An Efficient Approach to Compute Zernike Moments With GPU-Accelerated Algorithm

by

Zhuohao Jia

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
July 2023

Abstract

The utilization of Zernike moments has been extensive in various fields, including image processing and pattern recognition, owing to their desirable characteristics. However, the application of Zernike moments is hindered by two significant obstacles: computational efficiency and accuracy. These issues become particularly noticeable when computing high-order moments. This study presents a novel GPU-based method for efficiently computing Zernike moments by leveraging the computational power of the Single Instruction Multiple Data(SIMD) architecture. The experimental results demonstrate that the proposed method can compute Zernike moments up to order 500 within 0.5 seconds for an image of size 512×512. To achieve greater accuracy in Zernike moments computation, a $k \times k$ sub-region scheme was incorporated into the approach. The results show that the PSNR value of the Lena image reconstructed from 500-order Zernike moments computed using the 9×9 scheme can reach 39.20 dB. Furthermore, a method for leaf recognition that leverages Zernike moments as image features, with the k-Nearest Neighbors (k-NN) algorithm serving as the classifier is proposed. The proposed method is evaluated on the Flavia leaf dataset, and the results affirm the effectiveness of the approach.

Keywords: Zernike moments, Fast computation, GPU, Image reconstruction.

Acknowledgements

I would like to express my profound appreciation to my supervisor, Dr. Simon Liao, for providing me with the invaluable opportunity to pursue my master's degree at the University of Winnipeg. Your guidance, support, and encouragement throughout my research have been of immeasurable value to me. Your exceptional expertise, patience, and commitment have been instrumental in helping me navigate the complexities and challenges encountered during this project.

Furthermore, I would like to extend my gratitude to Dr. Christopher Henry for introducing me to the concept of GPU parallel computing, which has broadened my understanding of high-performance computing. Without the solid foundation established in your course, the completion of this project would not have been possible.

I would also like to express my appreciation to Dr. Sheela Ramanna. The field of AI has been a prevalent topic in recent years, and it is your teaching that has allowed me to step into the world of AI and obtain a fundamental understanding of this field.

I am deeply grateful to Dr. Yangjun Chen for your knowledge and expertise in the areas of data structure and algorithms, which have enriched my understanding of these crucial domains. Your invaluable teachings have enabled me to enhance my skillset and become more competitive in the job market.

Additionally, I wish to extend my heartfelt gratitude to all the esteemed members of the thesis committee, Dr. Christopher Henry and Dr. Christopher Bidinosti, for the invaluable insights and comments.

Lastly, I would like to extend my deepest gratitude to my parents for their unconditional love, nurturing, and unwavering support. It is because of your encouragement and trust in my decisions that I was able to come to Canada to pursue my academic aspirations.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Zernike moments, initially introduced by Teague in 1980 [21], constitute a set of functions that map an image onto a collection of orthogonal complex Zernike polynomials [22]. These moments not only represent an image with minimal information redundancy, but also possess the inherent property of being invariant to image rotation and reflection [25]. Owing to these advantageous characteristics, Zernike moments have been widely employed in various fields including image recognition [19, 3, 10], image retrieval [12, 11], and image watermarking [6, 7].

However, due to the intricate definition of Zernike moments, which incorporate a series of factorials and trigonometric functions, the computation process of these moments can be computationally intensive, rendering them inapplicable for real-time applications or scenarios involving a substantial number of images. To address this issue, many studies have been conducted by scholars to accelerate the computation process.

Several researchers have proposed the utilization of recursive relationships between Zernike radial polynomials to circumvent the need for direct calculation by definition, as the factorial terms of the radial polynomials are a significant contributor to the prolonged computation time. Prata [17] and Kintner [8] developed their own recursive relationships. However, both methods have limitations and are not applicable to certain cases with specific values of order n and repetition m, and thus, those polynomials must be calculated through the direct method. Singh [20] and Chong [1] have modified these methods respectively to make them applicable to any cases. Additionally, Chong [1] has also proposed a q-recursive method.

Other approaches include utilizing the symmetry properties of trigonometric functions to reduce computational workload. In [4], Hwang first intro-

duced a 4-symmetric method for computing the Angular Radial Transform (ART). He subsequently proposed an 8-symmetric method, which was applied to the computation of Zernike moments [5]. This method requires the computation of only one octant of the entire image, thus reducing the computation time by approximately one-eighth.

The methods previously discussed are based on Central Processing Unit (CPU). However, in recent years, Graphics Processing Unit (GPU), which utilizes a Single Instruction Multiple Data (SIMD) architecture, has demonstrated remarkable capability for high-performance computing and has accelerated various applications that require processing-intensive operations. Several researchers have effectively employed GPUs to enhance the computation process of Zernike moments [18, 26]. However, their methods involve the direct calculation of factorials in the definition of Zernike radial polynomials, which can result in overflow errors, leading to numerical instability for orders greater than 45, even when double precision is used [20]. Therefore, their methods are not suitable for the computation of high-order moments.

In this study, a GPU-based method for the computation of high-order Zernike moments was developed using the CUDA C++ platform. The proposed method employs the radial polynomials' recursive relationship and symmetry property and has been optimized by applying various techniques to adapt it to the architecture of GPU. The results of the experimentation demonstrated that the proposed method is efficient and accurate in computing Zernike moments. Furthermore, to fully leverage the high computational capabilities of GPU, a $k \times k$ sub-region scheme was also implemented, resulting in more accurate Zernike moments, particularly for high-order moments.

Furthermore, given that Zernike moments can efficiently depict information contained within an image, they can be implemented for image classification. This study proposed a method for the recognition of leaves that utilizes Zernike moments as image features, with the k-Nearest Neighbors algorithm (k-NN) serving as the classifier. The method was subsequently

evaluated utilizing the Flavia leaf dataset, with the outcomes demonstrating the effectiveness of the approach.

The remaining chapters of the thesis are structured as follows. In Chapter 2, the definition and properties of Zernike moments will be presented. Chapter 3 will delve into the existing approaches that enhance the computational efficiency and accuracy of Zernike moments. Subsequently, the proposed GPU-based method will be thoroughly expounded in Chapter 4. To assess the computational efficiency and accuracy of the proposed method, experiments were conducted and the results will be presented in Chapter 5. The method that leverages Zernike moments to perform leaf recognition will be introduced in Chapter 6. Lastly, the conclusion will be given in Chapter 7.

Chapter 2

Zernike Moments

In [27], Zernike proposed a set of complex orthogonal polynomials defined on the unit disk, i.e., $x^2 + y^2 \leq 1$. An image's Zernike moments are a set of projections of that image onto these polynomials.

The initial step towards obtaining Zernike moments of a given image is the computation of Zernike radial polynomials. The real-valued radial polynomial $R_{nm}(\rho)$ of order n with repetition m is defined as:

$$R_{nm}(\rho) = \sum_{s=0}^{(n-|m|)/2} c(n, m, s)\rho^{n-2s}. \tag{1}$$

The coefficient of the radial polynomial $c(n, m, s)$ is defined as:

$$c(n, m, s) = (-1)^s \frac{(n-s)!}{s!((n+|m|)/2-s)!((n-|m|)/2-s)!}, \tag{2}$$

where the order n is required to be a non-negative integer, while the repetition m is an integer that can be positive, negative, or zero. Furthermore, it is also necessary for them to satisfy the conditions of $n - |m| = even$ and $|m| \leq n$.

The complex-valued Zernike polynomial $V_{nm}(x, y)$, can subsequently be constructed from the radial polynomial $R_{nm}(\rho)$:

$$V_{nm}(x, y) = R_{nm}(\rho)e^{jm\theta}, \quad x^2 + y^2 \leq 1, \tag{3}$$

where $\rho = \sqrt{x^2 + y^2}$ represents the distance between the pixel (x, y) and the origin, $\theta = tan^{-1}(y/x)$ represents the angle between the x axis and the line formed by the pixel (x, y) and the origin. And j is the imaginary unit, which is $\sqrt{-1}$.

The two-dimensional Zernike moments of order n with repetition m, de-

noted as Z_{nm}, are defined as follows:

$$Z_{nm} = \frac{n+1}{\pi} \iint_{x^2+y^2 \leq 1} V_{nm}^*(x,y) f(x,y) \mathrm{d}x\mathrm{d}y, \tag{4}$$

where $f(x,y)$ represents the image intensity function and $*$ denotes the complex conjugate.

In order to compute Zernike moments of a digital image, the image must be mapped onto the unit disk, as illustrated in Figure 1. Assuming the image has a size of $N \times N$, the transformed coordinates (x_i, y_j) of the pixel located in the j-th row and i-th column of the original image will be:

$$\begin{cases} x_i = \frac{2i+1-N}{N} \\ y_j = \frac{2j+1-N}{N} \end{cases} \tag{5}$$

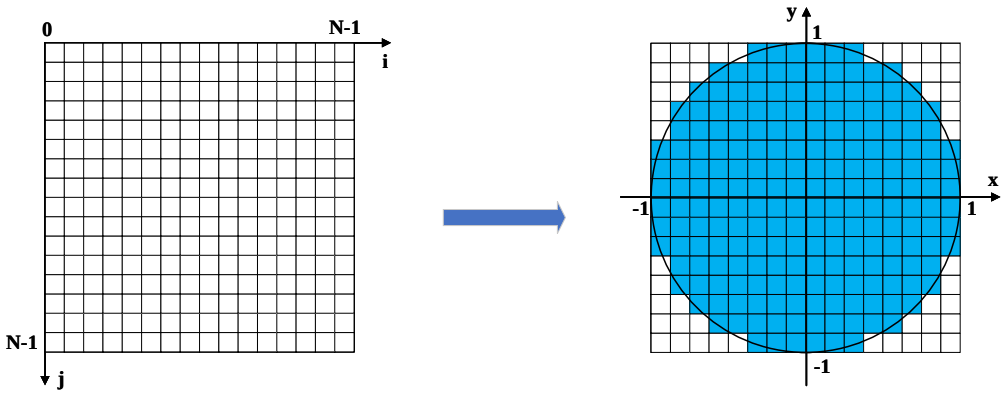


Figure 1: Illustration of an image with a size of $N \times N$ mapped over the unit disk

The pixels with centers located within the unit disk will be utilized for the computation of Zernike moments. Conversely, those located outside of the unit disk will be disregarded. In accordance with Eq.(4), in order to calculate Zernike moments of an image in the discrete domain, the integration will be

substituted with summations, resulting in:

$$\hat{Z}_{nm} = \frac{n+1}{\pi} \sum_{x_i^2+y_j^2 \leq 1} \sum V_{nm}^*(x_i, y_j) f(x_i, y_j) \Delta x \Delta y, \tag{6}$$

where Δx and Δy are the sampling intervals in the x and y directions [13], with the constant value $\Delta x = \Delta y = \frac{2}{N}$.

As defined in the above explanation of the Zernike radial polynomials, it follows that any two radial polynomials with the same repetition m but differing order n satisfy:

$$\int_0^1 R_{nm}(\rho) R_{n'm}(\rho) \rho \mathrm{d}\rho = \begin{cases} \frac{1}{2(n+1)} & n = n' \\ 0 & n \neq n' \end{cases}. \tag{7}$$

The Zernike polynomials are orthogonal and satisfy:

$$\iint_{x^2+y^2 \leq 1} V_{nm}^*(x, y) V_{pq}^*(x, y) \mathrm{d}x \mathrm{d}y = \begin{cases} \frac{\pi}{n+1} & n = p, m = q \\ 0 & \text{otherwise} \end{cases}. \tag{8}$$

The orthogonality property results in minimal redundancy and confers the capability of each individual Zernike moment to depict various components of the image. Consequently, the original image can be reconstructed by the summation of these individual Zernike moments:

$$f(x, y) = \sum_n^{\infty} \sum_m Z_{nm} V_{nm}(\rho, \theta). \tag{9}$$

In actuality, it is only feasible to obtain a limited number of Zernike moments. Assuming that a set of Zernike moments Z_{nm} for an image $f(x, y)$ up to a specified order N_{max} is available, the approximated image can be

reconstructed as follows:

$$\hat{f}(x,y) = \sum_{n=0}^{N_{max}} \sum_{m} Z_{nm} V_{nm}(\rho, \theta). \tag{10}$$

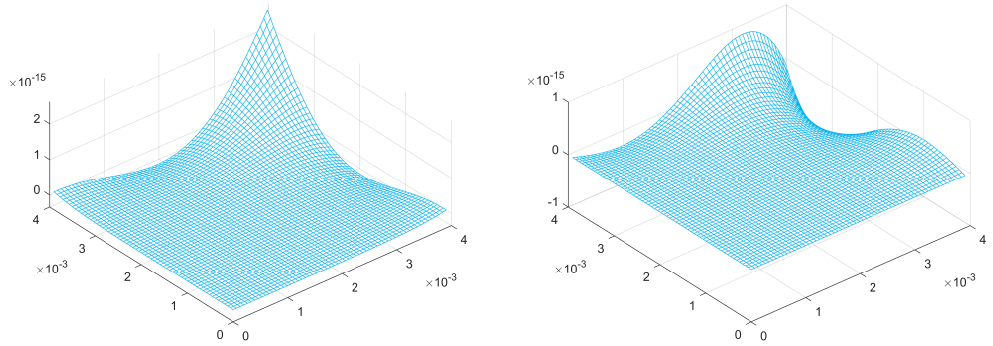
Chapter 3

Zernike Moments Computation

As for the computation of Zernike moments, accuracy and efficiency are the two aspects the researchers mainly focused on over the past years.

