.23}$$

to compute the moments. It should be emphasized that when an original pixel has any parts that are out of the unit circle, all the new pixels in this original pixel are discarded.

3.3 Summary

There are mainly three kinds of methods used to compute the Zernike and pseudo-Zernike moments. We have compared them and chosen the most efficient and accurate recursive methods in our algorithm. This chapter also explains how the symmetric property is utilized to reduce the computation time, and how the sub-region scheme is used to improve the accuracy of the moments computation.

Chapter 4

GPU Implementation of Moments Computation

4.1 CUDA Programming Model

Our GPU-accelerated algorithm runs on NVIDIA GPU and is developed based on the CUDA programming model. In the CUDA programming model, a program is executed by the *host*, which refers to the CPU and its memory, and the *device*, which refers to the GPU and its memory. A program function expected to run on GPU is called a *kernel*. Host CPU code manages memory on both the host and the device and launches kernels to GPU. After being called by host CPU code, a kernel is executed N times in parallel by N different CUDA *threads*. As shown in Fig. 4.1, threads are organized in one-, two-, or three-dimensional structures named *blocks*. Similarly, blocks are organized in *grids*. Each thread within the block and each block within the grid can be identified by a three-component vector. The numbers of threads per block and blocks per grid are defined by programmers.

CUDA provides multiple types of memory for the host and the device to access data. Different memory types have different scopes and speeds. Global memory, constant memory, and local memory are off-chip memory. Global memory resides in the device and can be written and read both by the host and the device. It is by far the largest and most commonly used, though the slowest memory storage on the GPU [14]. Constant memory can be read and written by the host, but are read-only for the device. Local memory is private for each thread, accessible only to a single thread, and as slow as global memory. Registers and shared memory are on-chip memory

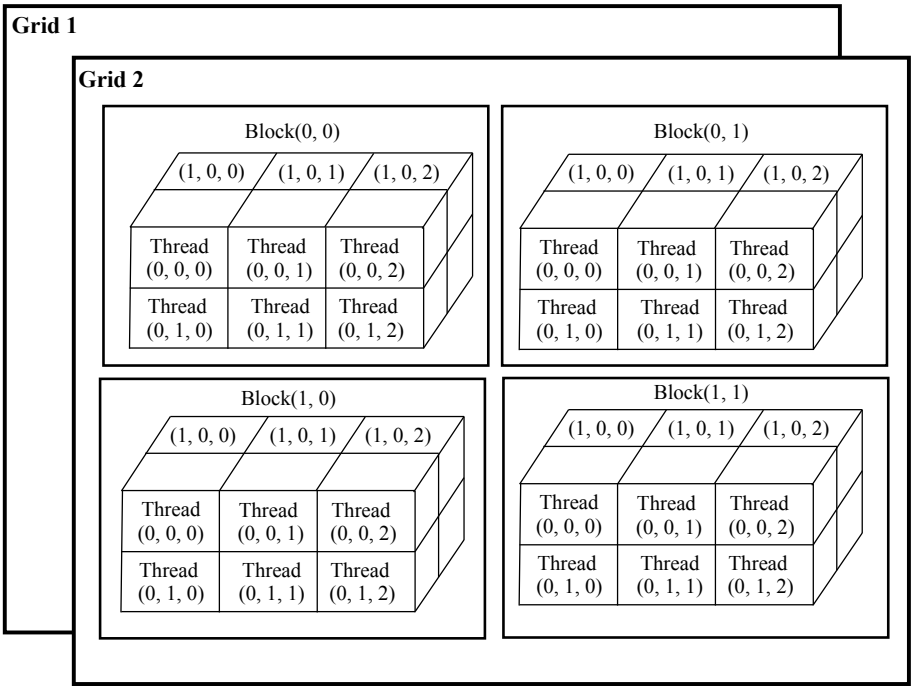


Figure 4.1: The three-level hierarchy of threads in CUDA programming model. The threads are organized in blocks, and the blocks are organized in grids. The threads shown in this figure are three-dimensional, and the blocks are two-dimensional.

that provides data at a very high speed. Registers have the same scope as local memory. Shared memory of a block is accessible for all threads within the block. It is often used for threads in a block to communicate and share data.

To execute a kernel on a device, a CUDA program needs to declare and allocate global memory on the device, and transfer data from the host memory to the device memory. After the kernel execution is finished, the results are copied from the device to the host. Minimizing data transfer between the device and the host could improve the overall performance because the peak theoretical bandwidth between host memory and device memory is lower

than that between the device memory and the GPU [28].

4.2 GPU Implementation Structure

The programs for Zernike and pseudo-Zernike moments both include four main phases: pre-processing, image data reordering, moments computing, and image reconstruction. One or more CUDA kernels are used in each phase, and will be introduced in the following.

Pre-processing

In the pre-processing phase, the CPU program reads the image, divides each pixel into $k \times k$ sub-regions and gets the coordinates of the symmetric pixels. The GPU kernel *Origin_Kernel* is used to calculate the values of ρ and θ for each pixel and mask the image to discard the pixels with any sub-region falling out of the unit disk.

Image Data Reordering

In NVIDIA GPU implementations, 32 threads within a block are executed in a group, known as a *warp*. Threads within a warp must execute the same instruction at the same time. To implement the symmetric optimization, the computing kernel needs to judge whether a pixel is eight-symmetric or four-symmetric and choose corresponding instructions. If kernels process image data in the initial sequence, there will be a number of warp divergences in CUDA, resulting in a poor computation efficiency [13].

To solve this problem, Xia [27] proposed an algorithm to reorder the image data. An example of how the pixels are reordered is shown in Fig. 4.2. With this method, pixels are reordered in a sequence that all eight-symmetric

pixels are put in the first part of the sequence, followed by all four-symmetric pixels. Thus only one warp will have divergence and the efficiency is improved significantly. This method is implemented in *Reorder_Kernel*.

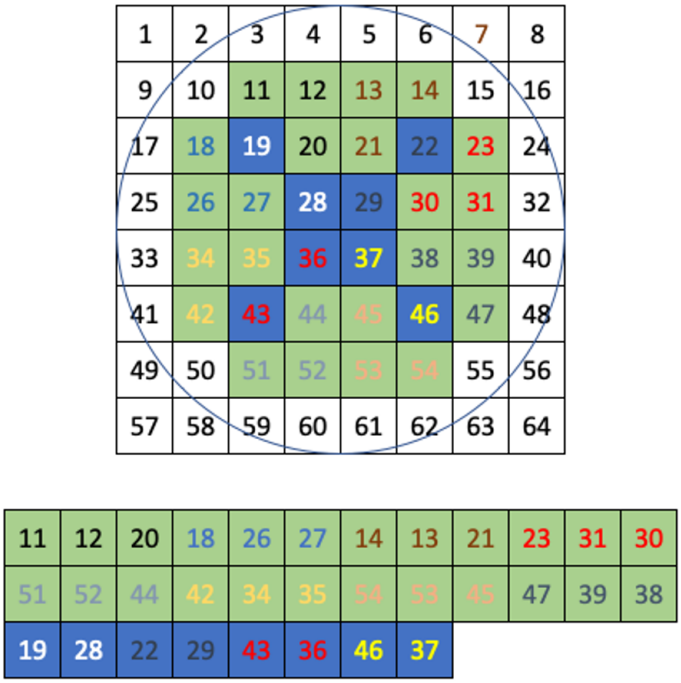


Figure 4.2: An example of how the pixels are reordered with Xia's method [27]. The pixels are reordered in a sequence that all eight-symmetric pixels are put in the first part of the sequence, followed by all four-symmetric pixels.

Moments Computing

Our moments computing phase involves four kernels: *Rho_Kernel*, *Exp_Kernel*, *Znm_Kernel*, and *Znm_Sum_Kernel*.

Rho_Kernel and *Exp_Kernel* calculate the power functions of ρ and the exponential functions respectively for each pixel. Assuming the number of

pixels involved in the moments computation is N, and the maximum order computed is T, then the output of *Rho_Kernel* is like

$$\begin{bmatrix} \rho_0^0 & \rho_1^0 & \cdots & \rho_{N-2}^0 & \rho_{N-1}^0 \\ \rho_0^1 & \rho_1^1 & \cdots & \rho_{N-2}^1 & \rho_{N-1}^1 \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \rho_0^{T-1} & \rho_1^{T-1} & \cdots & \rho_{N-2}^{T-1} & \rho_{N-1}^{T-1} \\ \rho_0^T & \rho_1^T & \cdots & \rho_{N-2}^T & \rho_{N-1}^T \end{bmatrix} \tag{4.1}$$

and the output of *Exp_Kernel* is like

$$\begin{bmatrix} e^{-j0\theta_0} & e^{-j0\theta_1} & \cdots & e^{-j0\theta_{N-2}} & e^{-j0\theta_{N-1}} \\ e^{-j1\theta_0} & e^{-j1\theta_1} & \cdots & e^{-j1\theta_{N-2}} & e^{-j1\theta_{N-1}} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ e^{-j(T-1)\theta_0} & e^{-j(T-1)\theta_1} & \cdots & e^{-j(T-1)\theta_{N-2}} & e^{-j(T-1)\theta_{N-1}} \\ e^{-jT\theta_0} & e^{-jT\theta_1} & \cdots & e^{-jT\theta_{N-2}} & e^{-jT\theta_{N-1}} \end{bmatrix} \tag{4.2}$$

Within the two kernels, each thread is responsible for computing data of one pixel. Their outputs will be the inputs of *Znm_Kernel*.

Znm_Kernel uses the outputs of *Rho_Kernel* to computes the values of $R_{nm}(\rho)$. Equations (3.10) and (3.11) are employed in *Znm_Kernel* for Zernike moments, and Eqs. (3.12) to (3.15) are used for pseudo-Zernike moments. After this, it calculates the values of $f(x_{is}, y_{jt})V_{nm}^*(x_{is}, y_{jt})$ (noted as $Z_{nm}^*(x_{is}, y_{jt})$ for Zernike moments and $PZ_{nm}^*(x_{is}, y_{jt})$ for pseudo-Zernike moments) in Eq. (3.23) with all different n and m for each pixel. The pseudocode of *Znm_Kernel* for Zernike moments is provided in Algorithm 1, and that of pseudo-Zernike moments is shown in Algorithm 2.

Since the values of $f(x_{is}, y_{jt})V_{nm}^*(x_{is}, y_{jt})$ are given by *Znm_Kernel*, *Znm_Sum_Kernel* calculates the moments based on Eq. (3.23). It adds all $Z_{nm}^*(x_{is}, y_{jt})$ or $PZ_{nm}^*(x_{is}, y_{jt})$ with the same n and m up. More details about *Znm_Sum_Kernel* and the parallel reduction algorithm used in it will be introduced in Section 4.3.

Algorithm 1 Znm_Kernel for Zernike moments

function Znm_Kernel($ar\_znm, ar\_img\_k, ar\_exp\_k, ar\_rho\_k$)
  $total\_tmp \leftarrow 0$
  if current thread should compute a value **then**
    if current thread processes a pixel that is eight-symmetric **then**
      Load gray values of pixel P_i ($1 \leq i \leq 8$) into $loaded\_pixels[i]$
      $total\_tmp \leftarrow \sum_{i=1}^{8} loaded\_pixels[i]$
      $ar\_znm[0][0] \leftarrow total\_tmp$
      $Rpp[0] \leftarrow 1$
      $Rp[1] \leftarrow \rho$
      for n from 1 to maximum order T **do**
        $total\_tmp \leftarrow \sum_{i=1}^{8} loaded\_pixels[i] \times e^{-jn\theta_i}$
        ▷ $e^{jm\theta_i}$ is the value of $e^{jm\theta}$ for P_i in Table 3.1
        $R[n] \leftarrow \rho^n$
        $ar\_znm[n][n] \leftarrow total\_tmp \times R[n]$
        if n is even **then**
          $R[0] \leftarrow 2 \times \rho \times Rp[1] - Rpp[0]$
        end if
        for m from $n-2$ to 1 by -2 **do**
          $total\_tmp \leftarrow \sum_{i=1}^{8} loaded\_pixels[i] \times e^{-jm\theta_i}$
          $R[m] \leftarrow \rho \times (Rp[m-1] + Rp[m+1]) - Rpp[m]$
          $ar\_znm[n][m] \leftarrow total\_tmp \times R[m]$
        end for
        if n is larger than 1 **then**
          for i from 0 to n **do**
            $Rpp[i] \leftarrow Rp[i]$
            $Rp[i] \leftarrow R[i]$
          end for
        end if
      end for
    end if
    if current thread processes a pixel that is four-symmetric **then**
      Take same steps with the above if block, except that i is always from 1 to 4, and the value of $e^{jm\theta}$ for P_i is from Table 3.2
    end if
  end if
end function

Algorithm 2 Znm_Kernel for pseudo-Zernike moments

function Znm_Kernel($ar\_znm, ar\_img\_k, ar\_exp\_k, ar\_rho\_k$)
  $total\_tmp \leftarrow 0$
  if current thread should compute a value **then**
    if current thread processes a pixel that is eight-symmetric **then**
      Load gray values of pixel P_i ($1 \leq i \leq 8$) into $loaded\_pixels[i]$
      $total\_tmp \leftarrow \sum_{i=1}^{8} loaded\_pixels[i]$
      $ar\_znm[0][0] \leftarrow total\_tmp$
      $R[n] \leftarrow 1$
      $A[n] \leftarrow 1$
      for n from 1 to maximum order T **do**
        $R[n] \leftarrow \rho^n$
        $A[n] \leftarrow R[n]$
        $Rad \leftarrow R[n]$
        $total\_tmp \leftarrow \sum_{i=1}^{8} loaded\_pixels[i] \times e^{-jn\theta_i}$
        ▷ $e^{jm\theta_i}$ is the value of $e^{jm\theta}$ for P_i in Table 3.1
        $ar\_znm[n][n] \leftarrow total\_tmp \times Rad$
        for m from $n-1$ to 1 **do**
          $A[m] \leftarrow \rho \times (R[m-1] + R[m]) - A[m]$
          $Rad \leftarrow A[m] + A[m+1] - R[m]$
          $R[m] \leftarrow Rad$
          $total\_tmp \leftarrow \sum_{i=1}^{8} loaded\_pixels[i] \times e^{-jm\theta_i}$
          $ar\_znm[n][m] \leftarrow total\_tmp \times Rad$
        end for
        $A[0] \leftarrow 2 \times \rho \times R[0] - A[0]$
        $Rad \leftarrow A[0] + A[1] - R[0]$
        $total\_tmp \leftarrow \sum_{i=1}^{8} loaded\_pixels[i]$
        $ar\_znm[n][0] \leftarrow total\_tmp \times Rad$
        $R[0] \leftarrow Rad$
      end for
    end if
    if current thread processes a pixel that is four-symmetric **then**
      Take same steps with the above if block, except that i is always from 1 to 4, and the value of $e^{jm\theta}$ for P_i is from Table 3.2
    end if
  end if
end function

Image Reconstruction

The phase of image reconstruction has four kernels: *Rho_Kernel*, *Exp_Kernel*, *Vnm_Kernel*, and *Fxy_Kernel*. *Rho_Kernel* and *Exp_Kernel* are similar to those in moments computing. *Vnm_Kernel* calculates $V_{nm}(x_i, y_j)$ of each pixel according to Eq. (2.5), where the values of $R_{nm}(\rho)$ are calculated with the same recursive methods in *Znm_Kernel*. *Fxy_Kernel* uses the output of *Vnm_Kernel* and the moments computed in *Znm_Sum_Kernel* to compute $\hat{f}(x_i, y_j)$ based on Eq. (2.12).

4.3 GPU Algorithm Optimizations

Dynamic Data Partition

GPU's resources, such as registers and memory, will limit its processing capacity. As discussed in Section 4.1, when transferring data between host and device, data is copied from host memory to device global memory or verse vice. In GPU implementation, if either the size of an image or the maximum order T is too large, there will be too much data needed to be transferred and it will require more memory than GPU can provide. Under such conditions, we need to partition pixels into several parts and launch related kernels multiple times through a loop.

To make full use of global memory, instead of setting a fixed number of pixels for a loop to process, we partition pixels dynamically before launching kernels. The maximum number of pixels that could be processed in a loop is limited by current free memory and the size of memory required per pixel. The former can be obtained by calling CUDA API function *cudaMemGetInfo*. For the latter, we have analyzed the major memory used by each pixel in moments computation phase, as displayed in Table 4.1. Other than the memory discussed in Table 4.1, we also need extra memory to deal with

Table 4.1: Major memory used by each pixel in the moments computation phase.

Data	Type	Type Size	Amount	Total Size
$exp^{jk\theta}$	Complex64	8 Bytes	$T+1$	$8 \times (T+1)$
ρ^k	Float32	4 Bytes	$T+1$	$4 \times (T+1)$
$Z^*_{nm}(x,y)$	Complex64	8 Bytes	$\frac{(T+1)(T+2)}{2}$	$4 \times (T+1)(T+2)$
$PZ^*_{nm}(x,y)$	Complex64	8 Bytes	$\frac{(T+2)^2-(T\%2)}{4}$	$2\times((T+2)^2-(T\%2))$

some integer parameters. Based on the above analysis, we have carefully set the number of pixels in each loop of moments computation, to ensure that enough memory is provided to avoid allocation failure, and the global memory is efficiently utilized.