3.1 Computational Accuracy

As previously discussed in Chapter 2, when computing Zernike moments for a digital image in the discrete domain, the double integration in Eq.(4) is estimated through double summation, as shown in Eq.(6), which inevitably leads to an approximation error. If the distribution of the Zernike polynomial within each pixel is consistent, the error is generally minimal. Nonetheless, in the majority of cases, the distribution can exhibit substantial variations within a single pixel. Figure 2 presents the depiction of the distribution of the real and imaginary parts of $V_{20,8}(x, y)$ for a single central pixel located at $(256, 257)$ in an image with a size of 512×512.



(a) Real part of $V_{20,8}(x, y)$ (b) Imaginary part of $V_{20,8}(x, y)$

Figure 2: Distribution of the real and imaginary part of $V_{20,8}(x, y)$ within the pixel at location (256, 257) of an image with a size of 512×512

To mitigate the approximation error, the $k \times k$ sub-region scheme [23] can be employed. Under this scheme, each pixel is further divided into $k \times k$ sub-regions with equal weight, which means the sampling intervals become $\frac{\Delta x}{k}$ and $\frac{\Delta y}{k}$. Consequently, Eq.(6) can be reformulated as follows:

$$\hat{Z}_{nm} = \frac{n+1}{\pi} \sum_{x_i^2+y_j^2 \leq 1} \sum f(x_i, y_j) h_{nm}(x_i, y_j), \tag{11}$$

where

$$h_{nm}(x_i, y_j) = \sum_{s=1}^{k} \sum_{t=1}^{k} V_{nm}^*(x_{is}, y_{jt}) \frac{\Delta x}{k} \frac{\Delta y}{k}. \tag{12}$$

By utilizing a higher value of k (i.e. incorporating more sampling points), higher computational accuracy can be achieved. However, it should be noted that this will also result in a corresponding increase in computational workload.

3.2 Computational Efficiency

The computation of Zernike moments through the application of their definition is referred to as the direct method. However, due to the presence of a multitude of factorials and trigonometric functions within the definition, the computational process can be time-consuming. To address this issue, various researchers have proposed alternative methods to expedite the computation.

3.2.1 Recursive Method

Kintner [8] has suggested a recursive relationship between radial polynomials of different orders, allowing for the derivation of a radial polynomial of a given order from two lower-order polynomials with the same repetition m.

The relationship is specified as follows:

$$R_{n,m}(\rho) = \frac{(K_2\rho^2 + K_3)R_{n-2,m}(\rho) + K_4 R_{n-4,m}(\rho)}{K_1} \quad n = m+4, m+6, ... \tag{13}$$

where the coefficients K_1, K_2, K_3 and K_4 are defined as:

$$\begin{aligned} K_1 &= \frac{(n+m)(n-m)(n-2)}{2}, \\ K_2 &= 2n(n-1)(n-2), \\ K_3 &= -m^2(n-1) - n(n-1)(n-2), \\ K_4 &= \frac{n(n+m-2)(n-m-2)}{2}. \end{aligned} \tag{14}$$

This approach circumvents the direct calculation of a series of factorials, resulting in a substantial increase in computational speed. However, the initial values of the radial polynomials, i.e. when $n = m$ and $n = m + 2$, must be obtained using the direct method. Chong [1] proposed a modified Kintner's method by introducing two definitions:

$$R_{n,n}(\rho) = \rho^n \quad n = 0, 1, 2, ... \tag{15}$$

$$R_{n,m}(\rho) = nR_{n,n}(\rho) - (n-1)R_{n-2,n-2}(\rho) \quad n = m+2, \tag{16}$$

so that the entire set of radial polynomials can be obtained without resorting to the direct method.

Building upon the modified Kintner's method, Singh [20] introduced a Kintner's fast method by redefining the coefficients, resulting in a reduction of required calculations:

$$R_{n,m}(\rho) = \frac{(K_2'\rho^2 + K_3')R_{n-2,m}(\rho) + K_4' R_{n-4,m}(\rho)}{K_1} \quad n = m+4, m+6, ... \tag{17}$$

where the coefficients K_2', K_3' and K_4' are defined as:

$$K_2' = K_2/K_1, \quad K_3' = K_3/K_1, \quad K_4' = K_4/K_1, \tag{18}$$

with K_1, K_2, K_3 and K_4 defined in Eq.(14).

In contrast to the fixed repetition m, Chong [1] proposed a q-recursive method that enables the derivation of a radial polynomial from two polynomials of higher repetition m but the same order n:

$$R_{n,m}(\rho) = Q_1 R_{n,m+4}(\rho) + (Q_2 + \frac{Q_3}{\rho^2}) R_{n,m+2}(\rho), \tag{19}$$

where the coefficients Q_1, Q_2 and Q_3 are defined as:

$$\begin{aligned} Q_1 &= \frac{m(m-1)}{2} - mQ_2 + \frac{Q_3(n+m+2)(n-m)}{8}, \\ Q_2 &= \frac{Q_3(n+m)(n-m+2)}{4(m-1)} + (m-2), \\ Q_3 &= \frac{-4(m-2)(m-3)}{(n+m-2)(n-m+4)}. \end{aligned} \tag{20}$$

For those cases where $n = m$ and $n = m + 2$, the following equations will be used:

$$R_{n,n}(\rho) = \rho^n \quad n = 0, 1, 2, ... \tag{21}$$

$$R_{n,m}(\rho) = nR_{n,n}(\rho) - (n-1)R_{n-2,n-2}(\rho) \quad n = m + 2. \tag{22}$$

Another commonly used recursive relationship is Prata's method, which is defined as follows:

$$R_{n,m}(\rho) = P_1 R_{n-1,m-1}(\rho) + P_2 R_{n-2,m}(\rho), \tag{23}$$

where the coefficients P_1 and P_2 are defined as:

$$\begin{aligned} P_1 &= \frac{2n}{n+m}, \\ P_2 &= \frac{n-m}{n+m}. \end{aligned} \tag{24}$$

Singh [20] further modified the Prata's method, which became:

$$R_{n,n}(\rho) = \rho^n \quad n = 0, 1, 2, ... \tag{25}$$

$$\begin{aligned} R_{n+m,m}(\rho) = P_1 \rho R_{n+m-1,|m-1|}(\rho) + P_2 R_{n+m-2,m}(\rho), \\ n = 2, 4, 6...(m = 0, 1, 2, ...). \end{aligned} \tag{26}$$

3.2.2 Utilizing the Symmetry Property

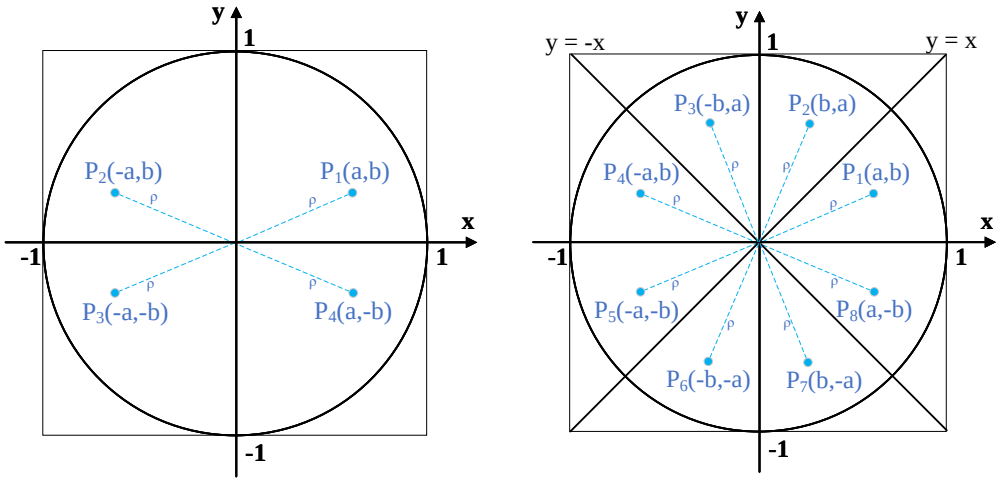
In [4], Hwang proposed a 4-symmetric method for computing the Angular Radial Transform(ART), which can also be utilized for the calculation of Zernike moments. As depicted in Figure 3a, given a pixel $P_1(a, b)$ in an image, it is possible to easily identify a set of pixels P_2, P_3 and P_4 that are symmetric to pixel P_1 about the y-axis, the origin, and the x-axis, respectively. The properties of these pixels are displayed in Table 1.

Table 1: The properties of the four symmetric pixels.

Pixel	Coordinates	Distance	Phase angle	Intensity
P_1	(a, b)	ρ	θ	f_1
P_2	$(-a, b)$	ρ	$\pi - \theta$	f_2
P_3	$(-a, -b)$	ρ	$\pi + \theta$	f_3
P_4	$(a, -b)$	ρ	$2\pi - \theta$	f_4

As per Euler's formula, the Zernike polynomials $V_{nm}(x, y)$ described in Eq.(3) can be expanded into:

$$V_{nm}(x, y) = R_{nm}(\rho) cos(m\theta) + j R_{nm}(\rho) sin(m\theta). \tag{27}$$



(a) Four symmetric (b) Eight symmetric

Figure 3: Illustration of symmetric methods

For the four symmetric pixels, the values of their radial polynomials $R_{nm}(\rho)$ will be equal as they have the same distance ρ. With respect to $cos(m\theta)$ and $sin(m\theta)$, being periodic and oscillatory, their values for the four points are either the same or opposite. Thus, once the value of one pixel is determined, the other three can be derived from it using the symmetric properties. As a result, the equation for Zernike moments can be rewritten as:

$$Z_{nm} = \frac{n+1}{\pi} \sum_{\substack{x_i^2+y_j^2 \leq 1, \\ x \geq 0, y \geq 0}} \sum R_{nm}(\rho)[g_m^r(x,y) - j g_m^i(x,y)] \Delta x \Delta y, \tag{28}$$

where the real part $g_m^r(x,y) = f(x,y)cos(m\theta)$ and the imaginary part $g_m^i(x,y) = f(x,y)sin(m\theta)$. Their values are separated into two cases depending on whether m is even or odd (i.e. $m = 2k$ or $m = 2k+1$, $k = 0, 1, 2, ...$):

$$g_m^r(x,y) = \begin{cases} (f_1 + f_2 + f_3 + f_4)cos(m\theta), & m = 2k, \\ (f_1 - f_2 - f_3 + f_4)cos(m\theta), & m = 2k+1. \end{cases} \tag{29}$$

$$g_m^i(x,y) = \begin{cases} (f_1 - f_2 + f_3 - f_4)sin(m\theta), & m = 2k, \\ (f_1 + f_2 - f_3 - f_4)sin(m\theta), & m = 2k+1. \end{cases} \tag{30}$$

Adopting the four symmetric property enables the computation of the radial polynomials for only one quadrant of the pixels, resulting in a reduction of the computational workload to one-fourth.

Based on the 4-symmetric method, Hwang subsequently proposed an 8-symmetric method [5]. As illustrated in Figure 3b, eight pixels are symmetric with respect to the x-axis, y-axis, $y = x$ and $y = -x$, and their properties are presented in Table 2.