Minimizing Data Transfers and Memory Allocation

As discussed in Section 4.1, when transferring data between host and device, a straightforward sequence is allocating memory, copying input data to device, executing kernel, copying output results to host, and releasing memory. Data transfer and memory allocation could consume considerable time when the size of data being processed is large. We have used the following two methods to minimize these operations and reduce time.

- Move memory allocation out of the loop and reuse the allocated memory. Except for the last loop, the numbers of pixels involved in each loop are the same. We do not need to allocate and free the memory for data showed in Table 4.1 in each loop. Instead, our program allocates memory for those data once before the loop, and free them after the loop is finished. For the last loop, although the number of pixels processed is less than that of the former loops, the memory can still be

used with some vacancy.

- Keep intermediate data in device for future operation. In our program, the output results of some kernels will be the input of the next kernels. For example, ρ^k from *Rho_Kernel* and $e^{-jk\theta}$ from *Exp_Kernel* will be used in *Znm_Kernel*, and $Z^*_{nm}(x_i, y_j)$ or $PZ^*_{nm}(x_i, y_j)$ computed by *Znm_Kernel* will be added up in *Znm_Sum_Kernel*. To minimize data transfer, instead of transferring data from device to host after the kernel, we leave the data in device and use them directly in the next kernel. The data is updated during each loop and will be destroyed after loops end.

Reduction Optimization

As we have introduced in Section 4.2, the function of *Znm_Sum_Kernel* is computing the moments based on Eq. (3.23). The following part explains the process in the program of computing Zernike moments, and that of pseudo-Zernike moments is analogous.

In *Znm_Sum_Kernel*, assuming the input Z^*_{nm} includes data of N pixels, then it is a matrix with N rows and $((T+2)^2 - (T\%2))/4$ columns, as

$$\boldsymbol{Z^*_{nm}} = \begin{bmatrix} Z^*_{0,0}(x_0, y_0) & \cdots & Z^*_{n,m}(x_0, y_0) & \cdots & Z^*_{T,T}(x_0, y_0) \\ Z^*_{0,0}(x_1, y_1) & \cdots & Z^*_{n,m}(x_1, y_1) & \cdots & Z^*_{T,T}(x_1, y_1) \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ Z^*_{0,0}(x_{N-1}, y_{N-1}) & \cdots & Z^*_{n,m}(x_{N-1}, y_{N-1}) & \cdots & Z^*_{T,T}(x_{N-1}, y_{N-1}) \end{bmatrix} \tag{4.3}$$

Each row shows the moments with different order n and repetition m of a single pixel. Each column includes the moments of the same order n and repetition m for all pixels. The function of *Znm_Sum_Kernel* is called to calculate the sum of each column.

Shared memory is used here for threads to share data, because it has much

higher bandwidth and much lower latency than local or global memory, and is accessible to all threads within a block [29]. We use parallel reduction [30] to achieve the adding operations in Eq. (3.23) efficiently.

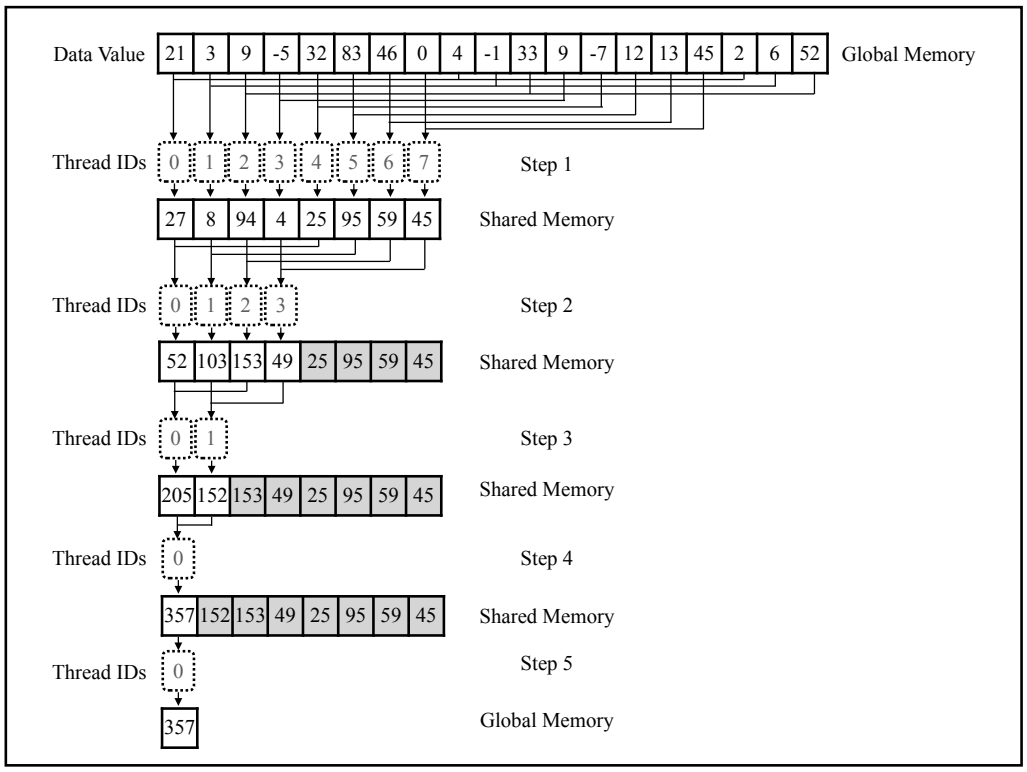


Figure 4.3: An example of parallel reduction process within an eight-threads block. The shared memory is utilized to improve the reduction speed.

When launching kernel, we set $((T+2)^2 - (T\%2))/4$ blocks in a grid, thus each block calculates one column. In the kernel, threads will load data from global memory into shared memory first. Because threads in a block are limited, when the number of threads is less than N, a thread needs to load more than one value, add them up and store the result in shared memory. An example of the reduction process within a block with eight threads is shown in Fig. 4.3. In this example, each of *thread* 0 to *thread* 2 will load three values and store their sum in shared memory, while *thread* 3 to *thread*

7 will load and add up two values. In the following steps, threads read data from shared memory, execute calculation, and write the result back to shared memory. After all the data are added up, *thread* 0 copies the result from shared memory to global memory.

4.4 Summary

There are four phases in the GPU implementation: pre-processing, image data reordering, moments computing, and image reconstruction. Based on the features of the CUDA programming model, we have utilized three measures to improve the speed of our program. These optimizations include partitioning the pixels dynamically to make full use of the global memory, minimizing the data transfers and memory allocation, and making use of the shared memory in parallel reduction.

Chapter 5

Experimental Results

5.1 Experiment Setup

In this research, our programs are developed in Python and CUDA C++. Python is used in the CPU part of the program, and CUDA C++ is used to write the GPU kernels. The PyCUDA library is imported to access NVIDIA's CUDA parallel computation API. Most real numbers and computations are processed with the 32-bit floating-point format. The complex numbers are treated by data type complex64, which is composed of two 32-bit precision floats. The computer system employed in this experiment is equipped with an Intel Core i5-8400 2.80GHz CPU with 16GB RAM, running on Ubuntu 18.04.4 LTS operating system. The GPU on our system is TITAN V with a max clock rate of 1.46 GHz and 12GB global memory.

Figure 5.1 shows the two testing images utilized in our experiment and both are sized at 512×512 with 256 gray levels.

(a) (b)

Figure 5.1: The two utilized testing images, which are sized at 512×512 with 256 gray levels.

To evaluate the accuracy of image reconstruction performance, we have utilized the Peak Signal to Noise Ratio (PSNR) to measure the qualities of the reconstructed images. The definition of PSNR is given by

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

where Max is the maximum gray level of the evaluated image, which is 255 in our experiment, and MSE is the Mean Square Error relating the image $f(x_i, y_j)$ and its reconstructed version $\widehat{f}(x_i, y_j)$, assuming both images are sized at $M \times N$, then

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

The minimum and maximum integer gray values of a pixel in our testing images are 0 and 255. Therefore, in our reconstructed image $\widehat{f}(x_i, y_j)$, all values smaller than 0 and larger than 255 are set to 0 and 255, respectively. In most cases, the higher PSNR value indicates the less difference between the two compared images.

Since Zernike moments and pseudo-Zernike moments are defined on the unit disk, during our moments calculating, we will discard any image pixel with one of the k^2 sub-regions falling outside of the unit disk. On the other hand, when we perform image reconstructions from Zernike or pseudo-Zernike moments, only the image pixels with all k^2 sub-regions sitting inside the unit disk are reconstructed.

5.2 Experimental Results of Zernike Moments

Computational Efficiency

We have calculated Zernike moments with different maximum order T and sub-region scheme k. The running times for computing Zernike moments of

image Fig. 5.1(a) are shown in Table 5.1. Due to the same image size, the computation times of Zernike moments for image Fig. 5.1(b) are the same as those of Fig. 5.1(a).

Table 5.1: Zernike moments computation time (in seconds) for Fig. 5.1(a), with maximum order T from 100 to 700 and sub-region scheme value k from 1 to 11. The times increase when T or k increases.

k\T	100	200	300	400	500	600	700
1	0.11	0.23	0.43	0.53	0.76	1.17	1.72
3	0.60	1.83	3.22	3.79	5.74	9.40	13.78
5	1.55	5.04	8.95	10.30	15.68	25.91	38.05
7	2.95	9.83	17.56	20.11	30.58	50.61	74.36
9	4.82	16.24	29.04	33.19	50.52	83.51	122.74
11	7.17	24.13	43.25	49.48	75.36	124.63	181.13

As demonstrated in Table 5.1, without applying $k \times k$ sub-region scheme, it takes 1.72 seconds to calculate Zernike moments with maximum order $T = 700$ of an image sized at 512×512. When the numerical sub-region scheme is applied, the computation time is about 181.13 seconds with $k = 11$.

We have compared our results with the Zernike moments computation times of Deng's recursive method in Ref. [9], where the experiments are executed in CPU. They have computed the Zernike moments of images sized at 512×512 up to 500 orders. Table 5.2 shows the comparison results with the maximum number T from 100 to 500. The speedup rate is the ratio of the time used in Deng's method divided by that in our experiment. From Table 5.2, when T increases, the speedup rate increases, indicating a better acceleration of the GPU implementations. When T is 500, our result speeds up the computation by seven times.

The execution time of some GPU kernels and their corresponding CPU

Table 5.2: Comparison of Zernike moments computation times in CPU [9] and GPU. The speedup rate is the ratio of the time used in CPU divided by that in GPU. when T increases, the speedup rate increases, indicating a better acceleration of the GPU implementations.

Maximum Order T	100	200	300	400	500
CPU Results in Ref. [9]	0.22	0.87	1.87	3.32	5.29
Our GPU Results	0.11	0.23	0.43	0.53	0.76
Speedup Rate	2.0	3.8	4.4	6.3	7.0

functions is also compared. We have adapt the *Origin_Kernel*, *Rho_Kernel* and *Exp_Kernel* into functions that executed in CPU. The data of images with different sizes are computed through the CPU functions and the GPU kernels. The maximum moments order T is set as 700, and the $k \times k$ sub-region scheme is not used. The result is shown in Table. 5.3. The speeds of GPU for these kernels are hundreds or thousands of times faster than CPU.

Computational Accuracy of Image Reconstructions

Some reconstructed images from Zernike moments of Fig. 5.1(a) are shown in Figs. 5.2 and 5.3. The PSNR values of the reconstructed images are shown in Table 5.4.

From the reconstructed images of Fig. 5.1(a) shown in Figs. 5.2 and 5.3, we can observe that when the maximum Zernike moment order T increases, more details of the original image will be reconstructed. However, when $k = 1$, since the $k \times k$ sub-region scheme optimization has not been applied, the reconstructed images are distorted in the edge areas when T goes higher. As demonstrated in Figs. 5.2 and 5.3, when k increases, the distorted areas will be reduced. When k is greater than five, the distortion will visually be eliminated in the reconstructed images.

Table 5.3: The execution time of some GPU kernels and their corresponding CPU functions with different image sizes. The maximum moments order T is 700, and the $k \times k$ sub-region scheme is not used. The speeds of GPU for these kernels are hundreds or thousands of times faster than CPU.

Image Size	Kernel Name	CPU	GPU	Speedup Rate
128×128	Origin_Kernel	1.49E-02	2.65E-05	563
	Rho_Kernel	1.66E-02	4.98E-05	333
	Exp_Kernel	3.04E-02	3.50E-05	867
256×256	Origin_Kernel	5.83E-02	2.67E-05	2,184
	Rho_Kernel	6.04E-02	1.88E-04	322
	Exp_Kernel	1.18E-01	9.16E-05	1,293
512×512	Origin_Kernel	2.48E-01	3.41E-05	7,269
	Rho_Kernel	2.40E-01	6.10E-04	393
	Exp_Kernel	4.52E-01	3.35E-04	1,352

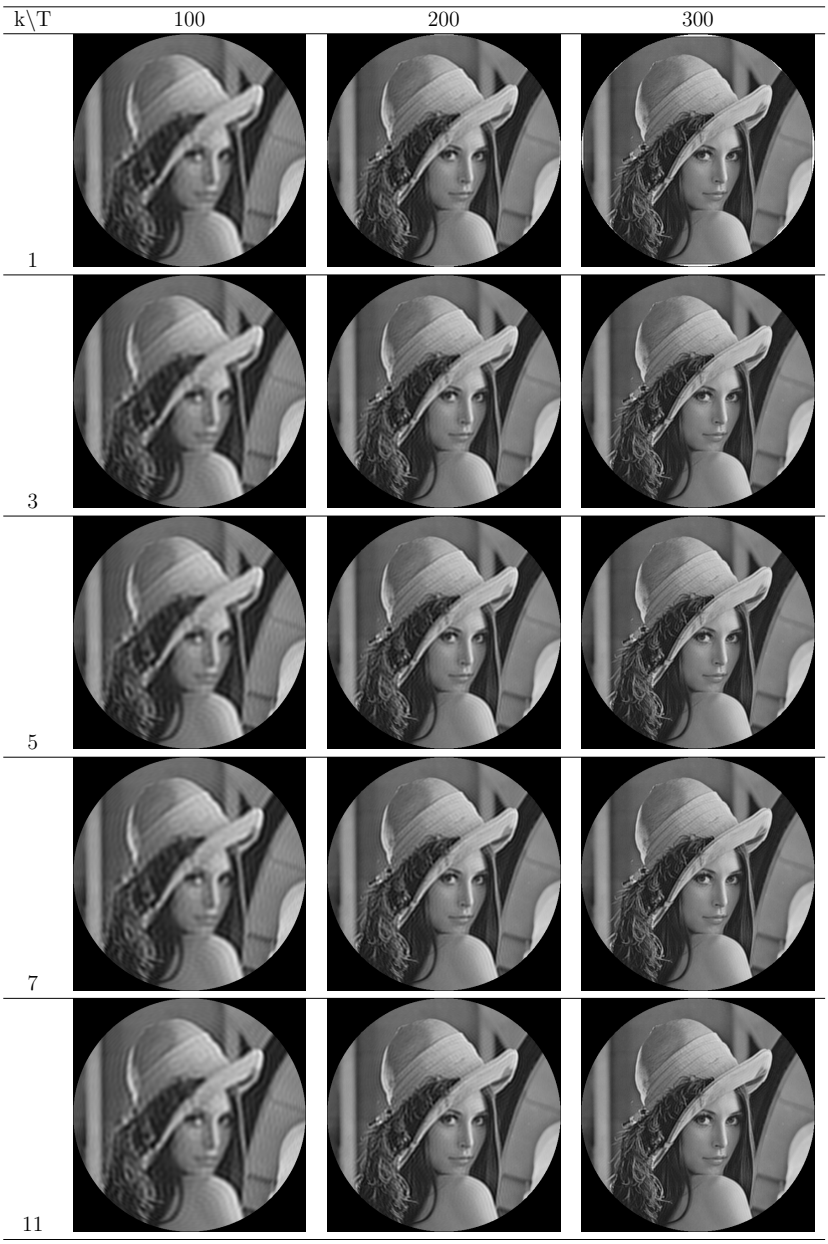


Figure 5.2: Images reconstructed from Zernike moments of Fig. 5.1(a) with maximum order T from 100 to 300 and sub-region scheme value k from 1 to 11.