Table 2: The properties of the eight symmetric pixels.

Pixel	Coordinates	Distance	Phase angle	Intensity
P_1	(a, b)	ρ	θ	f_1
P_2	(b, a)	ρ	$\frac{\pi}{2} - \theta$	f_2
P_3	$(-b, a)$	ρ	$\frac{\pi}{2} + \theta$	f_3
P_4	$(-a, b)$	ρ	$\pi - \theta$	f_4
P_5	$(-a, -b)$	ρ	$\pi + \theta$	f_5
P_6	$(-b, -a)$	ρ	$\frac{3\pi}{2} - \theta$	f_6
P_7	$(b, -a)$	ρ	$\frac{3\pi}{2} + \theta$	f_7
P_8	$(a, -b)$	ρ	$2\pi - \theta$	f_8

Similar to the 4-symmetric method, Zernike moments can be calculated using the following formula:

$$Z_{nm} = \frac{n+1}{\pi} \sum_{\substack{x_i^2 + y_j^2 \leq 1, \\ 0 \leq x \leq 1, 0 \leq y \leq x}} \sum R_{nm}(\rho)[g_m^r(x,y) - jg_m^i(x,y)]\Delta x \Delta y, \tag{31}$$

where $g_m^r(x,y)$ and $g_m^i(x,y)$ are organized into four categories:

$$g_m^r(x,y) = \begin{cases} (f_1+f_2+f_3+f_4+f_5+f_6+f_7+f_8)cos(m\theta), & m=4k, \\ (f_1-f_4-f_5+f_8)cos(m\theta)+(f_2-f_3-f_6+f_7)sin(m\theta), & m=4k+1, \\ (f_1-f_2-f_3+f_4+f_5-f_6-f_7+f_8)cos(m\theta), & m=4k+2, \\ (f_1-f_4-f_5+f_8)cos(m\theta)+(-f_2+f_3+f_6-f_7)sin(m\theta), & m=4k+3. \end{cases} \tag{32}$$

$$g_m^i(x,y) = \begin{cases} (f_1-f_2+f_3-f_4+f_5-f_6+f_7-f_8)sin(m\theta), & m=4k, \\ (f_1+f_4-f_5-f_8)sin(m\theta)+(f_2+f_3-f_6-f_7)cos(m\theta), & m=4k+1, \\ (f_1+f_2-f_3-f_4+f_5+f_6-f_7-f_8)sin(m\theta), & m=4k+2, \\ (f_1+f_4-f_5-f_8)sin(m\theta)+(-f_2-f_3+f_6+f_7)cos(m\theta), & m=4k+3. \end{cases} \tag{33}$$

Therefore, the computation can be restricted to a single octant, resulting in a reduction of the workload to one-eighth.

Chapter 4

GPU-Based Implementation to Compute Zernike Moments

4.1 GPU Architecture Introduction

The research utilized a GPU designed by NVIDIA, consisting of a collection of multi-threaded Streaming Multiprocessors(SMs). The SMs are based on Single Instruction Multiple Data(SIMD) architecture, as depicted in Figure 4. Each SM in the GPU features a control unit and multiple processing units, allowing it to simultaneously execute numerous threads through fetching a single instruction and processing different data elements. The fundamental operational unit within a GPU is referred to as a warp, which comprises 32 threads, and multiple warps can be executed concurrently on a SM. The distinction between a CPU and GPU lies in their design, as the former prioritizes a robust Arithmetic Logic Unit(ALU) to attain the highest possible speed of execution for a single thread. On the other hand, the latter has a lower performance for a single thread, but it can process a vast number of threads simultaneously, resulting in an overall higher throughput. Thus, a GPU is more appropriate for handling tasks that necessitate intensive parallel computations.

In order to create programs that utilize a GPU, the use of CUDA is required. CUDA is a general-purpose parallel computing platform developed by NVIDIA, enabling developers to write code specifically for GPU implementation. A CUDA program consists of two components, namely the host component and the device component. The host component refers to the program that is executed on the CPU, while the device component refers to the portion of the program that is executed on the GPU. The execution of a CUDA program starts on the host component, which then invokes the func-

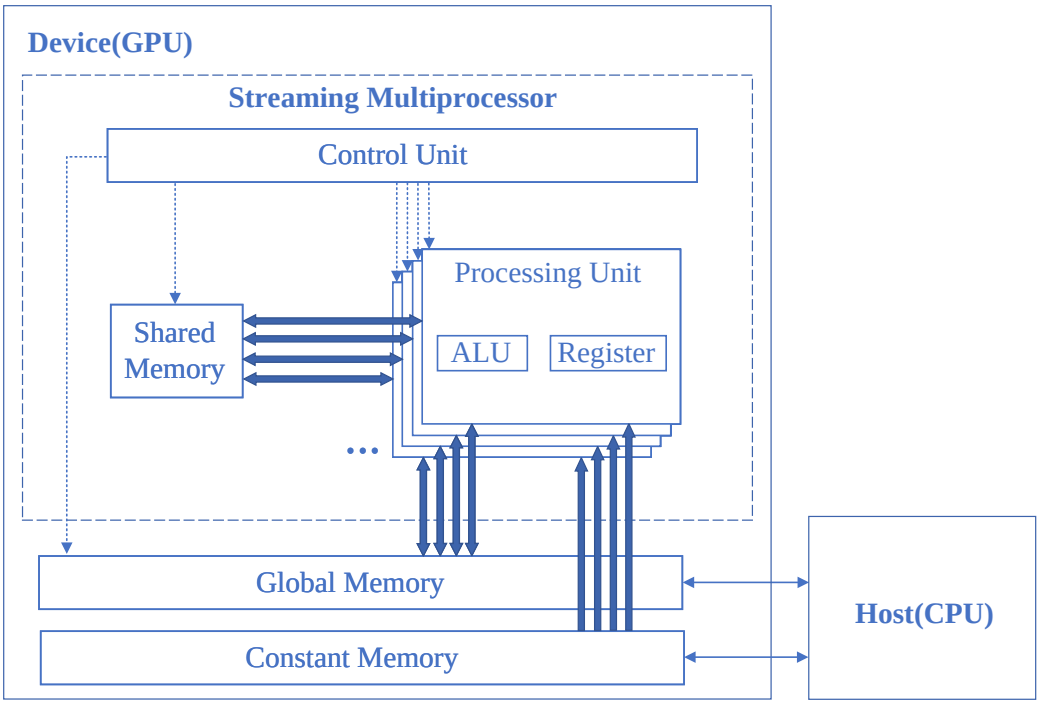


Figure 4: The architecture of the Graphics Processing Unit(GPU)

tions on the device, referred to as kernel functions. Developers are required to specify the quantity of CUDA threads to be executed upon invoking the kernel functions. These threads are organized into thread blocks, which are subsequently distributed to the SMs for execution.

It is important to note that prior to the invocation of kernel functions, the data to be processed must be transferred from the CPU's memory to the GPU's memory. Upon completion of the kernel function execution, the data will then be transferred back. The duration of the data transfer process is typically substantial and must be taken into account when evaluating the computational efficiency of algorithms.

4.2 Main Structure of the Proposed Algorithm

As previously discussed in Chapter 2, the computation of Zernike moments Z_{nm} of order n with repetition m requires multiplying the intensity $f(x, y)$ of each pixel (x, y) on the unit disk with the corresponding Zernike polynomial $V^*_{nm}(x, y)$. Subsequently, these products are summed to produce the final result. The computation process for the product of each pixel is independent, thus demonstrating a substantial intrinsic parallelism within the calculation of Zernike moments, making it suitable for the SIMD structure of the GPU. The fundamental concept of the GPU-based algorithm is to leverage the parallel processing capability of the GPU by utilizing each thread to calculate the product of individual pixels simultaneously. This approach theoretically allows for a reduction in time complexity from $O(N^2 n^2_{max})$ to $O(n^2_{max})$, provided that adequate computational resources are available.

The proposed method is comprised of four stages: The first stage involves preprocessing the input image and migrating the data from the CPU memory to the GPU. In the second stage, a GPU kernel will be activated to calculate the value of $f(x, y) \times V^*_{nm}(x, y)$ for every pixel. Subsequently, the sum of the products for all pixels obtained from the second stage will be computed in the third stage. At the end the data will be transferred back to the CPU to finish the whole process.

4.3 Data Pre-Processing

The data pre-processing part is executed by a CPU utilizing the pseudocode described in Algorithm 1. It is assumed that the image to be processed has a size of $N \times N$. Furthermore, the $k \times k$ sub-region scheme is implemented, resulting in each pixel being divided into $k \times k$ sub-pixels. Consequently, the image can be regarded as having a size of $kN \times kN$. It should be noted that all sub-pixels possess the same intensity as their respective parent pixel. Prior to executing the computation via the GPU kernel, it is necessary to

transfer the intesity, ρ and θ values of the pixels from the memory of the CPU to that of the GPU. The most direct approach to accomplish this objective is to allocate memory on the CPU for three arrays, $f[kN \times kN]$, $rho[kN \times kN]$ and $theta[kN \times kN]$, that store the respective values of all pixels within the image, followed by transferring these arrays to the GPU. However, this would result in a thread divergence issue, leading to a decrease in the computational efficiency.

The computation process of Zernike moments only involves pixels located on the unit disk. As such, if the raw data is transferred to the GPU without modification, a data selection task must be performed by the GPU, as depicted in Figure 5. The data is stored in a row-major layout, wherein the contiguous elements of a given row are positioned adjacent to each other. As previously discussed in Section 4.1, 32 threads within a warp can be executed simultaneously on the GPU if they all follow the same execution path. However, the task of data selection would result in the threads within a warp taking divergent control flow paths. For instance, elements 0 and 1 in Figure 5 lie outside the unit disk and as a result, they will proceed along one branch that skips the computation process. Conversely, elements 2 to 5, which are situated within the unit disk, must follow another branch to partake in the computation. Although these elements belong to the same warp, the two branches must be executed sequentially, leading to an increase in execution time. A similar divergence problem also occurs among the remaining elements.

To achieve increased parallelism, it is necessary to perform data pre-processing prior to invoking the GPU kernel. This enables the CPU to handle the data selection task and eliminate those pixels that will not participate in the computation process, as the CPU is optimized for complex and branch-intensive tasks. Following the data pre-processing step, as demonstrated in Situation 2 of Figure 5, the thread divergence problem is resolved and all elements within a warp can be executed simultaneously, leading to improved

computational efficiency. Additionally, pre-processing reduces the amount of data that needs to be transferred to the GPU, potentially decreasing the total execution time.