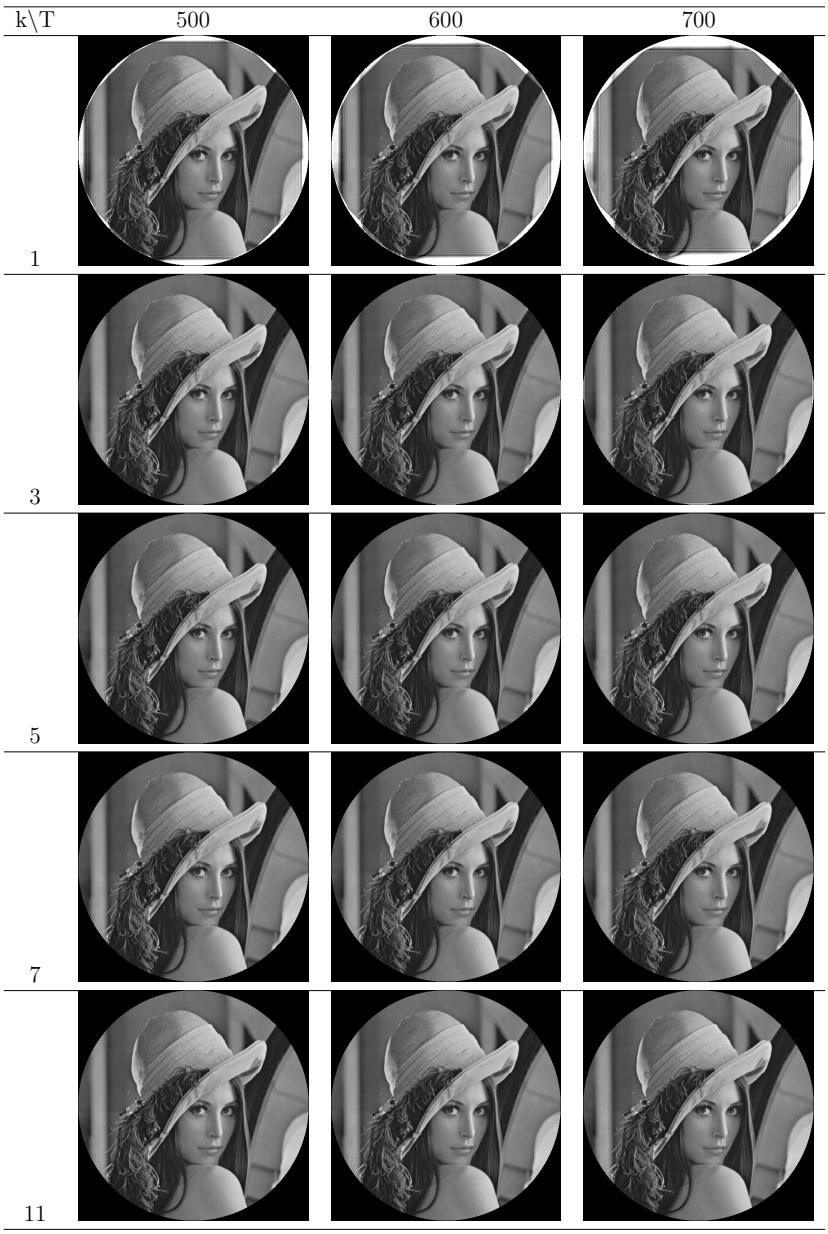


Figure 5.3: Images reconstructed from Zernike moments of Fig. 5.1(a) with maximum order T from 400 to 700 and sub-region scheme value k from 1 to 11.

Table 5.4: PSNR values of images reconstructed from Zernike moments of Fig. 5.1(a) with maximum order T from 100 to 700 and sub-region scheme value k from 1 to 11.

k\T	100	200	300	400	500	600	700
1	26.93	29.61	27.50	23.95	20.87	18.545	16.59
3	26.90	30.59	33.81	36.65	37.45	34.59	32.69
5	26.90	30.60	33.80	36.74	39.36	41.77	43.93
7	26.90	30.60	33.80	36.74	39.35	41.89	44.50
9	26.90	30.60	33.80	36.74	39.36	41.88	44.52
11	26.90	30.59	33.80	36.74	39.37	41.88	44.52

In this research, we have also performed the image reconstructions from Zernike moments of Fig. 5.1(b). Some reconstructed images of Fig. 5.1(b) are presented in Figs. 5.4 and 5.5. Table 5.5 shows the PSNR values of the reconstructed images. These results demonstrate that the performances of image reconstructions from Zernike moments improve in a similar way to those of Fig. 5.1(a), though the PSNR values are lower. This can be explained by the fact that the image reconstruction performance will be affected by the contents of an image.

It should be noted that we have also applied the 64-bit floating-point format in our experiments. The times used to compute Zernike moments are shown in Table 5.6. Although the running times are longer, the PSNR values of the reconstructed images are almost identical with those of 32-bit floating-point format. Therefore, we have applied the 32-bit precision in all other experiments.

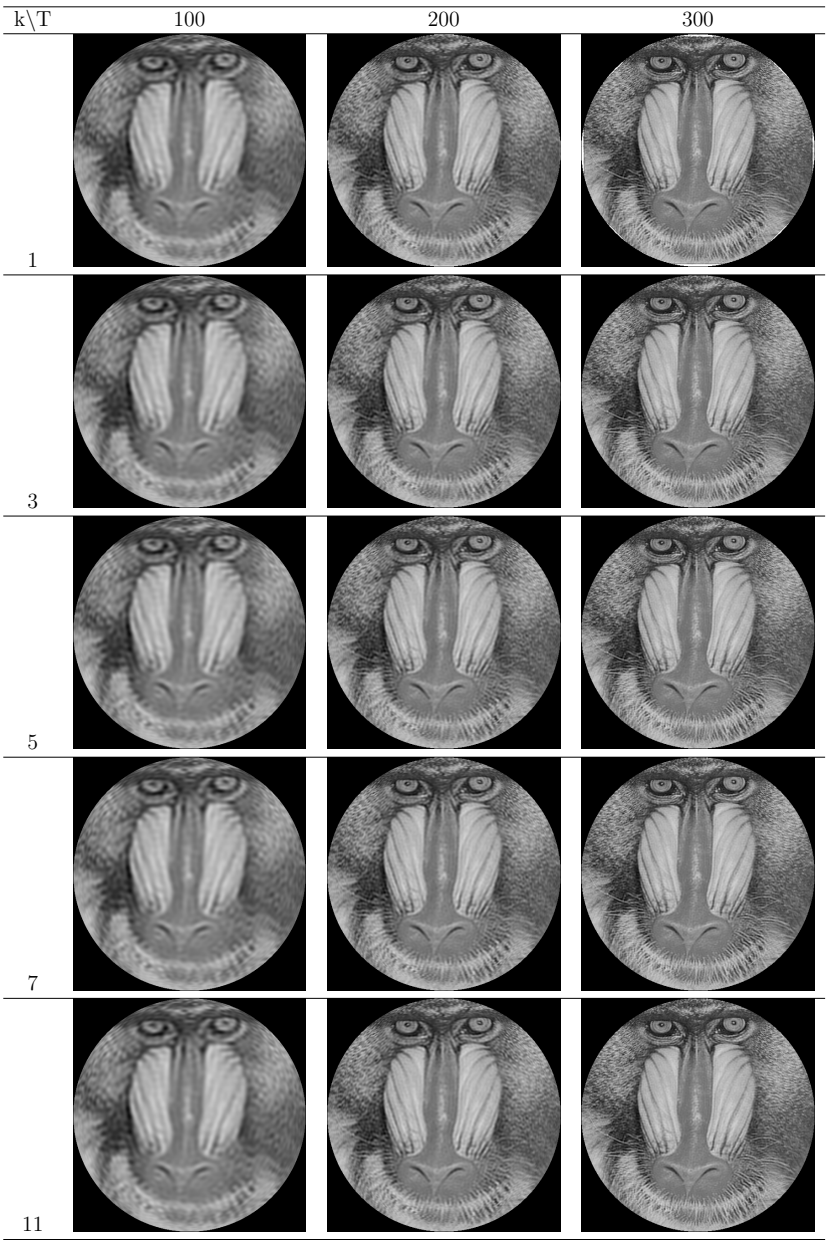


Figure 5.4: Images reconstructed from Zernike moments of Fig. 5.1(b) with maximum order T from 100 to 300 and sub-region scheme value k from 1 to 11.

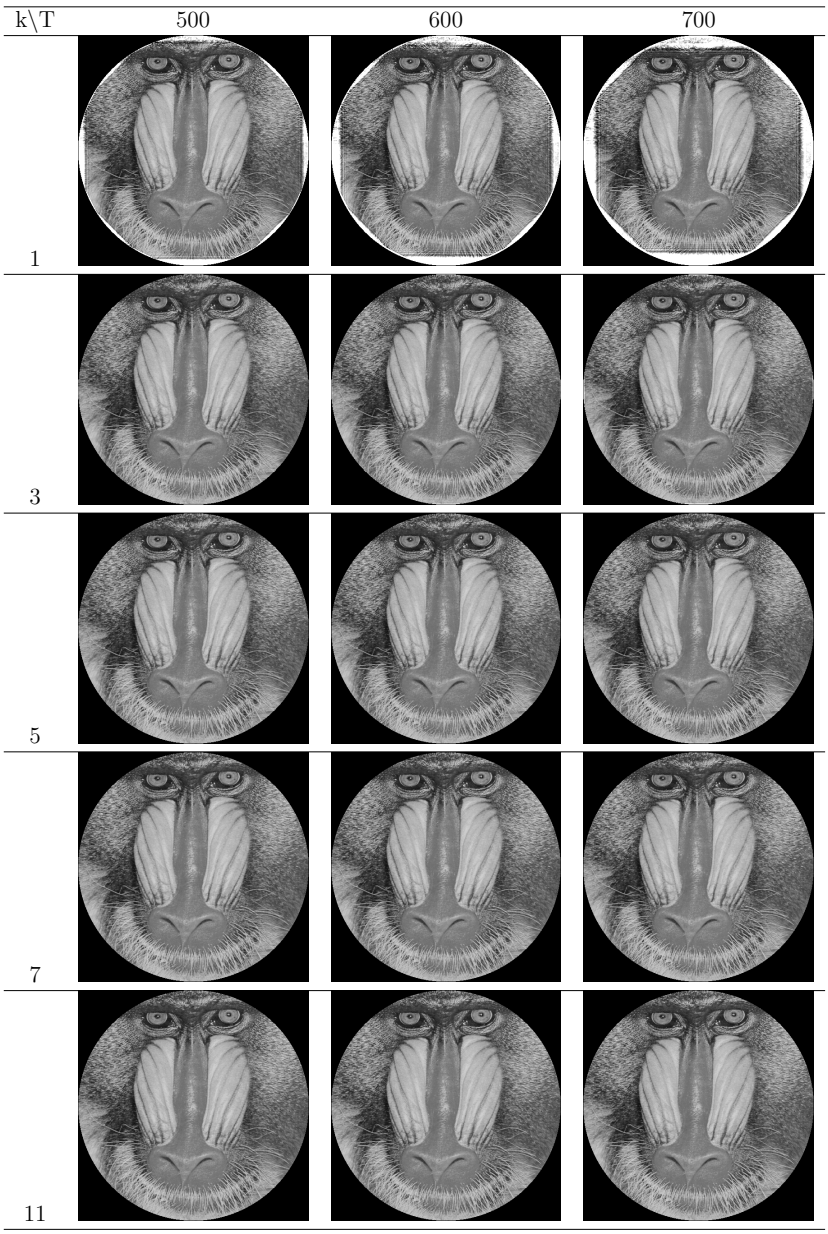


Figure 5.5: Images reconstructed from Zernike moments of Fig. 5.1(b) with maximum order T from 400 to 700 and sub-region scheme value k from 1 to 11.

Table 5.5: PSNR values of images reconstructed from Zernike moments of Fig. 5.1(b) with maximum order T from 100 to 700 and sub-region scheme value k from 1 to 11.

k\T	100	200	300	400	500	600	700
1	21.96	23.17	23.50	22.55	20.62	18.72	16.78
3	21.95	23.36	24.97	26.78	28.63	30.02	31.01
5	21.95	23.36	24.96	26.78	28.87	31.33	34.43
7	21.95	23.36	24.96	26.78	28.86	31.33	34.50
9	21.95	23.36	24.96	26.78	28.86	31.33	34.49
11	21.95	23.36	24.96	26.78	28.86	31.32	34.49

Image Reconstructions from Partial Sets of Zernike Moments

To verify the image information preserved by different sets of Zernike moments with various orders of n, we have reconstructed the image Fig. 5.1(a) from partial sets of the moments calculated with $k = 11$. Figure 5.6 shows images reconstructed from partial sets of Zernike moments with $n = 0$ to 50, $n = 51$ to 100, $n = 101$ to 200, $n = 201$ to 300, $n = 301$ to 500, and $n = 501$ to 700, respectively. To make the displayed images more visible, the gray values of Fig. 5.6(b) to Fig. 5.6(f) are multiplied by 10. Figure 5.6(a), the image reconstructed from the lower orders of moments, shows the principle segments of the original image. The images reconstructed from the higher orders of Zernike moments provide more image details.

We have added up the reconstructed six images shown in Fig. 5.6 to compose a new image, which is identical to the image shown at the bottom row and right column in Fig. 5.3. This indicates that the information provided by Zernike moments of each order is non-redundant.

Table 5.6: Zernike moments computation time (in seconds) for Fig. 5.1(a) with 64-bit precision. The computation times are longer than those in Table 5.1.

k\T	100	200	300	400	500	600	700
1	0.14	0.33	0.57	0.77	1.14	1.91	3.12
3	0.70	2.42	4.00	5.41	9.15	15.69	26.99
5	1.87	6.43	10.96	14.92	25.14	43.22	74.65
7	3.57	12.54	21.42	29.15	49.03	84.53	145.72
9	5.85	20.68	35.55	48.08	80.94	139.52	240.70
11	8.72	30.81	52.71	71.75	120.73	208.36	359.23

We have also performed image reconstructions from Zernike moments with zero, positive, or negative values of repetition m. Figure 5.7 demonstrates some reconstructed images of Fig. 5.1(a), with $n = 0$ to 700 and $k = 11$. From Fig. 5.7(a) to Fig. 5.7(f), the values of m are $0 < m \leq n$, $-n \leq m < 0$, $m = 0$, $0 \leq m \leq n$, $-n \leq m \leq 0$, and $-n \leq m \leq n$, respectively. With such a setting in this experiment, the moments used to reconstruct Fig. 5.7(d) is the collection of those for Figs. 5.7(a) and 5.7(c), and the same relationships for Figs. 5.7(e), 5.7(b) and 5.7(c). In addition, the moments involved to reconstruct Fig. 5.7(f) are composed of those in Figs. 5.7(a), 5.7(b) and 5.7(c). Our experimental results have shown that Figs. 5.7(a) and 5.7(b) are identical, and Figs. 5.7(d) and 5.7(e) are the exactly same image as well.

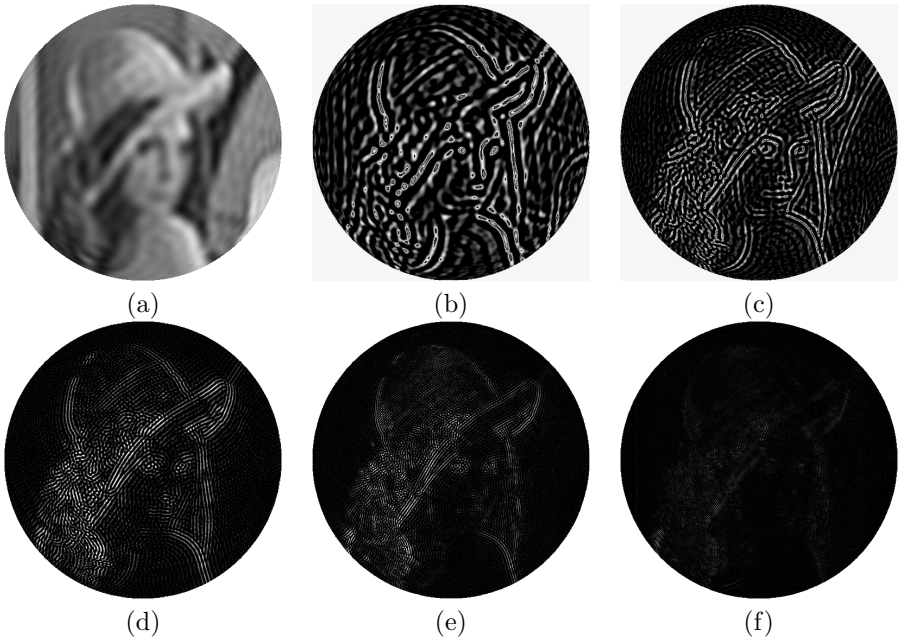


Figure 5.6: Reconstructed images of Fig. 5.1(a) from sets of partial Zernike moments with $k = 11$ and different order n. (a) $n = 0$ to 50. (b) $n = 51$ to 100. (c) $n = 101$ to 200. (d) $n = 201$ to 300. (e) $n = 301$ to 500. (f) $n = 501$ to 700. Image in (a) shows the principle segments of the original image. Images in other sub-figures provide more image details.

5.3 Experimental Results of Pseudo-Zernike Moments

We have also done all the above experiments on pseudo-Zernike moments.