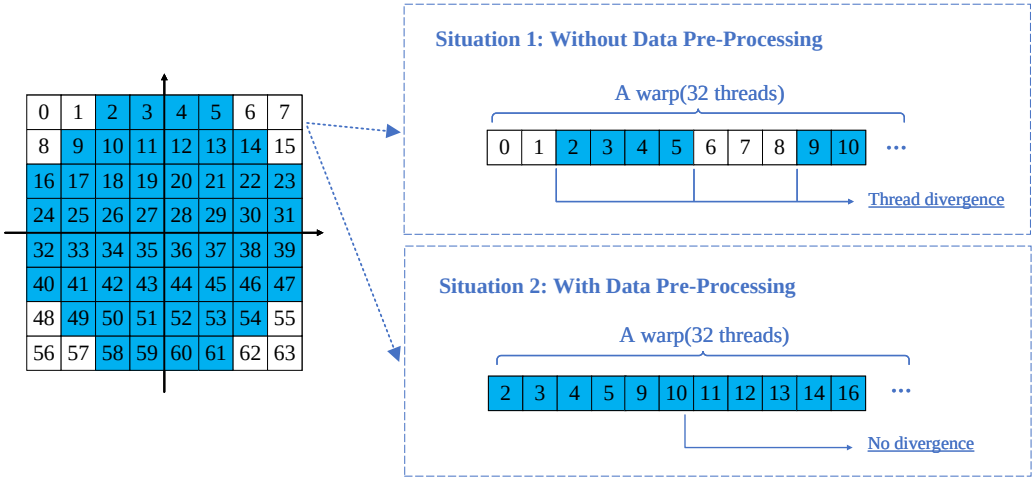


Figure 5: Illustration of the thread divergence problem

Algorithm 1 Data Pre-Processing

function ZERNIKE_DATA_PROCESSING(img, max_order, k)
  $width \leftarrow$ The width of the image
  $width\_k \leftarrow width \times k$ ▷ Split into $k \times k$ sub-region
  $total\_pixel\_num \leftarrow 0$ ▷ The total number of pixels on the unit disk
  $f[width\_k \times width\_k] \leftarrow 0$ ▷ Store the intensity of the pixels
  $rho[width\_k \times width\_k] \leftarrow 0$ ▷ Store the ρ value of the pixels
  $theta[width\_k \times width\_k] \leftarrow 0$ ▷ Store the θ value of the pixels
  for Each pixel of the image **do**
    if All the $k \times k$ sub-pixels of the current pixel are on the unit disk **then**
      for Each sub-pixel **do**
        $f[total\_pixel\_num] \leftarrow$ Current pixel's intensity
        $rho[total\_pixel\_num] \leftarrow$ Current sub-pixel's ρ
        $theta[total\_pixel\_num] \leftarrow$ Current sub-pixel's θ
        $total\_pixel\_num \leftarrow total\_pixel\_num + 1$
      end for
    end if
  end for
  Copy $f[\,], rho[\,], theta[\,]$ respectively from CPU's memory to GPU's
  $z\_num \leftarrow$ The total number of Zernike moments up to max_order
  Allocate memory for $Znm\_real[z\_num], Znm\_imag[z\_num]$ on both CPU and GPU
  Invoke $Zernike\_Kernel()$ to compute Zernike moments on GPU
  Copy $Znm\_real[\,], Znm\_imag[\,]$ from GPU's memory to CPU's
end function

4.4 GPU Kernel Function to Compute $f(x, y) \times V_{nm}^{*}(x, y)$

One of the essential components of the proposed methodology is the kernel function, which calculates the product of $f(x, y) \times V_{nm}^{*}(x, y)$, as depicted in Algorithm 2. The kernel is based on Kintner's fast method. Upon each invocation of the kernel, the results of a fixed repetition m with different orders n, ranging from m to the maximum order, will be calculated. In order to obtain the complete results, the kernel must be repeatedly invoked by incrementally varying the repetition value from 0 to the maximum order.

As described in Section 4.3, following the data pre-processing stage, the intensity $f(x, y)$, ρ and θ values of the pixels are transferred to the global memory of the GPU. These values can then be utilized by each thread to calculate $f(x, y) \times V_{nm}^{*}(x, y)$. However, this results in a suboptimal performance due to the repetitive access of the values from the global memory in the for loop of the kernel. This is because the global memory is located off the processor chip, resulting in a prolonged latency in accessing the data. As illustrated in Figure 4, a GPU comprises several types of memory in addition to global memory, such as shared memory and registers. These two types of memory are located on chip and can be accessed with approximately 100 times greater speed than the global memory [16]. Shared memory can be accessed by all threads within a block, while each thread possesses its own individual access to registers. Furthermore, registers represent the quickest form of memory as they involve fewer instructions when compared to accessing data located in shared memory, which necessitates additional load operations [9].

Given that each thread is assigned to a single pixel, the values of $f(x, y)$, ρ and θ for each pixel can be stored in the registers of the respective thread, thereby enhancing access efficiency. The utilization of Kintner's recursive method also offers an additional advantage, as the repetition m is fixed for each invocation of the kernel. This means that when computing $f(x, y) \times$

$V^*_{nm}(x,y) = f(\rho,\theta)R_{nm}(\rho)cos(m\theta) - f(\rho,\theta)R_{nm}(\rho)sin(m\theta)j$, $f(\rho,\theta)cos(m\theta)$ of the real part and $f(\rho,\theta)sin(m\theta)$ of the imaginary part only need to be calculated once, and stored in the registers as demonstrated in Lines 10 and 11 of Algorithm 2. These values can then be reused for computation with different orders n throughout the current kernel invocation. For a single kernel invocation to compute the results of a given repetition m, there will be x different orders n, where $x = (\lfloor(n-1)/2\rfloor+1)*(\lfloor(n-1)/2\rfloor+1+((n-1)\%2))+\lfloor m/2\rfloor$. This means that $(x-1)$ cosine and sine functions, as well as $2(x-1)$ multiplications, can be omitted. Furthermore, the radial polynomial $R_{nm}(\rho)$ is stored in the registers of each thread and calculated through recursion. The results produced by each thread will either be placed in the global memory and the sum will be calculated through the application of the parallel reduction method, or be aggregated directly through the utilization of the atomic add operation. The discussion of both methods and an analysis of their performance will be presented in subsequent sections.

4.5 GPU Kernel Function Further Optimization

As depicted in lines 23 to 25 of Algorithm 2, it is the responsibility of all threads to calculate the coefficients K_1, K_2 and K_3 for their respective pixels. However, it should be noted that for pixels with the same order n and repetition m, the coefficients are identical. As a result, the calculation process for these coefficients is redundant and inefficient. A practical solution to eliminate this redundancy is to calculate these coefficients using the CPU and then transferring them to the GPU's global memory, where they can be accessed directly by all threads. This results in the coefficients of a given order n with repetition m being calculated only once, instead of N^2k^2 times. Despite this improvement, accessing these coefficients through the global memory is still expected to result in suboptimal performance due to the long access latency. Further optimization can be achieved by utilizing the GPU's constant memory.

Algorithm 2 Zernike Kernel Function to Compute $f(x,y) \times V_{nm}^{*}(x,y)$

```
1: function ZERNIKE_KERNEL(Znm_real[ ][ ], Znm_imag[ ][ ],
   f[ ], rho[ ], theta[ ], m, max_order, total_pixel_num)
2:     tid ← Current thread ID
3:     if tid < total_pixel_num then                                  ▷ Boundary check
4:         R_2 ← 0                                                    ▷ Used to store R_{n-2,m}(ρ)
5:         R_4 ← 0                                                    ▷ Used to store R_{n-4,m}(ρ)
6:         R ← 0                                                      ▷ Used to store R_{n,m}(ρ)
7:         rho_t ← rho[tid]                                           ▷ Load into register to increase access speed
8:         f_t ← f[tid]
9:         theta_t ← theta[tid]
10:        cos_f ← cos(m * theta_t) * f_t;
11:        sin_f ← −sin(m * theta_t) * f_t;
12:        for n ← m to max_order do
13:            nm_position ← (⌊(n − 1)/2⌋ + 1) * (⌊(n − 1)/2⌋ + 1 + ((n − 1)%2)) + ⌊m/2⌋
14:            if n == m then
15:                R_4 ← pow(rho_t, n)
16:                Znm_real[nm_position][tid] ← R_4 * cos_f
17:                Znm_imag[nm_position][tid] ← R_4 * sin_f
18:            else if n == (m + 2) then
19:                R_2 ← (m + 2) * R_4 * rho_t * rho_t − (m + 1) * R_4
20:                Znm_real[nm_position][tid] ← R_2 * cos_f
21:                Znm_imag[nm_position][tid] ← R_2 * sin_f
22:            else
23:                K_1 ← 4n(n−1) / ((n+m)(n−m))
24:                K_2 ← −2(n−1)(n^2−2n+m^2) / ((n+m)(n−m)(n−2))
25:                K_3 ← −n(n+m−2)(n−m−2) / ((n+m)(n−m)(n−2))
26:                R ← K_1 * rho_t * rho_t + K_2 * R_2 + K_3 * R_4
27:                Znm_real[nm_position][tid] ← R * cos_f
28:                Znm_imag[nm_position][tid] ← R * sin_f
29:                R_4 ← R_2
30:                R_2 ← R
31:            end if
32:        end for
33:    end if
34: end function
```

The GPU's constant memory, as the name implies, is designated as read-only, and therefore capable of storing data that will remain unchanged throughout the entire kernel execution process. Currently, the capacity of the constant memory typically limited to 65536 bytes for CUDA devices. It possesses unique features, such as cache and broadcasting, which allow for rapid access speeds, comparable to that of registers, if all threads simultaneously read data from the same address.

For Algorithm 2, the coefficients for a given order n and repetition m are constant, and all threads within a warp will simultaneously access the same elements of those coefficients. Therefore, the use of GPU's cached constant memory to store these coefficient elements is an optimal solution.

A coefficient element can be loaded into the cache and utilized by all threads within a warp. If 32 threads within a warp need to access the coefficients, 31 of them will be served by the cache, resulting in a reduction of nearly 97% in access to the global memory. Should the threads of other warps access the same coefficients, they can also be retrieved from the cache, further enhancing the access efficiency.

Furthermore, the cosine and sine functions in lines 10 and 11 of Algorithm 2 are computationally expensive. They can be substituted with CUDA's hardware trigonometry functions __cos() and __sin(). These hardware functions are executed on the Special Function Units (SFUs) of the CUDA devices, resulting in significantly improved performance compared to their equivalent CPU versions [9].

4.6 GPU Sum Operation

Another crucial aspect of the methodology involves obtaining the sum of the products generated by each thread through the kernel function described in Section 4.4. There are two methods to achieve this objective.

4.6.1 Strategy 1: Parallel Reduction

One approach is through the utilization of parallel reduction, which was employed by Xuan et al. in their study of GPU-based Zernike computations [26]. The products generated by each thread will initially be stored in the global memory. This results in a two-dimensional array that comprises the values of $f(x, y) \times V^*_{nm}(x, y)$ for all pixels on the unit disk, for orders n and repetitions m up to the specified maximum order. Given that T represents the total number of pixels on the unit disk, the two-dimensional array can be represented as follows:

$$\begin{bmatrix} f(x_0,y_0)V^*_{0,0}(x_0,y_0) & f(x_1,y_1)V^*_{0,0}(x_1,y_1) & \dots & f(x_{T-1},y_{T-1})V^*_{0,0}(x_{T-1},y_{T-1}) \\ f(x_0,y_0)V^*_{1,1}(x_0,y_0) & f(x_1,y_1)V^*_{1,1}(x_1,y_1) & \dots & f(x_{T-1},y_{T-1})V^*_{1,1}(x_{T-1},y_{T-1}) \\ f(x_0,y_0)V^*_{2,0}(x_0,y_0) & f(x_1,y_1)V^*_{2,0}(x_1,y_1) & \dots & f(x_{T-1},y_{T-1})V^*_{2,0}(x_{T-1},y_{T-1}) \\ f(x_0,y_0)V^*_{2,2}(x_0,y_0) & f(x_1,y_1)V^*_{2,2}(x_1,y_1) & \dots & f(x_{T-1},y_{T-1})V^*_{2,2}(x_{T-1},y_{T-1}) \\ \vdots & \vdots & \vdots & \vdots \\ f(x_0,y_0)V^*_{n,m}(x_0,y_0) & f(x_1,y_1)V^*_{n,m}(x_1,y_1) & \dots & f(x_{T-1},y_{T-1})V^*_{n,m}(x_{T-1},y_{T-1}) \end{bmatrix}. \tag{34}$$

Assuming we are addressing a specific row in Eq.(34), which can be denoted as:

$$\begin{bmatrix} Data[0] & Data[1] & \dots & Data[T-1] \end{bmatrix}. \tag{35}$$

The traditional sequential approach utilized by CPU to determine the sum of $Data[0] + Data[1] + ... + Data[T-1]$ involves iteratively visiting each element and incrementing the sum with its value. The computational complexity of this approach is $O(T)$. However, T is typically massive in size, leading to a substantial workload that results in significant processing time. Given an image with a size of 512×512 utilizes a 9×9 sub-region scheme and $\frac{\pi}{4}$ of the pixels fall within the unit disk, the total number $T = 512 \times 512 \times 9 \times 9 \times \frac{\pi}{4} \approx 16.7$ million.

By utilizing the parallel reduction algorithm, more computational efficiency can be achieved. The core concept of the algorithm is to partition the entire data elements into multiple pairs and allocate multiple threads to concurrently compute the sum of each pair. Each thread will be assigned responsibility for a specific data pair. As an illustration, consider the case where there are 16 elements to be computed, as depicted in Figure 6. In the initial step, 8 threads are utilized. Thread 0 calculates the sum of $Data[0]$ and $Data[8]$, and stores the result in $Data[0]$. The other threads perform similar operations simultaneously. As a result of this step, the number of

data elements requiring processing is reduced to 8. With only 4 steps total, the sum of all data elements can be obtained.