Computational Efficiency

The running times for computing pseudo-Zernike moments of image Fig. 5.1(a) are shown in Table 5.7. Due to the same image size, the computation times

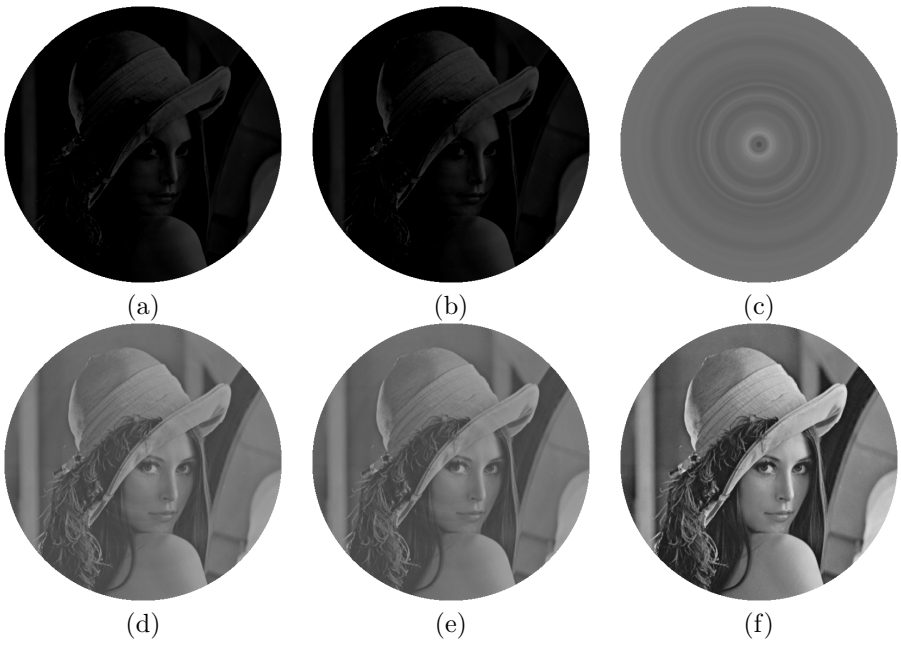


Figure 5.7: Reconstructed images of Fig. 5.1(a) from sets of partial Zernike moments with $n = 0$ to 700, $k = 11$ and different repetition m. (a) $0 < m \leq n$. (b) $-n \leq m < 0$. (c) $m = 0$. (d) $0 \leq m \leq n$. (e) $-n \leq m \leq 0$. (f) $-n \leq m \leq n$. Sub-figures (a) and (b) are identical. Sub-figures (d) and (e) are identical.

of pseudo-Zernike moments for image Fig. 5.1(b) are the same as those of Fig. 5.1(a).

As demonstrated in Table 5.7, without applying $k \times k$ sub-region scheme, it takes less than 3.06 seconds to calculate pseudo-Zernike moments with maximum order $T = 700$ of an image sized at 512×512. When the numerical sub-region scheme is applied, the computation time is about 343.6 seconds with $k = 11$.

From Table 5.1 and 5.7, the time used to compute pseudo-Zernike moments is approximately twice as much as that of Zernike moments. This is consistent with the statement in Section 2.3 that the number of polyno-

mials included in pseudo-Zernike moments is $(n+1)^2$ while that of Zernike moments is $\frac{1}{2}(n+1)(n+2)$.

Table 5.7: Pseudo-Zernike moments computation time (in seconds) for Fig. 5.1(a), with maximum order T from 100 to 700 and sub-region scheme value k from 1 to 11. The times increase when T or k increases.

k\T	100	200	300	400	500	600	700
1	0.15	0.41	0.56	0.82	1.30	1.97	3.06
3	0.94	3.19	4.64	6.17	10.75	16.60	25.77
5	2.48	8.65	12.70	17.00	29.65	45.85	71.15
7	4.77	16.98	24.73	33.20	57.94	89.73	139.26
9	7.85	27.93	40.81	54.84	95.74	148.14	230.00
11	11.61	41.71	61.14	81.90	142.95	221.19	343.60

The comparison result of pseudo-Zernike moments computation times between Deng's recursive method [10] and ours is shown in Table 5.8.

Table 5.8: Comparison of pseudo-Zernike moments computation times in CPU [10] and GPU. The speedup rate is the ratio of the time used in CPU divided by that in GPU.

Maximum Order T	100	200	300	400	500
CPU Results in Ref. [10]	0.39	1.44	3.10	5.43	8.39
Our GPU Results	0.15	0.41	0.56	0.82	1.30
Speedup Rate	2.6	3.5	5.5	6.6	6.5

Computational Accuracy of Image Reconstructions

Some reconstructed images from pseudo-Zernike moments of Fig. 5.1(a) are shown in Figs. 5.8 and 5.9. The PSNR values of the reconstructed images are shown in Table 5.9.

Similar to images reconstructed from Zernike moments, the images in Figs. 5.8 and 5.9 show that when the maximum pseudo-Zernike moment order T increases, more details of the original image will be reconstructed. When $k = 1$ and T goes higher, in addition to the distortions in the edge areas, the centre area becomes white. When k is greater than five, the distortion will visually be eliminated in the reconstructed images.

Table 5.9: PSNR values of images reconstructed from pseudo-Zernike moments of Fig. 5.1(a) with maximum order T from 100 to 700 and sub-region scheme value k from 1 to 11.

k\T	100	200	300	400	500	600	700
1	28.82	28.81	23.83	19.83	16.74	14.18	12.02
3	28.82	33.64	37.10	36.23	33.07	30.48	28.07
5	28.82	33.63	37.38	40.46	42.22	41.18	36.42
7	28.82	33.63	37.37	40.56	42.94	44.13	44.61
9	28.82	33.63	37.37	40.56	43.01	45.048	46.12
11	28.82	33.63	37.37	40.56	43.01	45.07	46.29

Some reconstructed images from pseudo-Zernike moments of Fig. 5.1(b) are presented in Figs. 5.10 and 5.11. Table 5.10 shows the PSNR values of the reconstructed images. These results demonstrate that the performances of image reconstructions from pseudo-Zernike moments improve in a similar way to those of Fig. 5.1(a), though the PSNR values are lower. This indicates that the image reconstruction performance of pseudo-Zernike moments is also affected by the contents of an image.

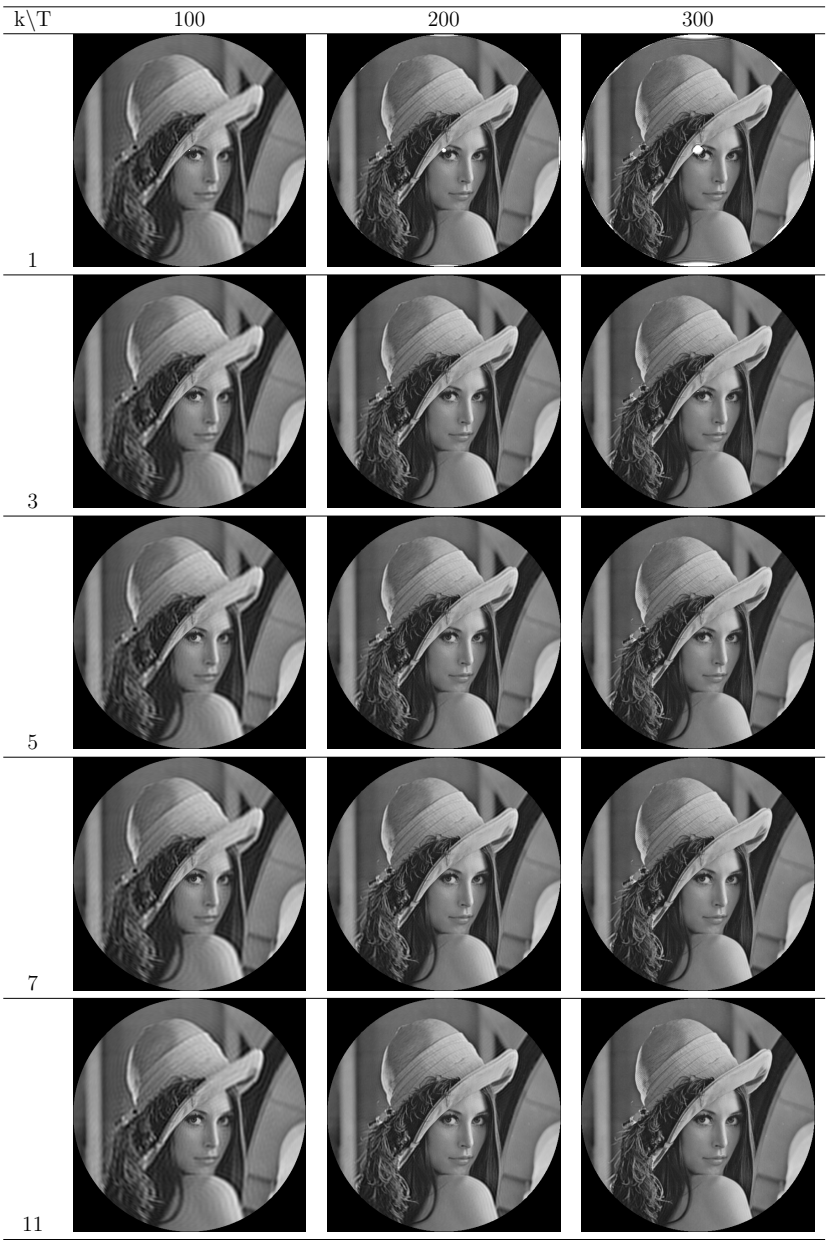


Figure 5.8: Images reconstructed from pseudo-Zernike moments of Fig. 5.1(a) with maximum order T from 100 to 300 and sub-region scheme value k from 1 to 11.

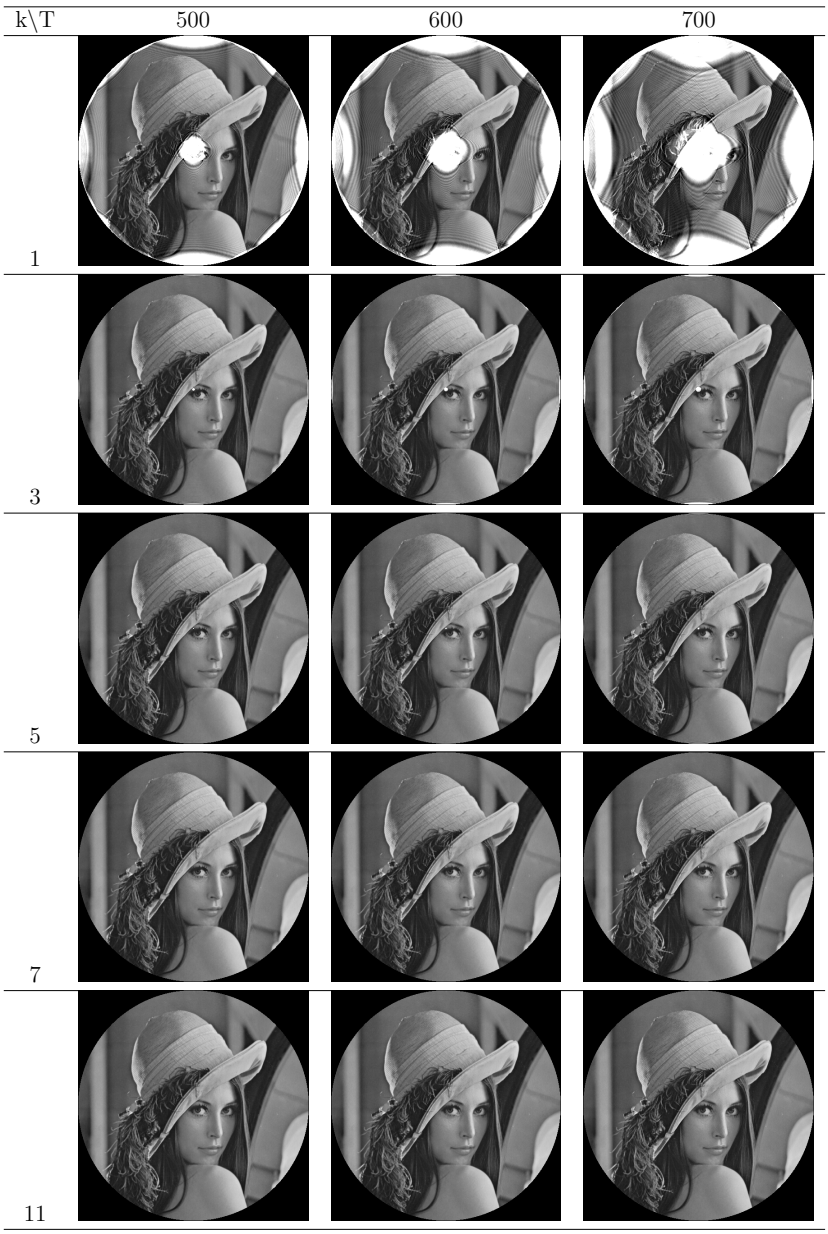


Figure 5.9: Images reconstructed from pseudo-Zernike moments of Fig. 5.1(a) with maximum order T from 400 to 700 and sub-region scheme value k from 1 to 11.

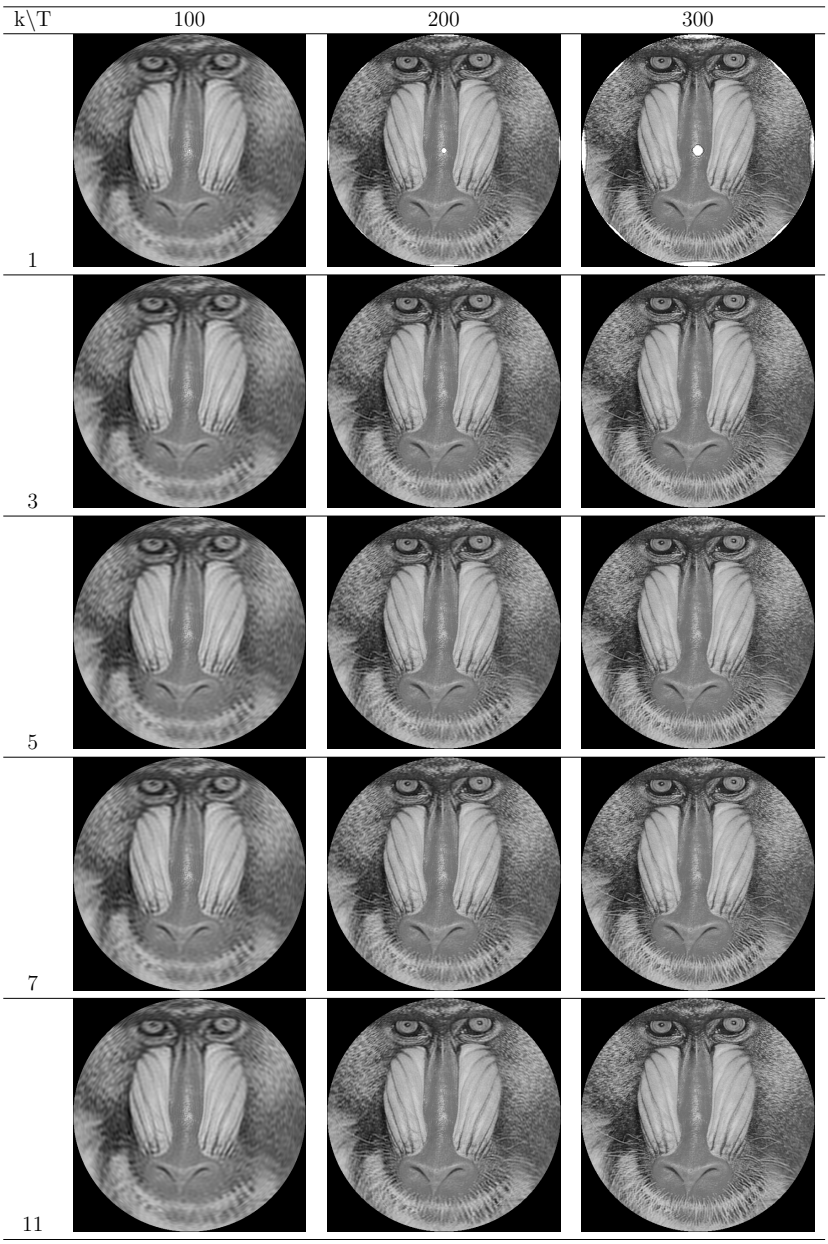


Figure 5.10: Images reconstructed from pseudo-Zernike moments of Fig. 5.1(b) with maximum order T from 100 to 300 and sub-region scheme value k from 1 to 11.

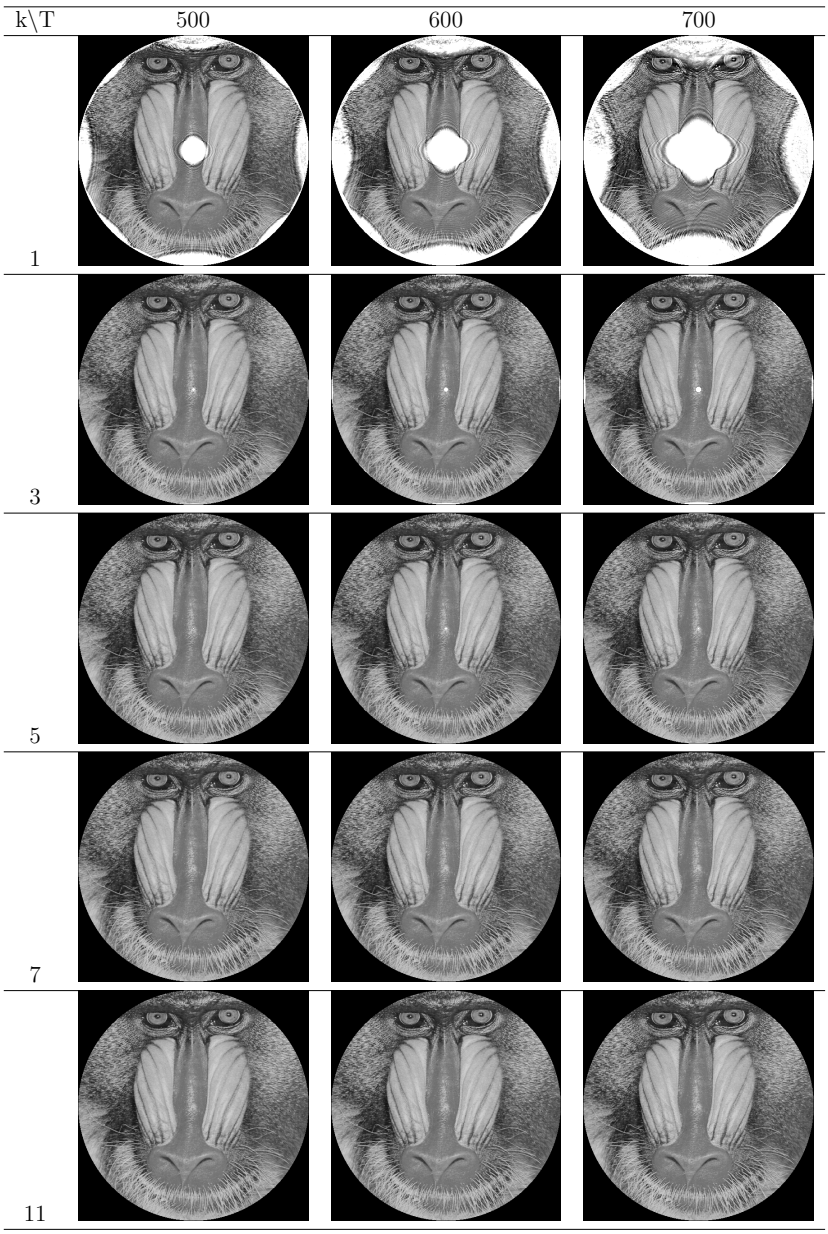


Figure 5.11: Images reconstructed from pseudo-Zernike moments of Fig. 5.1(b) with maximum order T from 400 to 700 and sub-region scheme value k from 1 to 11.