It is worth noting that, as previously mentioned, the global memory of GPU has a prolonged access latency. Directly implementing the parallel reduction algorithm on the global memory would result in suboptimal performance. To enhance efficiency, each thread block can be utilized to load a segment of the data into shared memory, and perform reduction on that segment within the shared memory. Additional optimization techniques for the parallel reduction algorithm can be found in [2].

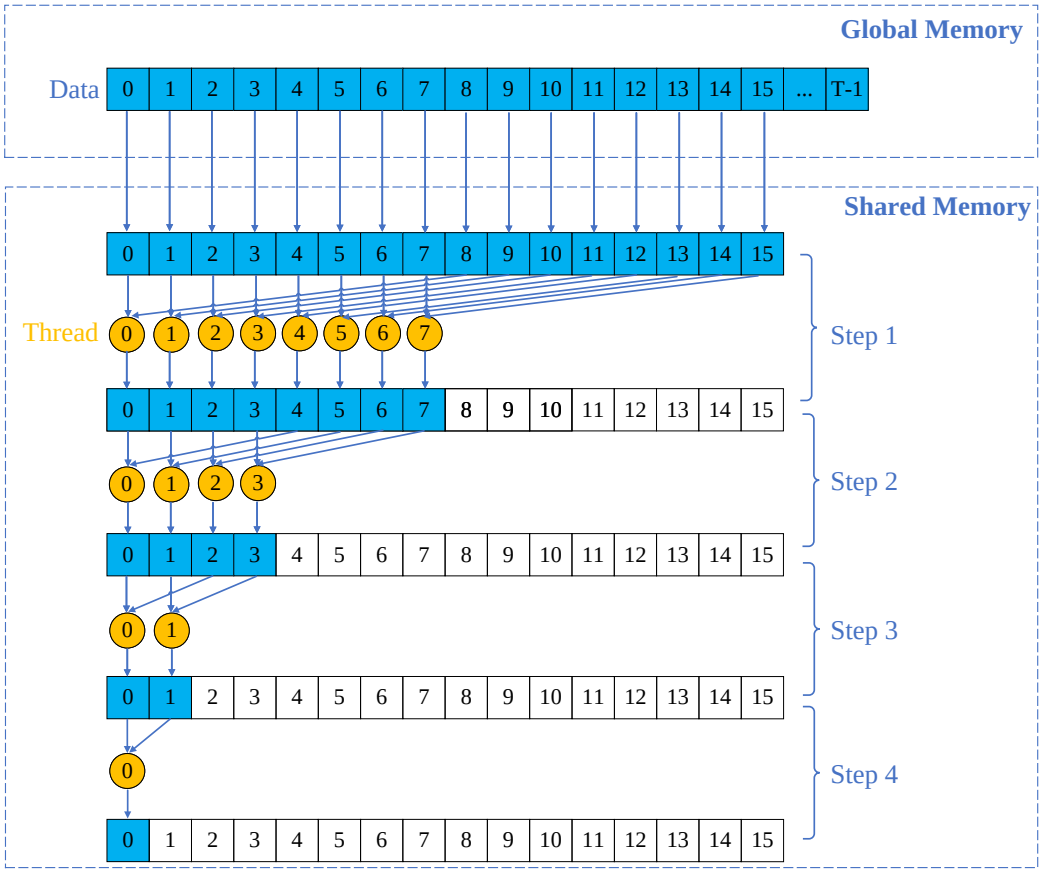


Figure 6: An example of parallel reduction algorithm

4.6.2 Strategy 2: Atomic Operation

Prior to executing the parallel reduction operation, it is imperative to ensure that all threads within the current kernel have completed their computations, which necessitates waiting for the return of the current kernel before calling another kernel to perform the parallel reduction. And this additional kernel call also incurs overhead.

Another drawback of utilizing the parallel reduction algorithm is that it requires a large number of memory-loading operations. As discussed in Section 4.4, each thread calculates the value $f(x, y) \times V_{nm}^*(x, y)$ for each pixel and stores the results in their respective registers. These values then need to be transferred to the GPU's global memory. Given a specific order n with repetition m, and assuming T pixels fall on the unit disk, then the number of elements that must be written to the global memory is T. In addition, the parallel reduction process entails $T + \frac{T}{2} + \frac{T}{4} + ... + 2 = 2(T - 1)$ read operations and $\frac{T}{2} + \frac{T}{4} + \frac{T}{8} + ... + 1 = T - 1$ write operations. Although the shared memory can increase the overall speed of the process, the memory loading overhead remains substantial.

To mitigate the extensive amount of memory reading and writing operations, the utilization of CUDA's atomic add operation can be employed. Previously, this operation had a subpar performance. However, with the advent of Fermi, the availability of an L2 cache significantly improves the performance by executing these operations at the L2 bank. By utilizing the atomic operation, the products computed by each thread can directly contribute to the sum. As depicted in Figure 7, an array designated to store the sum of various orders n and repetitions m will be allocated in the GPU's global memory. Upon any thread's computation of the value $f(x, y) \times V_{nm}^*(x, y)$ for a given order n with repetition m, it contributes the value to the corresponding sum using the atomic add operation.

The utilization of the atomic add operation instead of the simple addition of a value to the sum is necessitated by the possibility of race conditions

occurring during the process. Race conditions are a prevalent challenge in multi-thread programming, arising from the concurrent access attempts of multiple threads to a shared resource, potentially resulting in adverse outcomes. As an illustration, consider a scenario where two threads aim to increment the value of a shared memory location by one, and the current value at the location is 0. Under normal circumstances, the updated value should be 2. However, it is possible that both threads may retrieve the value of 0 simultaneously from the memory, then both increment the value by one, and write the result of 1 back to the memory. This would result in the incorrect outcome of the memory location being incremented by only one instead of two. In order to prevent race conditions, the use of atomic operations as provided by CUDA can be employed. As the name suggests, atomic operations ensure that the read-modify-write operations are indivisible, meaning that if one thread is currently executing a read-modify-write operation at a specific memory location, any other threads attempting to perform the same operation at that location will be suspended and queued, thereby guaranteeing the absence of race conditions. However, it should be noted that utilizing atomic operations comes at the cost of serialization, which can lead to decreased program performance.

It is also noteworthy that in our program, all data was stored using double-precision floating-point format to achieve the most accurate results. However, the atomic operations for double-precision data type are only supported by CUDA on GPU devices with a compute capability of 6.x or higher. This means that older GPU devices cannot utilize this method for computing the sum.

4.7 Symmetric Algorithm on GPU

The utilization of 4-symmetric and 8-symmetric algorithms can effectively decrease the computational workload to one-fourth and one-eighth, respectively, as discussed in Chapter 3. Nonetheless, their implementation on GPUs

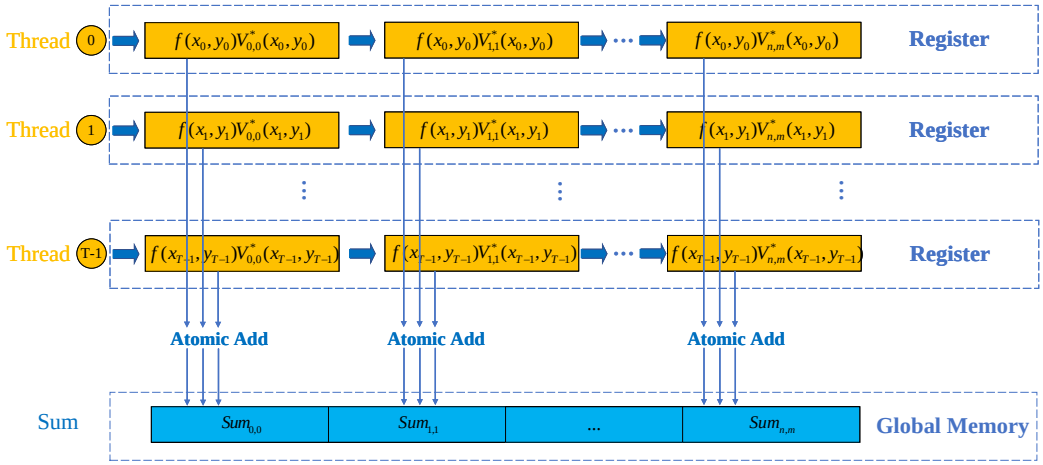


Figure 7: Illustration of the atomic add strategy

could be challenging due to the presence of the complex conditional statements, which can result in thread divergence issues, thereby negatively impacting performance. To resolve this issue, Xuan et al. [26] proposed a data re-layout method, which involves separating the symmetrical pixels of the original image into eight arrays and subsequently invoking the appropriate kernel function based on the repetition m. This approach resulted in a significant improvement in the acceleration rate. However, when computing a set of Zernike moments, the calculation of the pixel intensities as described in Eq. (32) and (33) becomes redundant. To further enhance the efficiency, we propose a novel image recomposition approach, wherein the calculation of pixel intensities is performed only once to complete the computation of the entire set of Zernike moments. Figure 8 presents a graphical representation of the recomposition process for the 4-symmetric algorithm.

The image recomposition is performed during the data preprocessing stage by a CPU. The original image is recomposed into four separate images, each containing one-quarter of the original image's pixels. In accordance with Eq. (29), the value of each pixel in the first image $F_1(x, y)$ can

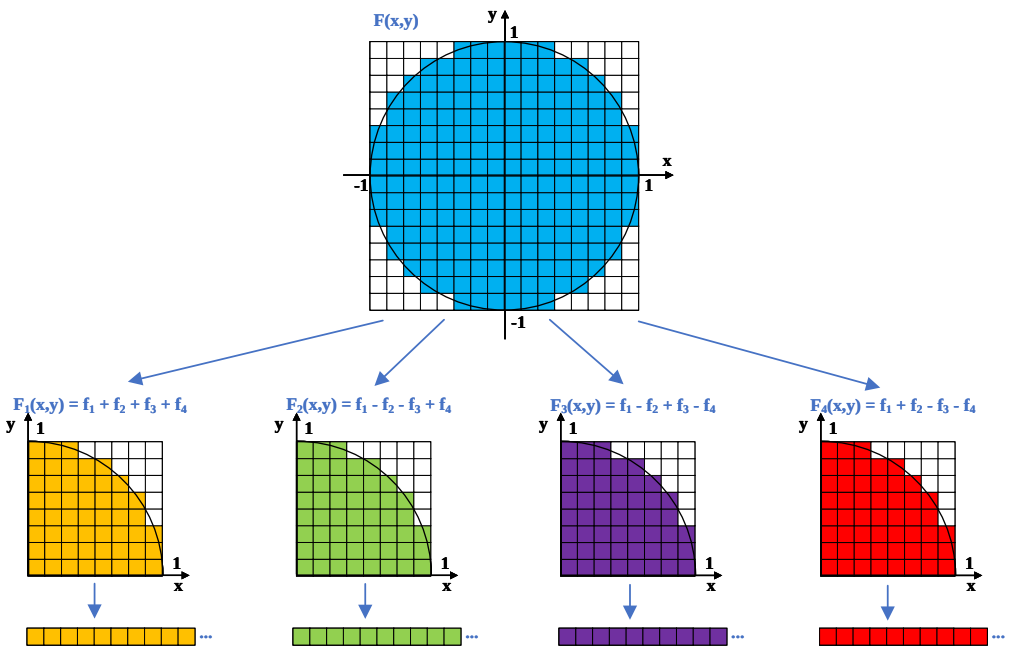


Figure 8: Illustration of the image re-composition method

be derived from its corresponding 4-symmetric pixels in the original image $F(x, y)$ through the formula $f_1 + f_2 + f_3 + f_4$. The second image $F_2(x, y)$ is formed by the formula $f_1 - f_2 - f_3 + f_4$. Similarly, the remaining two images $F_3(x, y)$ and $F_4(x, y)$ can be derived from Eq. (30). Furthermore, the pixels outside the unit disk are excluded to avoid the thread divergence problem. Subsequently, Eq. (28) can be rewritten as:

$$Z_{nm} = \frac{n+1}{\pi} \sum_{\substack{x_i^2+y_j^2 \leq 1, \\ x \geq 0, y \geq 0}} \sum R_{nm}(\rho)[f_m^r(x, y) - j f_m^i(x, y)] \Delta x \Delta y, \tag{36}$$

where

$$f_m^r(x, y) = \begin{cases} F_1(x, y) cos(m\theta), & m = 2k, \\ F_2(x, y) cos(m\theta), & m = 2k + 1. \end{cases} \tag{37}$$

$$f_m^i(x,y) = \begin{cases} F_3(x,y)sin(m\theta), & m = 2k, \\ F_4(x,y)sin(m\theta), & m = 2k+1. \end{cases} \tag{38}$$

The recomposed images can be fed into the kernel function according to different repetition m. If the result of dividing m by 2 leaves a remainder of 0, the images $F_1(x,y)$ and $F_3(x,y)$ will be input into the kernel, with $F_1(x,y)$ being designated to compute the real part, and $F_3(x,y)$ to compute the imaginary part. Conversely, if the division of m by 2 results in a remainder of 1, the images $F_2(x,y)$ and $F_4(x,y)$ will be input into the kernel. Similarly, the method can be altered to accommodate the implementation of the 8-symmetric algorithm on a GPU.