Table 5.10: PSNR values of images reconstructed from pseudo-Zernike moments of Fig. 5.1(b) with maximum order T from 100 to 700 and sub-region scheme value k from 1 to 11.

k\T	100	200	300	400	500	600	700
1	22.41	23.31	22.29	19.84	17.01	14.46	12.20
3	22.41	24.11	26.10	28.23	29.61	29.85	28.33
5	22.41	24.11	26.11	28.57	31.36	34.37	35.04
7	22.41	24.11	26.11	28.57	31.39	35.02	38.00
9	22.41	24.11	26.11	28.57	31.39	35.04	38.20
11	22.41	24.11	26.11	28.57	31.39	35.04	38.25

From Table 5.4 and 5.9, when $T = 700$ and $k = 11$, the PSNR value of image reconstructed from pseudo-Zernike moments of Fig. 5.1(a) is 46.29, higher than that of Zernike moments, which is 44.52. The same result is shown in Table 5.5 and 5.10. The reason is that the number of pseudo-Zernike moments used to reconstruct the image is almost twice as much as that of Zernike moments.

Image Reconstructions from Partial Sets of Pseudo-Zernike Moments

To verify the image information preserved by different sets of pseudo-Zernike moments with various orders of n, we have reconstructed the image Fig. 5.1(a) from partial sets of the moments calculated with $k = 11$. Figure 5.12 shows images reconstructed from partial sets of pseudo-Zernike moments $n = 0$ to 50, $n = 51$ to 100, $n = 101$ to 200, $n = 201$ to 300, $n = 301$ to 500, and $n = 501$ to 700, respectively. To make the displayed images more visible, the gray values of Fig. 5.12(b) to Fig. 5.12(f) are multiplied by 10. Figure 5.12(a),

the image reconstructed from the lower orders of moments, shows the principle segments of the original image. The images reconstructed from the higher orders of pseudo-Zernike moments provide more image details.

We have added up the reconstructed six images shown in Fig. 5.12 to compose a new image, which is identical to the image shown at the bottom row and right column in Fig. 5.9. This indicates that the information provided by pseudo-Zernike moments of each order is non-redundant.

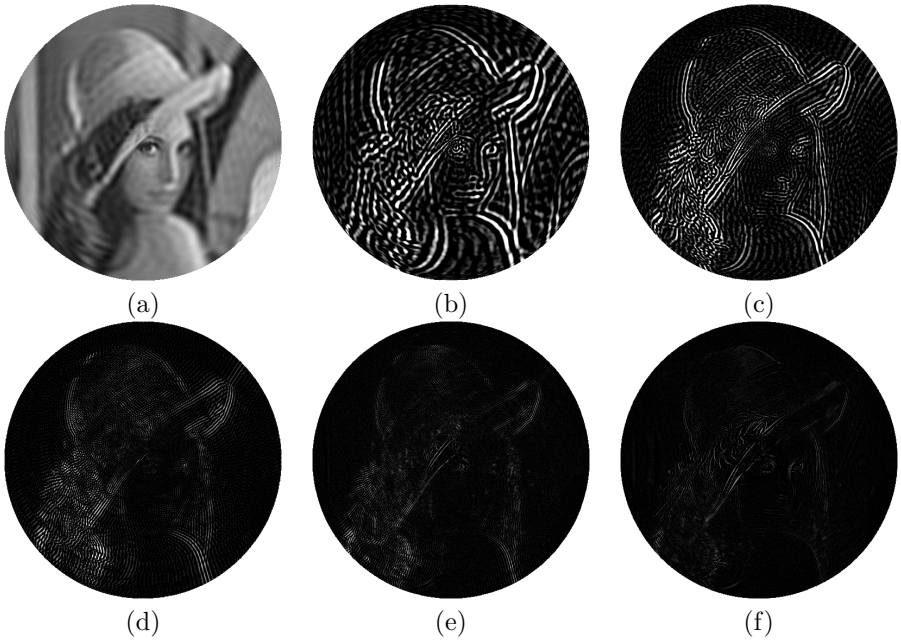


Figure 5.12: Reconstructed images of Fig. 5.1(a) from sets of partial pseudo-Zernike moments with different order n. (a) $n = 0$ to 50. (b) $n = 51$ to 100. (c) $n = 101$ to 200. (d) $n = 201$ to 300. (e) $n = 301$ to 500. (f) $n = 501$ to 700. Image in (a) shows the principle segments of the original image. Images in other sub-figures provide more image details.

We have also performed image reconstructions from pseudo-Zernike moments with zero, positive, or negative values of repetition m. Fig. 5.13 demonstrates some reconstructed images of Fig. 5.1(a), with $n = 0$ to 700 and

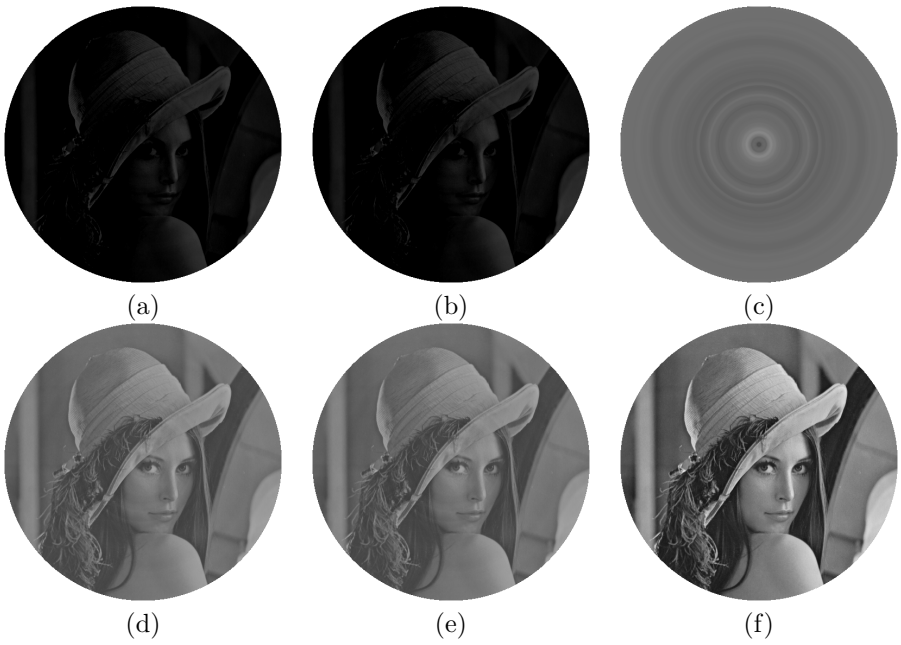


Figure 5.13: Reconstructed images of Fig. 5.1(a) from sets of partial pseudo-Zernike moments with $n = 0$ to 700, $k = 11$ and different repetition m. (a) $0 < m \leq n$. (b) $-n \leq m < 0$. (c) $m = 0$. (d) $0 \leq m \leq n$. (e) $-n \leq m \leq 0$. (f) $-n \leq m \leq n$. Sub-figures (a) and (b) are identical. Sub-figures (d) and (e) are identical.

$k = 11$. From Fig. 5.13(a) to Fig. 5.13(f), the values of m are $0 < m \leq n$, $-n \leq m < 0$, $m = 0$, $0 \leq m \leq n$, $-n \leq m \leq 0$, and $-n \leq m \leq n$, respectively. The experimental results are same with that of Zernike moments; Figs. 5.13(a) and 5.13(b) are identical, and Figs. 5.13(d) and 5.13(e) are the exactly same image as well.

5.4 Summary

In this chapter, we have demonstrated the efficiency of our GPU algorithm by computing the Zernike and pseudo-Zernike moments of an image sized

at 512×512 up to order 700. A larger k in the $k \times k$ sub-region scheme eliminates the distortions that appear in the image when the maximum order is high. This approves that the $k \times k$ sub-region scheme could reduce the approximation errors during the computation. Under the same settings, the computation time and the PSNR values for pseudo-Zernike moments are both higher than that of the Zernike moments, because a greater number of moments are included in the former. Compared to the 32-bit floating-point format, using the 64-bit precision in the computation increases the computation time, while the accuracy is not improved.

The images that reconstructed from different parts of the moments show the different information those moments preserve. The moments with lower orders provide information of the principle segments of the original image, while the moments with higher orders contain more image details. The moments whose repetition m are from 0 to n represent the identical information with those whose m are from 0 to $-n$.

Chapter 6

Chinese Character Recognition with Pseudo-Zernike Moments

In researches about Chinese character recognition, there are many feature extraction methods based on the local structures of Chinese characters, like strokes or feature points [31–34]. Many Chinese characters are similar in shape and differ only slightly, such as those shown in Fig. 6.1. Those characters with similar structures are hard to discriminate for structure-based recognition systems [35]. Since moments can capture the global features of images uniquely, they can be utilized as a complementary candidate to overcome this problem confronting those Chinese character recognition systems. By proposing moments-based feature vectors, we can recognize Chinese characters in the moment spaces and more efficiently to differentiate similar characters. In recent years, several types of moments have already been employed in Chinese character recognition systems [36–39].

Figure 6.1: Chinese character pairs with similar structures. It is hard for structure-based recognition systems to discriminate between them.

In this chapter, we will apply the feature vectors based on pseudo-Zernike moments to build the Chinese character recognition systems and analyze their performances. A set of 6,762 Chinese characters defined in the Chinese standard GB2312 is utilized as testing objects in this research, which are

composed of 24×24 binary pixels.

6.1 Chinese Character Recognition

The core of applying moments into Chinese character recognition is to extract moment feature vectors of Chinese characters. Different from real-valued moments, pseudo-Zernike moments are complex and have both real and imaginary parts. This enables us to utilize three types of values, the whole complex numbers, the real parts, and the magnitudes of the lower order pseudo-Zernike moments as feature vectors. We will use a set of 6,762 Chinese characters defined in China's national standard GB2312 as testing characters. Each Chinese character is a binary image with 24×24 pixels. The positions of '0's form the shape of the character, and the '1's form the background.

6.2 Pseudo-Zernike Moment Feature Vectors

Statistically, the moments with higher variances are more effective to differentiate objects than those of the lower ones [40]. To find the moments with higher variances, we have calculated the variances of the three types of values from the pseudo-Zernike moments with order $n \leq 5$ for all of the 6,762 Chinese characters. The results are shown in Table 6.1, Table 6.2, and Table 6.3, respectively. The four highest variances within each table are shown in shadow. Using the moment values with the highest variances, we have composed the following three pseudo-Zernike moment feature vectors

$$V_{complex}[f_1 = \hat{A}_{40}, f_2 = \hat{A}_{50}, f_3 = \hat{A}_{51}, f_4 = \hat{A}_{52}], \tag{6.1}$$

$$V_{real}[f_1 = Re(\hat{A}_{40}), f_2 = Re(\hat{A}_{50}), f_3 = Re(\hat{A}_{51}), f_4 = Re(\hat{A}_{52})], \tag{6.2}$$

and

$$V_{magnitude}[f_1 = |\hat{A}_{00}|, f_2 = |\hat{A}_{40}|, f_3 = |\hat{A}_{50}|, f_4 = |\hat{A}_{52}|]. \tag{6.3}$$

Table 6.1: Variance values of pseudo-Zernike moments $\hat{A}_{nm}$ with $n \leq 5$ calculated by complex number. The four highest variances are shown in shadow.

	$n=0$	$n=1$	$n=2$	$n=3$	$n=4$	$n=5$
$\|m\|=0$	0.0035	0.0015	0.0020	0.0034	0.0054	0.0054
$\|m\|=1$		0.0014	0.0017	0.0025	0.0040	0.0056
$\|m\|=2$			0.0027	0.0026	0.0044	0.0062
$\|m\|=3$				0.0026	0.0032	0.0048
$\|m\|=4$					0.0031	0.0042
$\|m\|=5$						0.0040

Table 6.2: Variance values of pseudo-Zernike moments $\hat{A}_{nm}$ with $n \leq 5$ calculated by their real parts. The four highest variances are shown in shadow.

	$n=0$	$n=1$	$n=2$	$n=3$	$n=4$	$n=5$
$\|m\|=0$	0.0035	0.0015	0.0020	0.0034	0.0054	0.0054
$\|m\|=1$		0.0008	0.0009	0.0016	0.0025	0.0038
$\|m\|=2$			0.0021	0.0017	0.0031	0.0048
$\|m\|=3$				0.0012	0.0018	0.0025
$\|m\|=4$					0.0019	0.0027
$\|m\|=5$						0.0019

Table 6.3: Variance values of pseudo-Zernike moments $\hat{A}_{nm}$ with $n \leq 5$ calculated by their magnitudes. The four highest variances are shown in shadow.

	$n=0$	$n=1$	$n=2$	$n=3$	$n=4$	$n=5$
$\|m\|=0$	0.0035	0.0013	0.0013	0.0015	0.0023	0.0029
$\|m\|=1$		0.0004	0.0004	0.0007	0.0010	0.0015
$\|m\|=2$			0.0007	0.0008	0.0011	0.0019
$\|m\|=3$				0.0006	0.0007	0.0012
$\|m\|=4$					0.0017	0.0017
$\|m\|=5$						0.0010

where $Re(\cdot)$ is the real part of a complex number and $|\cdot|$ is the magnitude of a complex number. Each kind of vectors formed a four-dimensional pseudo-Zernike moment feature space. Each Chinese character can be represented by a vector in the feature space.

We have used the Euclidean distance to measure the distance between any pair of two Chinese characters. For $V_{complex}$,

$$d(v_i, v_j) = \sqrt{|f_{1i} - f_{1j}|^2 + |f_{2i} - f_{2j}|^2 + |f_{3i} - f_{3j}|^2 + |f_{4i} - f_{4j}|^2}, \quad (6.4)$$

where v_i and v_j $(1 \leq i, j \leq 6762)$ are the two pseudo-Zernike moment feature vectors representing two Chinese characters, and $|f_{ki} - f_{kj}|$ is the magnitude of the difference between two complex numbers f_{ki} and f_{kj}. For V_{real} and $V_{magnitude}$,

$$d(v_i, v_j) = \sqrt{(f_{1i} - f_{1j})^2 + (f_{2i} - f_{2j})^2 + (f_{3i} - f_{3j})^2 + (f_{4i} - f_{4j})^2}. \quad (6.5)$$

With the total of 6,762 Chinese characters in the data set, we will obtain $(6762 \times 6761)/2 = 22,858,941$ distances in each of feature vector spaces.

Table 6.4: Statistics of distances between Chinese character pairs with three pseudo-Zernike moments feature vectors and the best Zernike moments feature vector [37]. A lower number of Chinese character pairs in the rows of low distance indicates a better performance.

$d_(v_i, v_j)$	$V_{complex}$	V_{real}	$V_{magnitude}$	$V_{complex}$ (Zernike)
0–0.01	2	382	1,866	1
0.01–0.02	160	5,618	26,502	103
0.02–0.03	1,458	23,291	103,019	1,431
0.03–0.04	6,656	60,438	244,556	9,356
0.04–0.05	20,698	119,687	443,697	34,357
0.05–0.06	49,165	202,445	680,139	93,979
0.06–0.07	99,181	308,695	933,184	204,098
0.07–0.08	175,069	420,358	1,172,664	376,078
0.08–0.09	274,844	566,927	1,384,329	602,301
0.09–0.10	401,117	707,207	1,544,734	869,781
Above 0.10	21,830,531	20,433,893	16,324,251	20,667,456
Average Distance	0.2018	0.1841	0.1350	0.1633

6.3 Experimental Results

To analyze the recognition power of the proposed pseudo-Zernike moment feature vectors, Table 6.4 shows the statistic of distances in different value intervals. As a comparison, the best results of applying Zernike moments feature vector in a previous research [37] are also represented in Table 6.4.