By employing this method, the calculation of the pixel intensity is required only once and can be applied to the entire set of Zernike moments. Furthermore, this approach obviates the need for the GPU kernel to carry out complex conditional statements and retrieve the symmetrical pixels' values from the original image data in an un-coalesced manner. Instead, the threads within the kernel function can consecutively fetch data from the recomposed images to perform computations in parallel, resulting in coalesced memory transactions and improved efficiency for the GPU kernel.

Chapter 5

Experimental Results

In order to assess the accuracy and efficiency of our proposed method, experiments were carried out to obtain Zernike moments from an image with varying maximum orders, and subsequently reconstruct the image using the computed moments. A test image of size 512×512 with 256 gray levels was employed in this research, which is depicted in Figure 9.

Figure 9: The testing Lena image sized at 512×512 with 256 gray levels

5.1 Computational Efficiency of the Proposed Method

In the course of our experiments, a program was developed utilizing CUDA C++ and subsequently evaluated on a desktop computer that was outfitted with an Intel i9 12900K CPU and an Nvidia GeForce RTX 4090 GPU that boasts 16384 CUDA cores. In the program, all data was stored utilizing the 64-bit double-precision floating-point format. The maximum order T of Zernike moments was systematically set to 100, 150, ..., and 500, respectively. Additionally, the k value of the sub-region scheme was varied from 1 to 9. Table 3 presents the total computation time, including the data transfer and GPU computation time, for the calculation of Zernike moments with varying

maximum orders and k values.

As can be observed from Table 3, without the application of the $k \times k$ sub-region scheme (i.e. $k = 1$), the computation of Zernike moments with maximum order up to 500 took less than 0.5 seconds, surpassing the performance of the other fast computation methods documented in the literature. After applying the $k \times k$ sub-region scheme, an augmented computational workload resulted in a corresponding increase in the computational time, yet remained within seconds.

Table 3: The whole computation time (in seconds) of Zernike moments with different maximum orders T and k values

$k \backslash T$	100	150	200	250	300	350	400	450	500
1	0.114	0.145	0.184	0.219	0.248	0.304	0.357	0.404	0.461
3	0.340	0.371	0.390	0.562	0.647	0.901	1.009	1.322	1.423
5	0.596	0.689	0.774	1.218	1.387	1.893	2.201	2.899	3.187
7	0.724	1.103	1.397	2.072	2.474	3.395	4.037	5.161	5.754
9	1.106	1.783	2.240	3.273	3.999	5.296	6.340	8.151	9.267

5.2 Computational Accuracy of the Proposed Method

Zernike moments computed through the original testing image with different maximum orders and k values were subsequently used for image reconstruction. A selection of the reconstructed images is presented in Figure 10.

To analyze the computational accuracy, the Peak Signal to Noise Ratio (PSNR) measurement was utilized to assess the quality of the reconstruction. A higher PSNR value for the reconstructed image indicates a closer resemblance to the original image. The PSNR is defined as:

$$PSNR = 10 \log_{10}(\frac{G_{max}^2}{MSE}), \tag{39}$$

where G_{max} is the maximum gray level of the original image (in our experiments $G_{max} = 255$) and MSE is the Mean Square Error between the original

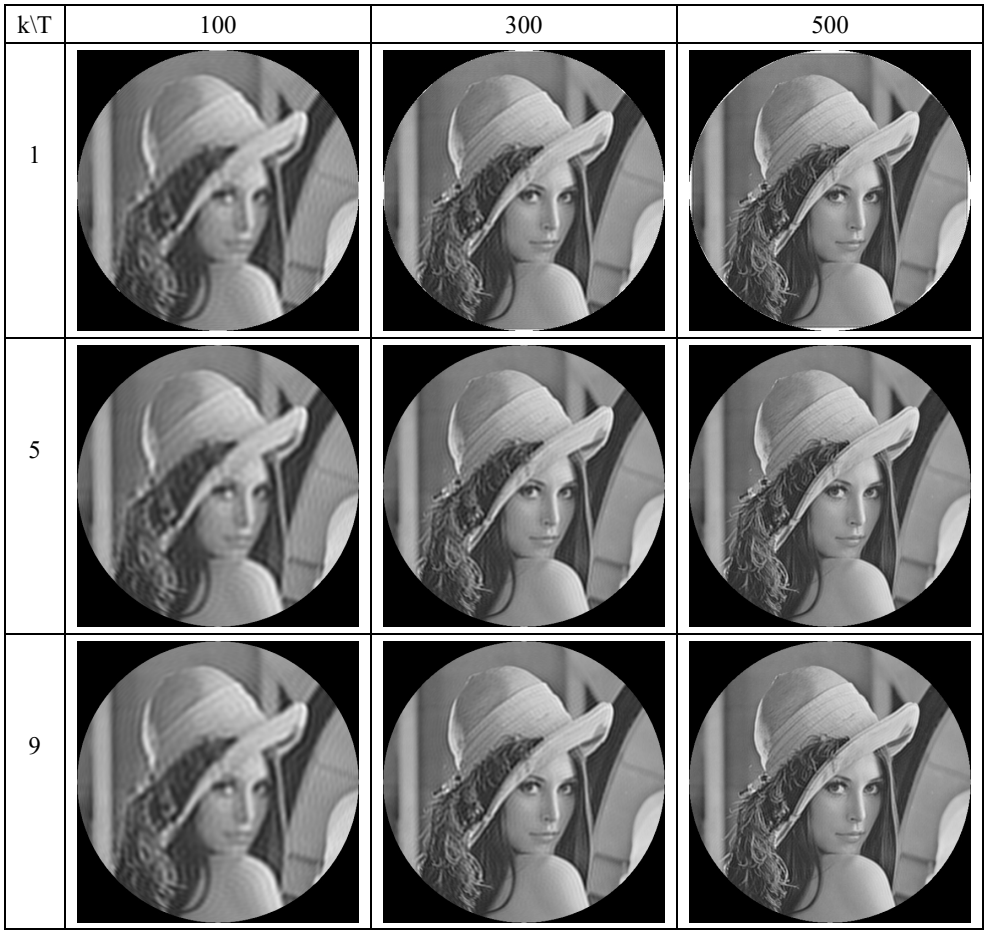


Figure 10: A selection of the reconstructed images obtained from Zernike moments with different maximum order T and k values

image $f(x, y)$ and the reconstructed image $\hat{f}(x, y)$:

$$MSE = \frac{1}{MN} \sum_{i=1}^{M} \sum_{j=1}^{N} [f(x_i, y_j) - \hat{f}(x_i, y_j)]^2, \tag{40}$$

where M and N are the width and height of the image respectively. However, it should be noted that since Zernike moments are defined over the unit disk, any pixels that fall outside of the unit disk are not used in the calculation

of Zernike moments, nor do they participate in the reconstruction process. Therefore, when calculating the PSNR, these pixels must also be excluded. The PSNR values of the reconstructed images from our experiments are presented in Table 4.

Table 4: The PSNR values of the reconstructed images obtained from Zernike moments with varying maximum order T and k values

$k\backslash T$	100	150	200	250	300	350	400	450	500
1	26.87	28.14	28.17	27.26	25.94	24.68	23.36	21.98	20.75
3	27.06	29.00	30.69	32.20	33.35	34.48	34.89	35.12	35.25
5	27.06	29.01	30.71	32.33	33.82	35.28	36.64	37.63	38.15
7	27.06	29.01	30.71	32.33	33.90	35.29	36.76	37.90	39.19
9	27.06	29.01	30.71	32.33	33.90	35.29	36.76	38.04	39.20

As evident from Figure 10, when Zernike moments with a higher maximum order were utilized, the reconstructed images displayed improved clarity, as they incorporated greater details from the original image. However, prior to the application of the $k \times k$ sub-region scheme(i.e., $k = 1$), an increase in the maximum order led to the emergence of white image distortion around the rim of the reconstructed images, resulting in a marked decrease in the PSNR values, as depicted in Table 4. The occurrence of the image distortion was the result of the accumulation of the approximation errors during the calculation of Zernike moments, as discussed in Section 3.1. The implementation of the $k \times k$ sub-region scheme allowed for the obtaining of more accurate Zernike moments, leading to the elimination of the image distortion phenomenon. For high-order moments, such as 400 and 500, the use of the 3×3 sub-region scheme significantly improved the quality of the reconstructed image. And the utilization of higher k values was also observed to result in higher PSNR values, making it particularly useful in situations where high precision results are required. When the maximum order of Zernike moments was 500 and the 9×9 sub-region scheme was used, the PSNR value of the reconstructed image was recorded as 39.20 dB.

5.3 Comparison Between Different GPU Optimization Strategies

This subsection presents the performance of different GPU optimization strategies discussed in Chapter 4. The different strategies are listed in Table 5. The same test image was utilized to conduct the performance comparison. Table 6 displays the computation time of S1, S2, and S3 for Zernike moments with maximum orders ranging from 100 to 500 without applying the sub-region scheme.

Table 5: Different GPU optimization strategies.

Strategy	Description
S1	Using parallel reduction
S2	Using atomic operation
S3	S2 + Using constant memory and hardware functions
S4	S3 + Using four symmetric strategy
S5	S3 + Using eight symmetric strategy

Table 6: The computation time(in seconds) of S1, S2, and S3 for Zernike moments with maximum orders ranging from 100 to 500 without applying the sub-region scheme

$Strategy \backslash T$	100	200	300	400	500
S1	2.072	7.533	17.068	29.699	46.357
S2	0.276	0.492	0.892	1.413	2.108
S3	0.266	0.444	0.691	1.033	1.449

As discussed in Section 4.6.2, the parallel reduction kernel can be invoked once all threads within a given kernel have completed their calculation of the values $f(x, y) \times V^*_{nm}(x, y)$ for their corresponding pixels. This kernel incurs not only a launch overhead, but also requires a substantial amount of memory read and write operations. The comparison of the computation times of S1 and S2, as presented in Table 6, demonstrates that the use of the atomic

add operation to calculate the sum for all pixels significantly enhances the computation efficiency in comparison to the parallel reduction algorithm.

Furthermore, a comparison of the computation times of S2 and S3 indicates that the utilization of the constant memory to store the coefficients K_1, K_2 and K_3, in conjunction with the hardware trigonometry functions provided by CUDA to compute the sine and cosine functions, resulted in further improvement of the computation efficiency. This improvement became more pronounced when higher maximum orders were used, as they involved more computational workload.

In order to evaluate the efficacy of the proposed image recomposition method for the implementation of the 4-symmetric and 8-symmetric algorithms on a GPU, experiments were conducted to calculate Zernike moments with varying maximum orders and distinct sub-region schemes, using the methods S3, S4, and S5. The results of these experiments are documented in Tables 7, 8, and 9.

Table 7: The computation time(in seconds) of S3, S4, and S5 for Zernike moments with maximum orders ranging from 100 to 500 without applying the sub-region scheme

$Strategy \backslash T$	100	200	300	400	500
S3	0.266	0.444	0.691	1.033	1.449
S4	0.153	0.214	0.331	0.450	0.591
S5	0.114	0.184	0.248	0.357	0.461

Table 8: The computation time(in seconds) of S3, S4, and S5 for Zernike moments with maximum orders ranging from 100 to 500 using the 5×5 sub-region scheme

$Strategy \backslash T$	100	200	300	400	500
S3	2.749	5.268	9.196	15.400	22.223
S4	0.832	1.496	2.623	4.324	6.039
S5	0.596	0.774	1.387	2.201	3.187

Table 9: The computation time(in seconds) of S3, S4, and S5 for Zernike moments with maximum orders ranging from 100 to 500 using the 9×9 sub-region scheme

$Strategy \backslash T$	100	200	300	400	500
S3	8.278	16.395	29.473	48.474	71.593
S4	2.233	4.472	8.087	12.671	18.450
S5	1.106	2.240	3.999	6.340	9.267

According to Table 7, the proposed method for implementing the 4-symmetric and 8-symmetric algorithms on the GPU demonstrated a notable improvement in computation efficiency. However, as the sub-region scheme was not utilized in this scenario, which led to a limited computational workload, the time for data preprocessing and memory transactions between the CPU and GPU made up a substantial portion of the total elapsed time. As a result, the computation times for S4 and S5 did not experience a $4\times$ and $8\times$ speed-up compared to S3, respectively. On the other hand, in Tables 8 and 9, with the implementation of the 5×5 and 9×9 sub-region schemes, a significantly larger computational workload was required. The time for data preprocessing and memory transactions became negligible in comparison to the total elapsed time, thus S4 and S5 experienced nearly 4-fold and 8-fold speed-up compared to S3. This confirms the validity of the proposed image recomposition method for the GPU implementation of the 4-symmetric and 8-symmetric algorithms.

5.4 Comparison With the CPU-Based Implementation

In order to determine the superiority of our proposed GPU-accelerated implementation over the CPU-based baseline implementation, which employs the Kintner's fast method in conjunction with the eight-symmetric method, an experiment was performed using the same test image. Table 10 presents the computation time of Zernike moments with varying maximum order T and

k values as calculated on both the CPU and GPU, enabling a comparative analysis to be performed.

Table 10: The computation time(in seconds) of Zernike moments with varying maximum order T and k values as calculated on both the CPU and GPU

$k\backslash T$	100		300		500	
	CPU	GPU	CPU	GPU	CPU	GPU
1	1.030	0.114	7.329	0.248	19.289	0.461
5	24.682	0.596	180.090	1.387	480.928	3.187
9	79.547	1.106	583.205	3.999	1549.963	9.267

As can be observed from Table 10, prior to dividing the original pixels into $k \times k$ sub-pixels, the GPU-based method achieved a modest speed-up compared to the baseline CPU method, with the rate increasing as the maximum order increased. When computing Zernike moments with a maximum order of 100, the speed-up rate was $\frac{1.030}{0.114} \approx 9.04\times$. For 500-order moments, the speed-up rate reached $\frac{19.289}{0.461} \approx 41.84\times$. Additionally, with the application of the $k \times k$ sub-region scheme, which required a significantly greater computational workload, the benefits of using GPU became even more pronounced. It took the CPU approximately 26 minutes to calculate 500-order moments using the 9×9 scheme, while the GPU only required approximately 9 seconds, yielding a speed-up rate of $\frac{1549.963}{9.267} \approx 167.26\times$.

5.5 Experiments on Additional Images

In order to thoroughly validate the proposed method, four additional testing images, as depicted in Figure 11, were utilized to conduct experiments. All of the images had a size of 512×512 with 256 gray levels. The total computation time of Zernike moments for the testing images is presented in Table 11, while the PSNR values of the reconstructed images from those moments are shown in Table 12.

As evidenced by Table 11, the computation time of moments with identi-

Figure 11: Four additional testing images: Cameraman, House, Peppers, and Tiffany

Table 11: The whole computation time(in seconds) of Zernike moments of the four additional testing images with different maximum orders T and k values

	Cameraman		House		Peppers		Tiffany	
$k\backslash T$	300	500	300	500	300	500	300	500
1	0.265	0.478	0.277	0.457	0.243	0.483	0.259	0.463
5	1.408	3.207	1.379	3.163	1.363	3.216	1.376	3.189
9	3.968	9.294	4.016	9.231	3.974	9.283	4.025	9.301

cal maximum order and k value for various images with equivalent dimensions were nearly identical. In terms of the quality of the reconstructed images from these moments, Table 12 reveals that higher PSNR values can be achieved by using greater maximum order and k values. However, the PSNR values for distinct images reconstructed from moments with the same maximum order and k value may differ, owing to variations in image content.

Table 12: The PSNR values of the reconstructed images from Zernike moments with varying maximum order T and k values

	Cameraman		House		Peppers		Tiffany	
$k\backslash T$	300	500	300	500	300	500	300	500
1	26.55	21.62	27.94	22.20	26.32	21.13	25.85	20.89
5	34.27	40.83	40.71	42.81	33.87	37.17	31.43	34.99
9	34.39	42.49	41.33	43.83	33.93	37.39	31.47	35.23

Chapter 6

Leaf Recognition With Zernike Moments

As elucidated in Chapter 2, Zernike moments possess several advanced properties, such as the ability to represent images with minimal redundant information and invariance to rotations and reflections. Therefore, they can be utilized as image features to perform image classification. In this Chapter, a method is proposed for leaf recognition that employs Zernike moments as image features and the k-Nearest Neighbors algorithm (k-NN) as a classifier. The method is depicted in Fig. 12. Initially, the leaf images undergo preprocessing, followed by computation of Zernike moments from the processed images. These moments are then utilized to construct the feature vectors of the images. Subsequently, the feature vectors are supplied to the k-NN classifier for training or classification purposes.

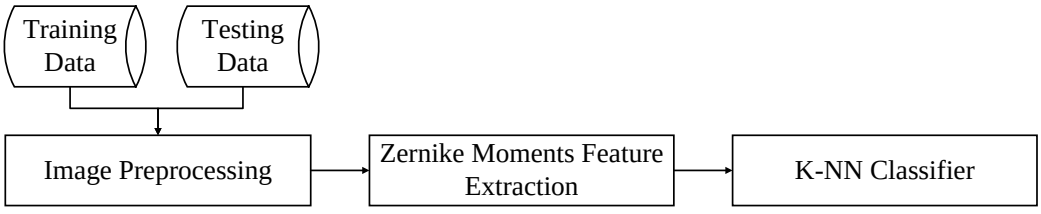


Figure 12: The proposed method for leaf recognition

6.1 Image Preprocessing

Prior to extracting the image features, it is necessary to preprocess the input image. This entails scaling the image and positioning the leaf at its center.

The initial step in the image preprocessing stage is to convert the input RGB image to a grayscale image, which is then resized to a suitable dimension to decrease the computational workload. In this study, the size is set to 256×256. To position the leaf in the center of the image, the centroid of the leaf must be determined. Otsu's method is employed to create a binary copy of the image, allowing the shape of the leaf to be obtained. Subsequently, the centroid can be obtained using the regular moments of the image. The regular moments of a two-dimensional image $f(x, y)$ are defined as the projections of $f(x, y)$ onto the monomial $x^p y^q$:

$$m_{pq} = \sum_x \sum_y x^p y^q f(x, y), \tag{41}$$

where p denotes the order of x, and q represents the order of y.

The centroid $(\overline{x}, \overline{y})$ can be obtained by:

$$\overline{x} = \frac{m_{10}}{m_{00}}, \overline{y} = \frac{m_{01}}{m_{00}}. \tag{42}$$

Assuming the input image $f(x, y)$ has a width of w and a height of h, the centered image $g(x, y)$ can be obtained as follows:

$$g(x, y) = f(x + [\frac{w}{2} - \overline{x}], y + [\frac{h}{2} - \overline{y}]). \tag{43}$$

Subsequently, the leaf is scaled using the formula:

$$g(x, y) = f(\frac{x}{\sqrt{\frac{\beta}{m_{00}}}}, \frac{y}{\sqrt{\frac{\beta}{m_{00}}}}), \tag{44}$$

so that the leaf of each image has a uniform area β, which is set to 10000

in this study. Following this, a bitwise operation is performed on the binary image and the grayscale image, allowing only the leaf to be retained while the background is set to a value of 0, which is black.

Fig. 13 illustrates an example of the image preprocessing process, where three input images of different types of leaves have undergone preprocessing and produced their respective output images. These preprocessed images can then be utilized to compute their Zernike moments as features.

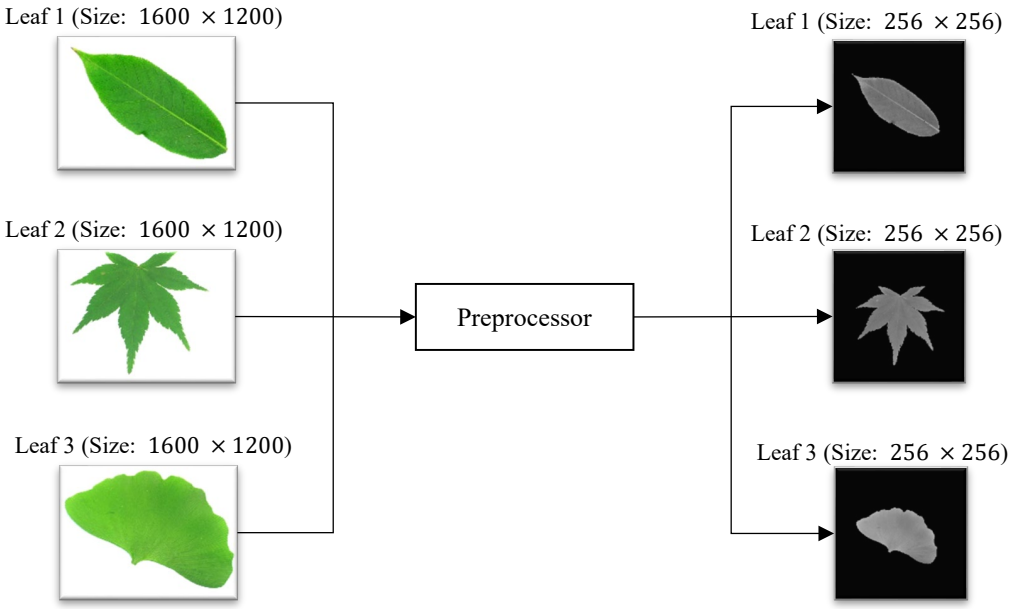


Figure 13: An example of the image preprocessing process, where three input images of different types of leaves have undergone preprocessing and produced their respective output images

6.2 K-Nearest Neighbors Algorithm

The Zernike moments obtained from each leaf image can be utilized to generate a feature vector: $\vec{v} = (v_1, v_2, v_3, ..., v_l) = (|Z_{00}|, |Z_{11}|, |Z_{20}|, ..., |Z_{n_{max}, n_{max}}|)$, where $|Z_{nm}|$ denotes the magnitude of Zernike moment of order n with rep-

etition m, and n_{max} is the chosen maximum order. Subsequently, machine learning algorithms can be employed on these feature vectors for the purposes of learning and recognition. In this study, the k-nearest neighbors algorithm (k-NN) has been utilized.

The k-NN algorithm is a non-parametric supervised learning algorithm that formulates predictions based on the distance between the object to be classified and all the available samples in the training set. To be specific, when the algorithm is provided with a new input object, it searches for the k closest objects in the training set and identifies the majority class of those k nearest neighbors as the predicted class for the new object.

The distance between any two objects can be determined by employing different methods. In this study, Euclidean distance has been utilized. Assume that two objects X and Y possess feature vectors of $\vec{X} = (x_1, x_2, ..., x_n)$ and $\vec{Y} = (y_1, y_2, ..., y_n)$, the Euclidean distance between them is defined as follows:

$$D(\vec{X}, \vec{Y}) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}. \tag{45}$$

6.3 Experiment Results

To assess the proposed method, several databases can be utilized, such as the Flavia dataset, Swedish dataset, ICL dataset, and ImageCLEF dataset [15]. In this study, the Flavia dataset was selected, which is frequently utilized by numerous researchers for plant leaf recognition. In [24], a Probabilistic Neural Network (PNN) approach was tested on the Flavia dataset and achieved a success rate of 90%. In [14], an approach using SVM as a classifier and Hu moments and uniform local binary pattern histogram parameters as features was tested on the Flavia dataset, which achieved an accuracy of 94.13%.