Comparing the second to fourth columns in Table 6.4, the amount of distances based on feature vectors $V_{complex}$ in each low-value intervals is less than

Table 6.5: The closest ten pairs of Chinese characters recognized by $V_{complex}$. The two Chinese characters in each pair are quite different in shapes, components, and the number of strokes.

	差露	碍礓	唷溼	亮拿	片移
$d(v_i, v_j)$	0.0067	0.0095	0.0101	0.0102	0.0114
	恰愉	痕吉	朱怃	工十	歌鷹
$d(v_i, v_j)$	0.0114	0.0115	0.0116	0.0118	0.0120

those of V_{real} and $V_{magnitude}$. This indicates that $V_{complex}$ is more powerful in recognizing Chinese characters.

We also compared our data to the data generated by Zernike moments with the same process. The most powerful feature vector in Ref. [37]'s experiment is also $V_{complex}$, which is listed in the fifth column of Table 6.4. Comparing the values of two $V_{complex}$s, when distances are less than 0.03, the numbers are very close. When distances are between 0.03 and 0.10, numbers from pseudo-Zernike moments are less than numbers of Zernike moments. The number of distances above 0.10 and the average distance of $V_{complex}$ from pseudo-Zernike moments are both larger than those of Zernike moments. All of the above observations demonstrate that, in terms of recognition power, the pseudo-Zernike moments are more powerful than Zernike moments with the same order n. The better performance of pseudo-Zernike moments may rely on the process of extracting feature vectors. Compared to Zernike moments, pseudo-Zernike provides more moments to select from, and the selected moments have larger variances than Zernike moments.

We pick out the closest ten pairs of Chinese characters recognized by $V_{complex}$, V_{real}, and $V_{magnitude}$, and show them in Table 6.5, Table 6.6, and Table 6.7, respectively.

Each pair of Chinese characters in the three tables are close in the pseudo-

Table 6.6: The closest ten pairs of Chinese characters recognized by V_{real}. The two Chinese characters in each pair are quite different in shapes, components, and the number of strokes.

	奸妊	钙脲	钡稼	缝钛	窘焘
$d(v_i, v_j)$	0.0019	0.0034	0.0036	0.0036	0.0039
	它阌	舛睨	亮拿	潭亳	靖平
$d(v_i, v_j)$	0.0042	0.0042	0.0043	0.0043	0.0044

Table 6.7: The closest ten pairs of Chinese characters recognized by $V_{magnitude}$. The two Chinese characters in each pair are quite different in shapes, components, and the number of strokes.

	隧剃	掂薏	弹箢	绰肴	炯茴
$d(v_i, v_j)$	0.0011	0.0015	0.0018	0.0018	0.0019
	财净	洁浴	邺麽	栽颉	呱觇
$d(v_i, v_j)$	0.0020	0.0020	0.0023	0.0024	0.0026

Zernike moment feature vector spaces. However, they are quite different in shapes, components, and the number of strokes. Although there are difficulties to recognize them by pseudo-Zernike moment features, these Chinese character pairs can be easily recognized by recognition systems based on structural features.

To further display the performance of pseudo-Zernike moment feature vectors on recognizing Chinese characters, we list the distances of several Chinese character pairs with similar shapes in the three different pseudo-Zernike moment feature spaces. These pairs of Chinese characters are not easy to recognize for Chinese character recognition systems that utilizing structure features [37]. From Table 6.8, we can see pseudo-Zernike moment features perform well in recognizing the two characters in each pair.

Based on the above performance, pseudo-Zernike moment feature vectors could be a good complement to some existing structure feature-based Chinese recognition systems.

Referring to Eq. (2.9), the pseudo-Zernike moments are rotational invariant. To verify this property, we have applied the proposed three pseudo-Zernike moment feature vectors to all Chinese characters which are rotated by 90°, 180°, and 270°, respectively. Our experiment results show that the magnitudes of moments are invariant, and the distances based on $V_{magnitude}$ are identical to the original ones.

6.4 Summary

In this chapter, the pseudo-Zernike moments feature vectors for Chinese characters recognition are investigated. We choose a set of four lower order pseudo-Zernike moments with highest variance values from different parts of the moments, the whole complex moments, the real parts of the moments, and the magnitudes of the moments to build three pseudo-Zernike moments

Table 6.8: Distances of some Chinese character pairs which are very close in shapes. The values of these distances are relatively high, meaning the pseudo-Zernike moment features could perform well in recognizing the two characters in each pair.

	搀馋	抄钞	芳坊	赢嬴	日曰
Complex plane	0.0666	0.0644	0.1714	0.1747	0.1545
Real part	0.0662	0.0570	0.1476	0.1715	0.1542
Magnitude	0.0629	0.0341	0.0913	0.1362	0.1320
	荻荻	朴扑	鞯鞴	末木	己已
Complex plane	0.0857	0.1495	0.0587	0.1322	0.1348
Real part	0.0772	0.1419	0.0379	0.1135	0.1166
Magnitude	0.0627	0.0890	0.0339	0.1128	0.1116
	主柱	壕嚎	诅阻	末未	夭毛
Complex plane	0.1598	0.1582	0.2806	0.0598	0.0963
Real part	0.1480	0.1576	0.2782	0.0596	0.0837
Magnitude	0.1348	0.1267	0.1443	0.0464	0.0780
	奖桨	垃拉	伦抡	沦论	大犬
Complex plane	0.0701	0.0851	0.0586	0.0950	0.0817
Real part	0.0687	0.0714	0.0396	0.0918	0.0726
Magnitude	0.0526	0.0284	0.0483	0.0734	0.0386
	搏博	俊浚	毫豪	竟竞	辩辨
Complex plane	0.1589	0.1423	0.1099	0.1275	0.1388
Real part	0.1560	0.1414	0.1082	0.1255	0.1385
Magnitude	0.0614	0.1199	0.0922	0.0727	0.1134

based feature vectors.

Our experiment results with a set of 6,762 Chinese characters show that the vectors generated by the whole complex numbers of pseudo-Zernike moments are the most powerful in distinguishing Chinese characters. Compared to Zernike moments, the pseudo-Zernike moments provide more moments to choose from. Therefore, the performance of pseudo-Zernike moments based feature vectors is also superior to those of Zernike moments.

The Chinese character pairs that are close in pseudo-Zernike moment feature vector spaces are not similar in structure, and can be recognized by recognition systems based on local structure features. On the other hand, the Chinese characters with similar structures are difficult to recognize for local structure-based systems, while our pseudo-Zernike moments method can differentiate them efficiently. In conclusion, the method we proposed could be a satisfying complementary scheme to other existing Chinese character recognition systems.

Chapter 7

Concluding Remarks

In this research, we have proposed and implemented a GPU-accelerated algorithm to compute Zernike and pseudo-Zernike moments of images. To improve the computational accuracy of the two kinds of moments, this algorithm employs the recursive relations of the radial polynomials for Zernike and pseudo-Zernike moments. In addition, we have applied the numerical $k \times k$ sub-region scheme to optimize double integrals and reduce the approximation errors. To use the full capability of GPU, we have adopted the symmetric methods, partitioned the pixels dynamically, managed GPU memory carefully, and used the shared memory in the process of parallel reduction.

To verify the efficiency of our GPU accelerating computation algorithm, we have calculated the Zernike and pseudo-Zernike moments of two testing images sized at 512×512 with high orders. The experimental results show that our approach can provide efficient computation performance to calculate the moments. Without applying the numerical $k \times k$ sub-region scheme, for an image sized at 512×512, it takes 1.7 seconds to calculate the Zernike moments up to order 700, and 3.1 seconds to calculate the pseudo-Zernike moments to the same order. When the sub-region scheme is adopted and $k = 11$, the running time for Zernike moments is about 181.1 seconds, and that of pseudo-Zernike moments is 343.6 seconds.

To investigate the accuracy of our GPU algorithm, we have calculated the PSNR values of the reconstructed images from Zernike and pseudo-Zernike moments with various maximum orders and different numerical $k \times k$ sub-region schemes. The experimental results show that the higher orders of moments in image reconstructions may introduce distortion, which can be

reduced significantly by increasing the numerical scheme k. For an image sized at 512×512, with the maximum order of 700 and $k = 11$, the PSNR values of its reconstructed versions from Zernike and pseudo-Zernike moments are 44.52 and 46.29 separately.

In this research, we have performed image reconstructions from partial sets of Zernike and pseudo-Zernike moments with various order n and different repetition m. The experimental results of the two kinds of moments are similar. We have concluded that the images reconstructed from the lower orders of moments preserve the principle contents of the original image, while the higher orders of moments contain more image information on details. We have also observed that the positive and negative repetition m will result in the identical reconstructed images.

We have also investigated the application of pseudo-Zernike moments in Chinese characters recognition. We choose a set of four lower order pseudo-Zernike moments with the highest variance values from different parts of the moments and build three feature vectors based on pseudo-Zernike. A set of 6,762 Chinese characters defined in the Chinese standard GB2312 is utilized in this research to test our proposed pseudo-Zernike moments feature vectors. Our experimental results show that the three different feature vectors can provide satisfied recognition performances independently, and the vectors generated by the whole complex numbers of pseudo-Zernike moments are the most powerful in distinguishing Chinese characters.

Appendix A

Source Code for the GPU Kernels

The programs of Zernike moments and pseudo-Zernike moments computation share the same GPU kernels of *Origin_Kernel*, *Rho_Kernel*, *Exp_Kernel*, *Reorder_Kernel*, and *Znm_Sum_Kernel*. Their *Znm_Kernel*, *Vnm_Kernel*, and *Fxy_Kernel* are similar. In the following we will give the source code for the GPU kernels of Zernike moments computation.

Source code A.1: Origin_Kernel

```
1  __global__ void origin_kernel(float *rho, float *theta,
2   float *mask, float *step, int *width)
3  {
4      const int tx = threadIdx.x;
5      const int ty = threadIdx.y;
6      const int bw = blockDim.x;
7      const int bh = blockDim.y;
8      const int bx = blockIdx.x;
9      const int by = blockIdx.y;
10
11     const int local_width = *width;
12     const float local_step = *step;
13     int half_width = local_width / 2;
14
15     int row = by * bh + ty;
16     int col = bx * bw + tx;
17     int pos = row * local_width + col;
18     // Calculate rho and theta of pixels
19     if(row < local_width && col < local_width){
20         float x = -1 + 1/float(local_width) +
```

```
21            local_step * float(col);
22          float y = 1 - 1/float(local_width)
23           - local_step * float(row);
24          rho[pos] = sqrt(powf(x, 2) + powf(y, 2));
25          theta[pos] = atan2(y, x);
26
27      // Calculate mask of the left upper quarter of
28      // the original image
29
30          if(row < half_width && col < half_width){
31              int left_corner_pos = row * half_width + col;
32              float left_corner_x = -1 + local_step * col;
33              float left_corner_y = 1 - local_step * row;
34              float left_corner_rho =
35               sqrt(powf(left_corner_x, 2)
36               + powf(left_corner_y, 2));
37              if(left_corner_rho <= 1){
38                  mask[left_corner_pos] = 1;
39              }
40          }
41      }
```

Source code A.2: Exp_Kernel

```
1  # include <pycuda-complex.hpp>
2  typedef pycuda::complex<float> scmplx;
3  __global__ void exp_kernel(scmplx *ar_exp,
4   float *ar_arctan, int *m, int *current_pixels)
5  {
6      const int tx = threadIdx.x;
7      const int ty = threadIdx.y;
8      const int bw = blockDim.x;
```

```
9       const int bh = blockDim.y;
10      const int bx = blockIdx.x;
11      int pos = tx + ty*bw + bx*bw*bh;
12      if(pos < (*m) * (*current_pixels)){
13          int idx_origin_y = pos / (*current_pixels);
14          int idx_origin_x = pos % (*current_pixels);
15          ar_exp[pos] = exp(-scmplx(0,
16          ar_arctan[idx_origin_x]*(float)(idx_origin_y)));
17      }
18  }
```

Source code A.3: Rho_Kernel

```
1   __global__ void rho_kernel(float *ar_rho,
2    float *rho_filtered, int *m, int *current_pixels)
3   {
4       const int tx = threadIdx.x;
5       const int ty = threadIdx.y;
6       const int bw = blockDim.x;
7       const int bh = blockDim.y;
8       const int bx = blockIdx.x;
9       int pos = tx + ty*bw + bx*bw*bh;
10      if(pos < (*m) * (*current_pixels)){
11          int idx_origin_y = pos / (*current_pixels);
12          int idx_origin_x = pos % (*current_pixels);
13          ar_rho[pos] =
14          pow(rho_filtered[idx_origin_x], idx_origin_y);
15      }
16  }
```

Source code A.4: Reorder_Kernel

```
1   #include <pycuda-complex.hpp>
```

```
2  __global__ void reorder_kernel(
3      float *img_out,
4      float *img_in,
5      int *octant_coord,
6      int *current_pixels,
7      int *current_four_count,
8      int *current_eight_count,
9      int *width_k,
10     int *k
11 )
12 {
13     const int bx = blockIdx.x;
14     const int bw = blockDim.x;
15     const int tx = threadIdx.x;
16     const int ty = threadIdx.y;
17
18     const int current_pixels_g =
19      *current_pixels;
20     const int current_four_count_g =
21      *current_four_count;
22     const int current_eight_count_g =
23      *current_eight_count;
24     const int width_k_g = *width_k;
25     const int k_g = *k;
26     const int N = width_k_g / k_g;
27
28     const int pos = bw * bw * bx + ty * bw + tx;
29
30     if(pos < current_pixels_g){
31         int row_k = octant_coord[2 * pos];
32         int col_k = octant_coord[2 * pos + 1];
```

```
33          int row = row_k / k_g;
34          int col = col_k / k_g;
35          int minus_row = N - 1 - row;
36          int minus_col = N - 1 - col;
37
38          if(pos < current_eight_count_g){
39
40              img_out[pos] = img_in[row * N + col];
41              img_out[pos + current_eight_count_g]
42               = img_in[col * N + row];
43              img_out[pos + 2 * current_eight_count_g]
44               = img_in[row * N + minus_col];
45              img_out[pos + 3 * current_eight_count_g]
46               = img_in[col * N + minus_row];
47              img_out[pos + 4 * current_eight_count_g]
48               = img_in[minus_row * N + col];
49              img_out[pos + 5 * current_eight_count_g]
50               = img_in[minus_col * N + row];
51              img_out[pos + 6 * current_eight_count_g]
52               = img_in[minus_row * N + minus_col];
53              img_out[pos + 7 * current_eight_count_g]
54               = img_in[minus_col * N + minus_row];
55
56          }else{
57              int pos_base =
58               current_eight_count_g * 7 + pos;
59              img_out[pos_base] = img_in[row * N + col];
60              img_out[pos_base + current_four_count_g]
61               = img_in[row * N + minus_col];
62              img_out[pos_base + 2 * current_four_count_g]
63               = img_in[minus_row * N + col];
```

```
64                 img_out[pos_base + 3 * current_four_count_g]
65                  = img_in[minus_row * N + minus_col];
66             }
67         }
68 }
```

Source code A.5: Znm_Kernel

```
1  # include <pycuda-complex.hpp>
2  // 'order' is the maximum order of moments taken into
3  // account, noted as T in the text, and passed by the
4  // host code.
5  # define local_order """ + order + """
6  typedef pycuda::complex<float> scmplx;
7  __device__ scmplx get_total_tmp(int sig, int num,
8   float *loaded_pixels, scmplx exp_n);
9  __global__ void znm_kernel(
10     scmplx *ar_znm,
11     float *ar_img_k,
12     float *ar_rho_k,
13     scmplx *ar_exp_k,
14     int *current_pixels,
15     int *current_four_count,
16     int *current_eight_count)
17 {
18     const int tx = threadIdx.x;
19     const int ty = threadIdx.y;
20     const int bw = blockDim.x;
21     const int bh = blockDim.y;
22     const int bx = blockIdx.x;
23
24     const int current_pixels_local =
```

```
25          *current_pixels;
26      const int current_eight_count_local =
27          *current_eight_count;
28      const int current_four_count_local =
29          *current_four_count;
30
31      //Calculate the total number of moments
32      const int znm_count = (int(local_order / 2) + 1)
33          * (int(local_order / 2) + 1 + local_order % 2);
34      float loaded_pixels[8];
35      int pos = tx + ty*bw + bx*bw*bh;
36      float rho = ar_rho_k[current_pixels_local + pos];
37
38      if(pos < current_pixels_local){
39
40          float R[local_order + 1];
41          float Rp[local_order + 1];
42          float Rpp[local_order + 1];
43          scmplx total_tmp = scmplx(0,0);
44          scmplx pixel_sum;
45          int znm_pos = 0;
46
47          //Process pixels that are eight-symmetric
48          if(pos < current_eight_count_local){
49
50              // Load the value of pixels
51              for(int i = 0; i < 8; i++){
52                  loaded_pixels[i] = ar_img_k[
53                  i * current_eight_count_local + pos];
54                  pixel_sum += loaded_pixels[i];
55              }
```

```
56
57              ar_znm[pos * znm_count] = pixel_sum;
58
59              Rpp[0] = 1;
60              Rp[1] = rho;
61
62              // Calculate znm by the recursive method
63              for(int n = 1; n <= local_order; n++){
64
65                  scmplx exp_n =
66                  ar_exp_k[n * current_pixels_local + pos];
67                  float exp_n_real = exp_n.real();
68                  float exp_n_imag = exp_n.imag();
69
70                  if (n == 1){
71                      total_tmp = scmplx(
72                      (loaded_pixels[0] - loaded_pixels[6]
73                       - loaded_pixels[2] + loaded_pixels[4])
74                       * exp_n_real
75                       + (loaded_pixels[1] - loaded_pixels[7]
76                       - loaded_pixels[3] + loaded_pixels[5])
77                       * exp_n_imag,
78
79                      (loaded_pixels[0] - loaded_pixels[6]
80                       + loaded_pixels[2] - loaded_pixels[4])
81                       * exp_n_imag
82                       + (loaded_pixels[1] - loaded_pixels[7]
83                       + loaded_pixels[3] - loaded_pixels[5])
84                       * exp_n_real
85                      );
86
```

```
87                      ar_znm[pos * znm_count + 1] =
88                       total_tmp * rho;
89                      continue;
90                  }
91
92                  total_tmp =
93                   get_total_tmp(0, n, loaded_pixels, exp_n);
94                  R[n] =
95                   ar_rho_k[n * current_pixels_local + pos];
96                  znm_pos = (int(n / 2) + 1) *
97                   (int(n / 2) + 1 + n % 2) - 1;
98                  ar_znm[pos * znm_count + znm_pos] =
99                  total_tmp * R[n];
100
101                 if(n % 2 == 0){
102                     R[0] = 2 * rho * Rp[1] - Rpp[0];
103                     ar_znm[pos * znm_count
104                      + znm_pos - n / 2]
105                      = pixel_sum * R[0];
106                 }
107
108                 total_tmp = scmplx(0,0);
109
110                 for(int m = n - 2; m > 0; m = m - 2 ){
111
112                     exp_n = ar_exp_k[
113                     m * current_pixels_local + pos];
114                     exp_n_real = exp_n.real();
115                     exp_n_imag = exp_n.imag();
116
117                     total_tmp = get_total_tmp(0, m,
```

```
118                         loaded_pixels, exp_n);
119                     R[m] =
120                         rho * (Rp[m-1] + Rp[m+1]) - Rpp[m];
121                     znm_pos = (int((n - 1) / 2) + 1) *
122                         (int((n - 1) / 2) + 1 + (n - 1) % 2)
123                         - 1 + int((m + 2) / 2);
124                     ar_znm[pos * znm_count + znm_pos] =
125                         total_tmp * R[m];
126                     total_tmp = scmplx(0,0);
127                 }
128
129                 for (int i = 0; i <= n; i++){
130                     Rpp[i] = Rp[i];
131                     Rp[i] = R[i];
132                 }
133
134             }
135
136         }else{
137             //Process pixels that are four-symmetric
138             for(int i = 0; i < 4; i++){
139                 loaded_pixels[i] =
140                     ar_img_k[current_eight_count_local * 7 +
141                         i * current_four_count_local + pos];
142                 pixel_sum += loaded_pixels[i];
143             }
144             ar_znm[pos * znm_count] = pixel_sum;
145             total_tmp = scmplx(0,0);
146
147             Rpp[0] = 1;
148             Rp[1] = rho;
```

```
149
150             for(int n = 1; n <= local_order; n++){
151                 scmplx exp_n =
152                  ar_exp_k[n * current_pixels_local + pos];
153                 float exp_n_real = exp_n.real();
154                 float exp_n_imag = exp_n.imag();
155
156                 if (n == 1){
157                     total_tmp =
158                      loaded_pixels[0] * exp_n +
159                       loaded_pixels[1]
160                      * scmplx(-exp_n_imag, exp_n_real) +
161                      loaded_pixels[2] * pycuda::conj(exp_n)
162                      + loaded_pixels[3] *
163                      scmplx(-exp_n_imag, -exp_n_real);
164                     ar_znm[pos * znm_count + 1] =
165                      total_tmp * rho;
166                     continue;
167                 }
168
169                 total_tmp = get_total_tmp(1, n,
170                  loaded_pixels, exp_n);
171                 R[n] =
172                 ar_rho_k[n * current_pixels_local + pos];
173                 znm_pos = (int(n / 2) + 1) *
174                  (int(n / 2) + 1 + n % 2) - 1;
175                 ar_znm[pos * znm_count + znm_pos] =
176                  total_tmp * R[n];
177
178                 if(n % 2 == 0){
179                     R[0] = 2 * rho * Rp[1] - Rpp[0];
```

```
180                         ar_znm[pos * znm_count
181                          + znm_pos - n / 2]
182                          = pixel_sum * R[0];
183                     }
184
185                     total_tmp = scmplx(0,0);
186
187                     for(int m = n - 2; m > 0; m = m - 2){
188                         exp_n = ar_exp_k[
189                         m * current_pixels_local + pos];
190
191                         total_tmp = get_total_tmp(1, m,
192                          loaded_pixels, exp_n);
193                         R[m] =
194                          rho * (Rp[m-1] + Rp[m+1]) - Rpp[m];
195                         znm_pos = (int((n - 1) / 2) + 1) *
196                          (int((n - 1) / 2) + 1 + (n - 1) % 2)
197                          - 1 + int((m + 2) / 2);
198                         ar_znm[pos * znm_count + znm_pos] =
199                          total_tmp * R[m];
200                         total_tmp = scmplx(0,0);
201                     }
202                     for (int i = 0; i <= n; i++){
203                         Rpp[i] = Rp[i];
204                         Rp[i] = R[i];
205                     }
206                 }
207
208             }
209
210         }
```

```
211 }
212 __device__ scmplx get_total_tmp(int sig, int num,
213  float *loaded_pixels, scmplx exp_n){
214
215     scmplx total_tmp = scmplx(0,0);
216     float exp_n_real = exp_n.real();
217     float exp_n_imag = exp_n.imag();
218
219     //Make use of symmetric properties
220     if(sig == 0){
221         if(num % 4 == 0){
222             total_tmp = (loaded_pixels[0] + loaded_pixels[6]
223              + loaded_pixels[3] + loaded_pixels[5]) * exp_n
224              + (loaded_pixels[1] + loaded_pixels[2]
225              + loaded_pixels[4] + loaded_pixels[7])
226              * pycuda::conj(exp_n);
227         }else if(num % 4 == 1){
228             total_tmp = (loaded_pixels[0] - loaded_pixels[6])
229             * exp_n + (loaded_pixels[1] - loaded_pixels[7])
230             * scmplx(exp_n_imag, exp_n_real)
231             + (loaded_pixels[4] - loaded_pixels[2])
232             * pycuda::conj(exp_n)
233             + (loaded_pixels[3] - loaded_pixels[5])
234             * scmplx(-exp_n_imag, exp_n_real);
235         }else if(num % 4 == 2){
236             total_tmp = (loaded_pixels[0] + loaded_pixels[6]
237             - loaded_pixels[3] - loaded_pixels[5]) * exp_n +
238             (loaded_pixels[2] + loaded_pixels[4]
239             - loaded_pixels[1] - loaded_pixels[7])
240             * pycuda::conj(exp_n);
241         }else{
```

```
242             total_tmp = (loaded_pixels[0] - loaded_pixels[6])
243             * exp_n - (loaded_pixels[2] - loaded_pixels[4])
244             * pycuda::conj(exp_n)
245             - (loaded_pixels[3] - loaded_pixels[5])
246             * scmplx(-exp_n_imag, exp_n_real)
247             - (loaded_pixels[1] - loaded_pixels[7])
248             * scmplx(exp_n_imag, exp_n_real);
249         }
250     }else{
251
252         if(num % 4 == 0){
253             total_tmp = (loaded_pixels[0] + loaded_pixels[1]
254             + loaded_pixels[2] + loaded_pixels[3]) * exp_n;
255         }else if(num % 4 == 1){
256             total_tmp = loaded_pixels[0] * exp_n +
257             loaded_pixels[1] *
258             scmplx(-exp_n_imag, exp_n_real)
259             + loaded_pixels[2] * pycuda::conj(exp_n)
260             + loaded_pixels[3] *
261             scmplx(-exp_n_imag, -exp_n_real);
262         }else if(num % 4 == 2){
263             total_tmp = (loaded_pixels[0] - loaded_pixels[1])
264             * exp_n + (loaded_pixels[2] - loaded_pixels[3])
265             * pycuda::conj(exp_n);
266         }else{
267             total_tmp = loaded_pixels[0] * exp_n -
268             loaded_pixels[1] *
269             scmplx(-exp_n_imag, exp_n_real)
270             + loaded_pixels[2] *
271             pycuda::conj(exp_n) - loaded_pixels[3] *
272             scmplx(-exp_n_imag, -exp_n_real);
```

```
273             }
274         }
275         return total_tmp;
276 }
```

Source code A.6: Znm_Sum_Kernel

```
1  /* The input is a matrix with size of number of
2   * current_pixels by number of moments. We need to
3   * calculate the sum of each column, i.e. adding up
4   * the znm of all pixels under one specific moment.
5   * Each moment is calculated by one block and each
6   * thread in this block may load and add up more
7   * than one values. The size of the shared memory
8   * is equal to the number of threads in a block.
9   * It is used for reduction. For example, when one
10  * block has 512 threads, the size of shared memory
11  * is 512*datasize. Assuming there are 3380 pixels,
12  * and 3080 ÷ 512 = 6.02. Then each thread loads and
13  * adds up 7 values, and 440 threads are truely used.
14  */
15
16 # include <pycuda-complex.hpp>
17
18 typedef pycuda::complex<float> scmplx;
19 __global__ void znm_sum_kernel(
20     scmplx *ar_zsum,
21     scmplx *ar_znm,
22     int *znm_count,
23     int *current_pixels)
24 {
25     const int tx = threadIdx.x;
```

```
26      const int bx = blockIdx.x;
27      const int bw = blockDim.x;
28      extern __shared__ scmplx s_data[];
29      // Calculate the number of values each thread loads.
30      int data_per_pos = ((*current_pixels) - 1) / bw + 1;
31      int pos_in_block = 0;
32      s_data[tx] = 0;
33
34      if(bx < (*znm_count)){
35          //Load and add up values and store to shared memory
36          for (int i = 0; i < data_per_pos; i++){
37              pos_in_block = i * bw + tx;
38              if ( pos_in_block < (*current_pixels)){
39                  s_data[tx] +=
40                   ar_znm[pos_in_block * (*znm_count) + bx];
41              }
42
43          }
44
45          __syncthreads();
46          //Reduction
47          for(int i = blockDim.x / 2; i > 0; i /= 2){
48              if(tx < i){
49                  s_data[tx] = s_data[tx] + s_data[tx + i];
50              }
51              __syncthreads();
52          }
53
54          if(tx == 0){
55              ar_zsum[bx] = s_data[0];
56          }
```

```
57
58      }
59
60  }
```

Source code A.7: Vnm_Kernel

```
1  # include <pycuda-complex.hpp>
2  # define local_order """ + order + """
3  typedef pycuda::complex<float> scmplx;
4  __device__ void get_ar_vnm(int sig, int num,
5   scmplx temp, scmplx *ar_vnm_temp);
6  __global__ void vnm_kernel(
7      scmplx *ar_vnm,
8      float *ar_rho,
9      scmplx *ar_exp,
10     int *current_pixels,
11     int *current_eight_count)
12 {
13     const int tx = threadIdx.x;
14     const int ty = threadIdx.y;
15     const int bw = blockDim.x;
16     const int bh = blockDim.y;
17     const int bx = blockIdx.x;
18     const int vnm_count = (int(local_order / 2) + 1)
19      * (int(local_order / 2) + 1 + local_order % 2);
20     const int current_pixels_local = *current_pixels;
21     const int current_eight_count_local =
22      *current_eight_count;
23
24     int pos = tx + ty*bw + bx*bw*bh;
25     float rho = ar_rho[current_pixels_local + pos];
```

```
26
27      if(pos < current_pixels_local){
28
29          float R[local_order + 1];
30          float Rp[local_order + 1];
31          float Rpp[local_order + 1];
32          scmplx temp = scmplx(0,0);
33          float temp_real = 0;
34          float temp_imag = 0;
35          int vnm_pos = 0;
36
37          if(pos < current_eight_count_local){
38              scmplx ar_vnm_temp[6];
39
40              ar_vnm[pos * 8 * vnm_count] = 1;
41              ar_vnm[(pos * 8 + 1) * vnm_count] = 1;
42              ar_vnm[(pos * 8 + 2) * vnm_count] = 1;
43              ar_vnm[(pos * 8 + 3) * vnm_count] = 1;
44              ar_vnm[(pos * 8 + 4) * vnm_count] = 1;
45              ar_vnm[(pos * 8 + 5) * vnm_count] = 1;
46              ar_vnm[(pos * 8 + 6) * vnm_count] = 1;
47              ar_vnm[(pos * 8 + 7) * vnm_count] = 1;
48
49              Rpp[0] = 1;
50              Rp[1] = rho;
51
52              for(int n = 1; n <= local_order; n++){
53
54                  if (n == 1){
55                      temp =
56                      ar_exp[(*current_pixels) + pos] * rho;
```

```
57                      ar_vnm[pos * 8 * vnm_count + 1]
58                       = temp;
59                      temp_real = temp.real();
60                      temp_imag = temp.imag();
61                      ar_vnm[(pos * 8 + 4)* vnm_count
62                       + 1] = pycuda::conj(temp);
63                      ar_vnm[(pos * 8 + 1)* vnm_count
64                       + 1] = scmplx(temp_imag, temp_real);
65                      ar_vnm[(pos * 8 + 2)* vnm_count
66                       + 1] = -pycuda::conj(temp);
67                      ar_vnm[(pos * 8 + 3)* vnm_count
68                       + 1] = scmplx(-temp_imag, temp_real);
69                      ar_vnm[(pos * 8 + 5)* vnm_count
70                       + 1] = scmplx(temp_imag, -temp_real);
71                      ar_vnm[(pos * 8 + 6)* vnm_count
72                       + 1] = -temp;
73                      ar_vnm[(pos * 8 + 7)* vnm_count
74                       + 1] = -scmplx(temp_imag, temp_real);
75                      continue;
76                  }
77                  R[n] =
78                   ar_rho[n * current_pixels_local + pos];
79                  vnm_pos = (int(n / 2) + 1) *
80                   (int(n / 2) + 1 + n % 2) - 1;
81                  temp = ar_exp[n * (*current_pixels)
82                   + pos] * R[n];
83                  ar_vnm[pos * 8 * vnm_count
84                  + vnm_pos] = temp;
85                  ar_vnm[(pos * 8 + 4)* vnm_count
86                   + vnm_pos] =  pycuda::conj(temp);
87                  get_ar_vnm(0, n, temp, ar_vnm_temp);
```

```
88                  ar_vnm[(pos * 8 + 1)* vnm_count
89                   + vnm_pos] = ar_vnm_temp[0];
90                  ar_vnm[(pos * 8 + 2)* vnm_count
91                   + vnm_pos] = ar_vnm_temp[1];
92                  ar_vnm[(pos * 8 + 3)* vnm_count
93                   + vnm_pos] = ar_vnm_temp[2];
94                  ar_vnm[(pos * 8 + 5)* vnm_count
95                   + vnm_pos] = ar_vnm_temp[3];
96                  ar_vnm[(pos * 8 + 6)* vnm_count
97                   + vnm_pos] = ar_vnm_temp[4];
98                  ar_vnm[(pos * 8 + 7)* vnm_count
99                   + vnm_pos] = ar_vnm_temp[5];
100
101                 if(n % 2 == 0){
102                     R[0] = 2 * rho * Rp[1] - Rpp[0];
103                     temp = R[0];
104                     temp_real = temp.real();
105                     temp_imag = temp.imag();
106                     ar_vnm[pos * 8 * vnm_count +
107                      vnm_pos - n / 2] = temp;
108                     ar_vnm[(pos * 8 + 4)* vnm_count +
109                      vnm_pos - n / 2] = temp;
110                     ar_vnm[(pos * 8 + 1)* vnm_count +
111                      vnm_pos - n / 2] = temp;
112                     ar_vnm[(pos * 8 + 2)* vnm_count +
113                      vnm_pos - n / 2] = temp;
114                     ar_vnm[(pos * 8 + 3)* vnm_count +
115                      vnm_pos - n / 2] = temp;
116                     ar_vnm[(pos * 8 + 5)* vnm_count +
117                      vnm_pos - n / 2] = temp;
118                     ar_vnm[(pos * 8 + 6)* vnm_count +
```

```
119                     vnm_pos - n / 2] = temp;
120                 ar_vnm[(pos * 8 + 7)* vnm_count +
121                     vnm_pos - n / 2] = temp;
122             }
123
124             temp =  scmplx(0,0);
125             for(int m = n - 2; m > 0; m = m - 2){
126                 R[m] = rho * (Rp[m-1] + Rp[m+1])
127                     - Rpp[m];
128                 vnm_pos = (int((n - 1) / 2) + 1) *
129                     (int((n - 1) / 2) + 1 + (n - 1) % 2)
130                     - 1 + int((m + 2) / 2);
131                 temp = R[m] *
132                     ar_exp[m * (*current_pixels) + pos];
133                 ar_vnm[pos * 8 * vnm_count
134                     + vnm_pos] = temp;
135                 ar_vnm[(pos * 8 + 4)* vnm_count
136                     + vnm_pos] = pycuda::conj(temp);
137                 get_ar_vnm(0, m, temp, ar_vnm_temp);
138                 ar_vnm[(pos * 8 + 1)* vnm_count
139                     + vnm_pos] = ar_vnm_temp[0];
140                 ar_vnm[(pos * 8 + 2)* vnm_count
141                     + vnm_pos] = ar_vnm_temp[1];
142                 ar_vnm[(pos * 8 + 3)* vnm_count
143                     + vnm_pos] = ar_vnm_temp[2];
144                 ar_vnm[(pos * 8 + 5)* vnm_count
145                     + vnm_pos] = ar_vnm_temp[3];
146                 ar_vnm[(pos * 8 + 6)* vnm_count
147                     + vnm_pos] = ar_vnm_temp[4];
148                 ar_vnm[(pos * 8 + 7)* vnm_count
149                     + vnm_pos] = ar_vnm_temp[5];
```

```
150                     temp =  scmplx(0,0);
151                 }
152                 for (int i = 0; i <= n; i++){
153                     Rpp[i] = Rp[i];
154                     Rp[i] = R[i];
155                 }
156             }
157         }else{
158             scmplx ar_vnm_temp[2];
159             int base_row =
160              current_eight_count_local * 8
161              + (pos - current_eight_count_local) * 4;
162             ar_vnm[base_row * vnm_count] = 1;
163             ar_vnm[(base_row + 1) * vnm_count] = 1;
164             ar_vnm[(base_row + 2) * vnm_count] = 1;
165             ar_vnm[(base_row + 3) * vnm_count] = 1;
166
167             Rpp[0] = 1;
168             Rp[1] = rho;
169
170             for(int n = 1; n <= local_order; n++){
171                 if (n == 1){
172                     temp =
173                      ar_exp[(*current_pixels) + pos] * rho;
174                     ar_vnm[base_row * vnm_count + 1] = temp;
175                     temp_real = temp.real();
176                     temp_imag = temp.imag();
177                     ar_vnm[(base_row + 2) * vnm_count + 1]
178                      = pycuda::conj(temp);
179                     ar_vnm[(base_row + 1)* vnm_count + 1]
180                      = scmplx(-temp_imag, temp_real);
```

```
181                     ar_vnm[(base_row + 3)* vnm_count + 1]
182                      = -scmplx(temp_imag, temp_real);
183                     continue;
184                 }
185                 R[n] = ar_rho[n * current_pixels_local + pos];
186                 vnm_pos = (int(n / 2) + 1) *
187                  (int(n / 2) + 1 + n % 2) - 1;
188                 temp =
189                  ar_exp[n * (*current_pixels) + pos] * R[n];
190
191                 ar_vnm[base_row * vnm_count + vnm_pos] = temp;
192                 ar_vnm[(base_row + 2)* vnm_count + vnm_pos]
193                  = pycuda::conj(temp);
194
195                 get_ar_vnm(1, n, temp, ar_vnm_temp);
196                 ar_vnm[(base_row + 1)* vnm_count + vnm_pos]
197                  = ar_vnm_temp[0];
198                 ar_vnm[(base_row + 3)* vnm_count + vnm_pos]
199                  = ar_vnm_temp[1];
200
201                 if(n % 2 == 0){
202                     R[0] = 2 * rho * Rp[1] - Rpp[0];
203                     temp = R[0];
204                     temp_real = temp.real();
205                     temp_imag = temp.imag();
206                     ar_vnm[base_row * vnm_count +
207                      vnm_pos - n / 2] = temp;
208                     ar_vnm[(base_row + 2)* vnm_count +
209                      vnm_pos - n / 2] = temp;
210                     ar_vnm[(base_row + 1)* vnm_count +
211                      vnm_pos - n / 2] = temp;
```

```
212                     ar_vnm[(base_row + 3)* vnm_count +
213                      vnm_pos - n / 2] = temp;
214                 }
215
216                 temp =  scmplx(0,0);
217                 for(int m = n - 2; m > 0; m = m - 2){
218                     R[m] = rho * (Rp[m-1]
219                      + Rp[m+1]) - Rpp[m];
220                     vnm_pos = (int((n - 1) / 2) + 1) *
221                      (int((n - 1) / 2) + 1 + (n - 1) % 2)
222                       - 1 + int((m + 2) / 2);
223                     temp = R[m] * ar_exp[
224                      m * (*current_pixels) + pos];
225
226                     ar_vnm[base_row * vnm_count
227                      + vnm_pos] = temp;
228                     ar_vnm[(base_row + 2)* vnm_count
229                      + vnm_pos]  = pycuda::conj(temp);
230                     get_ar_vnm(1, m, temp, ar_vnm_temp);
231                     ar_vnm[(base_row + 1)* vnm_count
232                      + vnm_pos]  = ar_vnm_temp[0];
233                     ar_vnm[(base_row + 3)* vnm_count
234                      + vnm_pos] = ar_vnm_temp[1];
235                     temp =  scmplx(0,0);
236                 }
237                 for (int i = 0; i <= n; i++){
238                     Rpp[i] = Rp[i];
239                     Rp[i] = R[i];
240                 }
241             }
242
```

```
243             }
244
245         }
246 }
247
248 __device__ void get_ar_vnm(int sig, int num,
249  scmplx temp, scmplx *ar_vnm_temp){
250     float temp_real = temp.real();
251     float temp_imag = temp.imag();
252     if (sig == 0){
253         if(num % 4 == 0){
254             ar_vnm_temp[0] = pycuda::conj(temp);
255             ar_vnm_temp[1] = pycuda::conj(temp);
256             ar_vnm_temp[2] = temp;
257             ar_vnm_temp[3] = temp;
258             ar_vnm_temp[4] = temp;
259             ar_vnm_temp[5] = pycuda::conj(temp);
260
261         }else if(num % 4 == 1){
262             ar_vnm_temp[0] = scmplx(temp_imag, temp_real);
263             ar_vnm_temp[1] = -pycuda::conj(temp);
264             ar_vnm_temp[2] = scmplx(-temp_imag, temp_real);
265             ar_vnm_temp[3] = scmplx(temp_imag, -temp_real);
266             ar_vnm_temp[4] = -temp;
267             ar_vnm_temp[5] = -scmplx(temp_imag, temp_real);
268
269         }else if(num % 4 == 2){
270             ar_vnm_temp[0] = -pycuda::conj(temp);
271             ar_vnm_temp[1] = pycuda::conj(temp);
272             ar_vnm_temp[2] = -temp;
273             ar_vnm_temp[3] = -temp;
```

```
274                 ar_vnm_temp[4] = temp;
275                 ar_vnm_temp[5] = -pycuda::conj(temp);
276
277             }else{
278                 ar_vnm_temp[0] = -scmplx(temp_imag, temp_real);
279                 ar_vnm_temp[1] = -pycuda::conj(temp);
280                 ar_vnm_temp[2] = -scmplx(-temp_imag, temp_real);
281                 ar_vnm_temp[3] = scmplx(-temp_imag, temp_real);
282                 ar_vnm_temp[4] = -temp;
283                 ar_vnm_temp[5] = scmplx(temp_imag, temp_real);
284
285             }
286
287         }else{
288             if(num % 4 == 0){
289                 ar_vnm_temp[0] = temp;
290                 ar_vnm_temp[1] = temp;
291
292             }else if(num % 4 == 1){
293                 ar_vnm_temp[0] = scmplx(-temp_imag, temp_real);
294                 ar_vnm_temp[1] = -scmplx(temp_imag, temp_real);
295
296             }else if(num % 4 == 2){
297                 ar_vnm_temp[0] = -temp;
298                 ar_vnm_temp[1] = -pycuda::conj(temp);
299
300             }else{
301                 ar_vnm_temp[0] = scmplx(temp_imag, -temp_real);
302                 ar_vnm_temp[1] = scmplx(temp_imag, temp_real);
303
304             }
```

```
305     }
306 }
```

Source code A.8: Fxy_Kernel

```
1  # include <pycuda-complex.hpp>
2  typedef pycuda::complex<float> scmplx;
3  __global__ void fxy_kernel(
4      scmplx *ar_fxy,
5      scmplx *ar_moments,
6      scmplx *ar_vnm,
7      int *order,
8      int *vnm_count,
9      int *current_pixels)
10 {
11     const int tx = threadIdx.x;
12     const int bx = blockIdx.x;
13     const int bw = blockDim.x;
14     const float pi = atan(1.)*4;
15
16     int pos = bx * bw + tx;
17     scmplx temp = scmplx(0, 0);
18
19     if(pos < (*current_pixels)){
20         int n0 = 0;
21         //When n is even
22         for (int i = 0; i <= (*order); i = i + 2){
23             n0 = i / 2 * (i / 2 + 1);
24             temp += (i + 1) / pi * ar_moments[n0] *
25              pycuda::conj(ar_vnm[pos * (*vnm_count) + n0]);
26             for (int j = i; j > 0; j = j - 2){
27                 temp += (i + 1) / pi * (ar_moments[
```

```
28                      n0 + j / 2] * pycuda::conj(ar_vnm[
29                      pos * (*vnm_count) + n0 + j / 2]) +
30                      pycuda::conj(ar_moments[n0 + j / 2]) *
31                      ar_vnm[pos * (*vnm_count) + n0 + j / 2]);
32                  }
33              }
34              //When n is odd
35              for (int i = 1; i <= (*order); i = i + 2){
36                  n0 = ((i - 1) / 2 + 1) * ((i - 1) / 2 + 1);
37                  for (int j = i; j > 0; j = j - 2){
38                      temp += (i + 1) / pi * (ar_moments[
39                      n0 + (j - 1) / 2] * pycuda::conj(ar_vnm[
40                      pos * (*vnm_count) + n0 + (j - 1) / 2]) +
41                      pycuda::conj(ar_moments[n0 + (j - 1) / 2])
42                      * ar_vnm[pos * (*vnm_count)
43                      + n0 + (j - 1) / 2]);
44                  }
45              }
46              ar_fxy[pos] = temp;
47          }
48      }
```

Bibliography

[1] M.-K. Hu, "Visual pattern recognition by moment invariants," *IRE TRANSACTIONS ON INFORMATION THEORY*, vol. 8, no. 2, pp. 179–189, 1962.

[2] K. M. Hosny, "Fast computation of accurate pseudo zernike moments for binary and gray-level images.," *Int. Arab J. Inf. Technol.*, vol. 11, no. 3, pp. 243–249, 2014.

[3] G. A. Papakostas, "Over 50 years of image moments and moment invariants," in *Moments and Moment Invariants - Theory and Applications* (G. A. Papakostas, ed.), vol. 1, Science Gate Publishing, 2014.

[4] K. M. Hosny, M. M. Darwish, K. Li, and A. Salah, "Parallel multi-core cpu and gpu for fast and robust medical image watermarking," *IEEE Access*, vol. 6, pp. 77212–77225, 2018.

[5] S. Li, M.-C. Lee, and C.-M. Pun, "Complex zernike moments features for shape-based image retrieval," *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, vol. 39, no. 1, pp. 227–237, 2008.

[6] C. Singh, E. Walia, Pooja, and R. Upneja, "Analysis of algorithms for fast computation of pseudo zernike moments and their numerical stability," *Digital Signal Processing*, vol. 22, no. 6, pp. 1031 – 1043, 2012.

[7] C.-W. Chong, P. Raveendran, and R. Mukundan, "A comparative analysis of algorithms for fast computation of zernike moments," *Pattern Recognition*, vol. 36, no. 3, pp. 731 – 742, 2003.

[8] C.-W. Chong, P. Raveendran, and R. Mukundan, "An efficient algorithm for fast computation of pseudo-zernike moments," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 17, no. 6, pp. 1011–1023, 2003.

[9] A.-W. Deng, C.-H. Wei, and C.-Y. Gwo, "Stable, fast computation of high-order zernike moments using a recursive method," *Pattern Recognition*, vol. 56, 03 2016.

[10] A.-W. Deng and C.-Y. Gwo, "Fast and stable algorithms for high-order pseudo zernike moments and image reconstruction," *Applied Mathematics and Computation*, vol. 334, pp. 239 – 253, 2018.

[11] M. Ujaldon, "Gpu acceleration of zernike moments for large-scale images," in *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing*, (USA), pp. 1–8, IEEE Computer Society, 2009.

[12] P. Toharia, O. D. Robles, R. Suárez, J. L. Bosque, and L. Pastor, "Shot boundary detection using zernike moments in multi-GPU multi-CPU architectures," *Journal of Parallel and Distributed Computing*, vol. 72, no. 9, pp. 1127 – 1133, 2012.

[13] M. J. Martín Requena, P. Moscato, and M. Ujaldón, "Efficient data partitioning for the gpu computation of moment functions," *Journal of Parallel and Distributed Computing*, vol. 74, no. 1, pp. 1994 – 2004, 2014.

[14] Y. Xuan, D. Li, and W. Han, "Efficient optimization approach for fast gpu computation of zernike moments," *Journal of Parallel and Distributed Computing*, vol. 111, pp. 104 – 114, 2018.

[15] J. Flusse, T. Suk, and B. Zitová, *2D and 3D Image Analysis by Moments*. Wiley & Sons Ltd., 2016.

[16] R. Mukundan, S.-H. Ong, and P. Lee, "Image analysis by tchebichef moments," *IEEE transactions on image processing : a publication of the IEEE Signal Processing Society*, vol. 10, pp. 1357–64, 02 2001.

[17] B. Xiao, L. Li, Y. Li, W. Li, and G. Wang, "Image analysis by fractional-order orthogonal moments," *Information Sciences*, vol. 382-383, pp. 135 – 149, 2017.

[18] K. M. Hosny, "A systematic method for efficient computation of full and subsets zernike moments," *Inf. Sci.*, vol. 180, pp. 2299–2313, 2010.

[19] von F. Zernike, "Beugungstheorie des schneidenver-fahrens und seiner verbesserten form, der phasenkontrastmethode," *Physica*, vol. 1, no. 7, pp. 689 – 704, 1934.

[20] C.-H. Teh and R. T. Chin, "On image analysis by the methods of moments," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 10, pp. 496–513, July 1988.

[21] S. Liao, "Accuracy analysis of moment functions," in *Moments and Moment Invariants - Theory and Applications* (G. A. Papakostas, ed.), vol. 1, pp. 33–56, Science Gate Publishing, 2014.

[22] A. B. Bhatia and E. Wolf, "On the circle polynomials of zernike and related orthogonal sets," *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 50, no. 1, pp. 40–48, 1954.

[23] A. Prata and W. Rusch, "Algorithm for computation of zernike polynomials expansion coefficients," *Applied Optics*, vol. 28, no. 4, pp. 749–754, 1989.

[24] E. Kintner, "On the mathematical properties of the zernike polynomials," *Journal of Modern Optics*, vol. August 1976, pp. 679–680, 11 2010.

[25] Q.-Y. Yang, F. Gao, and Q. Nie, "A modified l-iterative algorithm for fast computation of pseudo-zernike moments," in *2009 2nd International Congress on Image and Signal Processing*, pp. 1–5, 2009.

[26] M. S. Al-Rawi, "Fast computation of pseudo zernike moments," *Journal of Real-Time Image Processing*, vol. 5, no. 1, pp. 3–10, 2010.

[27] T. Xia, "Gpu-accelerated algorithm to compute bessel-fourier moments," Master's thesis, The University of Winnipeg, 2020.

[28] N. Corporation, "Cuda best practices guide," 07 2020.

[29] N. Corporation, "Cuda programming guide," 07 2020.

[30] M. Harris *et al.*, "Optimizing parallel reduction in cuda," *Nvidia developer technology*, vol. 2, no. 4, p. 70, 2007.

[31] K. Gan and K. Lua, "A new approach to stroke and feature point extraction in chinese character recognition," *Pattern Recognition Letter*, vol. 12, pp. 381–387, 1991.

[32] J. Liu and S. Ma, "An overview of printed chinese character recognition techniques," in *Proceedings of the International Conference on Chinese Computing*, (Singapore), pp. 325–333, June 4-7 1996.

[33] C. Lu, "A survey on chinese computing research," *Hong Kong Computer Journal*, vol. 9, no. 12, 1993.

[34] Y. H. Zhang and C. P. Liu, "The analysis and suggestion to the evaluation of chinese character recognition systems," *Proc. 1992 International Conference on Chinese Information Processing*, pp. 407–412, 1992.

[35] R. Dai, C.-L. Liu, and B. Xiao, "Chinese character recognition: History, status and prospects," *Frontiers of Computer Science in China*, vol. 1, pp. 126–136, 05 2007.

[36] S. Liao, A. Chiang, Q. Lu, and M. Pawlak, "Chinese character recognition via gegenbauer moments," in *Object recognition supported by user interaction for service robots*, vol. 3, pp. 485–488 vol.3, Aug 2002.

[37] T. Wang and S. Liao, "Chinese character recognition by zernike moments," in *2014 International Conference on Audio, Language and Image Processing*, pp. 771–774, July 2014.

[38] H. Bing and S. Liao, "Chinese character recognition by tchebichef moment features," *Lecture Notes on Software Engineering*, vol. 1, no. 4, p. 392, 2013.

[39] Y. Wu and S. Liao, "Chinese characters recognition via racah moments," in *2014 International Conference on Audio, Language and Image Processing*, pp. 691–694, July 2014.

[40] S. Liao and J. Chen, "Object recognition with lower order gegenbauer moments," *Lecture Notes on Software Engineering*, vol. 1, no. 4, pp. 387–391, 2013.