The dataset comprises 1800 images of leaves from 32 distinct species. The images in the dataset solely include the blades of the leaves, without any petioles, and have a resolution of 1600 × 1200. Some examples of the

leaves from this dataset are presented in Fig. 14.

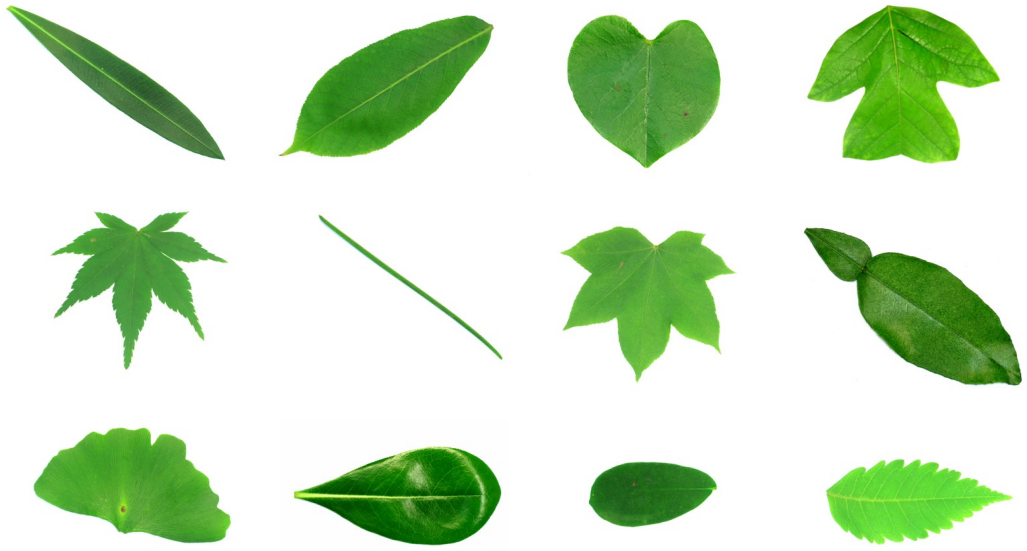


Figure 14: Some examples of the leaves from the Flavia dataset

In this study, a total of 50 samples were used for each of the 32 species from the Flavia dataset to perform the leaf recognition experiment. For each species, 40 samples were allocated for training and 10 samples for testing. Zernike moments with varying maximum orders were employed to evaluate the classification ability, and the results of the experiment are presented in Table 13. The maximum order of the Zernike moments was set to 6, 8, 10, 15, 20, 30, 40, and 50, respectively, which corresponded to the usage of 16, 25, 36, 72, 121, 256, 441, and 676 Zernike moments as features for the leaf images. Furthermore, the 5-NN algorithm was employed as the classifier.

As presented in Table 13, it can be observed that the recognition accuracy of the leaf classification task increased as the maximum order of the Zernike moments used as features was increased. When the maximum order was set to 6 or 8, the classification accuracy was low as the information captured in the Zernike moments was not sufficient to distinguish between the different

types of leaves. With the use of higher order moments, more details were taken into account, leading to improved recognition ability. The highest recognition accuracy of 97.19% was obtained when Zernike moments with a maximum order of 30 were utilized. However, as the maximum order was further increased, the high-dimensional features would lead to the curse of dimensionality and hence, a reduction in the recognition accuracy.

Table 13: The test results of the experiment to evaluate the classification ability of utilizing Zernike moments with varying maximum orders

Experiment	Maximum order of Zernike moments	Number of Zernike moments (Features)	Accuracy
1	6	16	64.38%
2	8	25	75.00%
3	10	36	79.06%
4	15	72	85.00%
5	20	121	89.06%
6	30	256	97.19%
7	40	441	95.31%
8	50	676	94.06%

Chapter 7

Conclusions

In this research, we examined the existing algorithms for fast computation of Zernike moments. Building on the radial polynomials' recursive relationship and symmetry property, we proposed a novel method for efficiently computing high-order Zernike moments. By utilizing thousands of CUDA cores on a GPU to compute intermediate results for image pixels simultaneously, our method demonstrates greater computation efficiency than traditional CPU-based methods. Additionally, we implemented the $k \times k$ sub-region scheme, so that more accurate moments can be obtained, which is useful for those situations where high-precision results are required.

Our implementation differs from existing GPU implementations that employ the direct method, as it is capable of computing high-order moments. Additionally, we have proposed a method to effectively utilize symmetric algorithms on the GPU. Our experimental results have demonstrated that this method can achieve almost four-fold and eight-fold speed-up rates. Regarding the summation of intermediate results, existing GPU methods have relied on the parallel reduction algorithm. We have explored the use of the atomic add operation for this purpose and achieved better performance. Nevertheless, given that atomic operations can result in serialization and performance limitations, there is a necessity for further research to explore more effective solutions aimed at enhancing the overall performance. Furthermore, we have also implemented diverse optimizations, such as the utilization of registers, constant memories, and the GPU's hardware trigonometry functions, with the aim of further augmenting the performance.

To evaluate the efficiency of our proposed algorithm, several testing images sized at 512×512 were employed to compute Zernike moments. The results demonstrated that it took less than 0.5 seconds to compute Zernike

moments with a maximum order of 500, which makes the real-time application of high-order Zernike moments become feasible. To assess the accuracy of our proposed algorithm, we used the computed Zernike moments to perform image reconstruction and evaluated the quality of the reconstruction using the PSNR measurement. The experiments showed that without the $k \times k$ sub-region scheme, the reconstructed image suffered from undesirable distortion, particularly when the maximum order was high. However, the introduction of the $k \times k$ scheme significantly improved computational accuracy and eliminated image distortion in the reconstructed images. Specifically, when using the 9×9 sub-region scheme to compute Zernike moments with a maximum order of 500, the PSNR value of the reconstructed Lena image reached 39.20 dB, and the total computation time was 9.267 seconds.

Furthermore, a method was proposed for the recognition of leaves, which leverages Zernike moments as image features and k-nearest neighbors as the classifier. The proposed approach was subsequently evaluated using the Flavia dataset. The results revealed that the use of low-order moments resulted in poor performance, as they contain insufficient information to distinguish between different types of leaves. When Zernike moments with a maximum order of 30 were employed, the best recognition ability was attained, with an accuracy of 97.19%. However, increasing the maximum order further led to the curse of dimensionality, resulting in decreased recognition performance. To tackle this issue, future research will necessitate data selection to identify moments with greater variances, thereby facilitating the improvement of both efficiency and performance.

Appendix A

Source Code of the GPU Implementation

This chapter provides the CUDA C++ source code snippets for the GPU implementation. To compute Zernike moments, the initial step involves invoking the function *Zernike_k_subregion*() for data preprocessing, followed by repeated invocations of the GPU kernel functions *calculate_Zernike_m*02() and *calculate_Zernike_m*13() based on varying repetitions of m.

Snippet 1: Function to compute Zernike moments

```
zernike_moments Zernike_k_subregion(Mat& img, int max_order,
    int k)
{
  int width = img.cols;
  int width_k = width * k;
  int mid = width / 2;
  int total_pixel_num = 0; // Those pixels out of the unit
   disk will not be counted

  // Store image pixels, and convert range from [0, 255] ->
   [0.0 - 1.0]
  // each original pixel will be seperated into k pixels and
   have the same value
  double* f_m0_real = new double[width_k * width_k];
  double* f_m0_real_d; // on GPU memory
  double* f_m0_imag = new double[width_k * width_k];
  double* f_m0_imag_d; // on GPU memory

  double* f_m1_real_cos = new double[width_k * width_k];
  double* f_m1_real_cos_d; // on GPU memory
  double* f_m1_real_sin = new double[width_k * width_k];
  double* f_m1_real_sin_d; // on GPU memory
  double* f_m1_imag_sin = new double[width_k * width_k];
  double* f_m1_imag_sin_d; // on GPU memory
```

```
21  double* f_m1_imag_cos = new double[width_k * width_k];
22  double* f_m1_imag_cos_d; // on GPU memory
23
24  double* f_m2_real = new double[width_k * width_k];
25  double* f_m2_real_d; // on GPU memory
26  double* f_m2_imag = new double[width_k * width_k];
27  double* f_m2_imag_d; // on GPU memory
28
29  double* f_m3_real_cos = new double[width_k * width_k];
30  double* f_m3_real_cos_d; // on GPU memory
31  double* f_m3_real_sin = new double[width_k * width_k];
32  double* f_m3_real_sin_d; // on GPU memory
33  double* f_m3_imag_sin = new double[width_k * width_k];
34  double* f_m3_imag_sin_d; // on GPU memory
35  double* f_m3_imag_cos = new double[width_k * width_k];
36  double* f_m3_imag_cos_d; // on GPU memory
37
38  //Calculate Rho and Theta for each pixel
39  double* rho = new double[width_k * width_k]; //Store each
     pixel's rho
40  double* rho_d; // on GPU memory
41  double* theta = new double[width_k * width_k]; //Store each
       pixel's theta
42  double* theta_d; // on GPU memory
43
44  double delta_xy = 2.0 / width_k; //width between sub-pixels
45
46  for (int j = 0; j < width / 2; j++) // j_row
47  {
48    for (int i = mid - 1 - j; i < mid; i++) // i_col
49    {
50      double x = ((2 * (i + mid) + 1 - double(width)) /
     double(width)) - (floor(k / 2) * delta_xy);
51      double y = ((double(width) - 2 * j - 1) / double(width)
     ) - (floor(k / 2) * delta_xy);
52
53      bool is_on_unit_disk = true;
```

```
54
55      for (int a = 0; (a < k) && is_on_unit_disk; a++)
56      {
57        for (int b = 0; (b < k) && is_on_unit_disk; b++)
58        {
59          double xx = x + a * delta_xy;
60          double yy = y + b * delta_xy;
61          double rho_temp = sqrt(pow(xx, 2) + pow(yy, 2));
62          if (rho_temp > 1.0)
63          {
64            is_on_unit_disk = false;
65            break;
66          }
67        }
68      }
69
70      if (is_on_unit_disk)
71      {
72        // Use eight symmetric method
73        double f1, f2, f3, f4, f5, f6, f7, f8;
74
75        if (i + j == mid - 1) // For those points on diagonal
    , only 4-symmetric
76        {
77          f1 = img.at<unsigned char>(j, mid + i) / 255.0;
78          f2 = 0;
79          f3 = 0;
80          f4 = img.at<unsigned char>(j, mid - 1 - i) / 255.0;
81          f5 = img.at<unsigned char>(width - 1 - j, mid - 1 -
     i) / 255.0;
82          f6 = 0;
83          f7 = 0;
84          f8 = img.at<unsigned char>(width - 1 - j, mid + i)
    / 255.0;
85        }
86        else // the other points have 8-symmetric
87        {
```

```
88          f1 = img.at<unsigned char>(j, mid + i) / 255.0;
89          f2 = img.at<unsigned char>(mid - 1 - i, width - 1 -
     j) / 255.0;
90          f3 = img.at<unsigned char>(mid - 1 - i, j) / 255.0;
91          f4 = img.at<unsigned char>(j, mid - 1 - i) / 255.0;
92          f5 = img.at<unsigned char>(width - 1 - j, mid - 1 -
     i) / 255.0;
93          f6 = img.at<unsigned char>(mid + i, j) / 255.0;
94          f7 = img.at<unsigned char>(mid + i, width - 1 - j)
    / 255.0;
95          f8 = img.at<unsigned char>(width - 1 - j, mid + i)
    / 255.0;
96        }
97
98        double f_m0_real_tmp = f1 + f2 + f3 + f4 + f5 + f6 +
    f7 + f8;
99        double f_m0_imag_tmp = f1 - f2 + f3 - f4 + f5 - f6 +
    f7 - f8;
100
101       double f_m1_real_cos_tmp = f1 - f4 - f5 + f8;
102       double f_m1_real_sin_tmp = f2 - f3 - f6 + f7;
103       double f_m1_imag_sin_tmp = f1 + f4 - f5 - f8;
104       double f_m1_imag_cos_tmp = f2 + f3 - f6 - f7;
105
106       double f_m2_real_tmp = f1 - f2 - f3 + f4 + f5 - f6 -
    f7 + f8;
107       double f_m2_imag_tmp = f1 + f2 - f3 - f4 + f5 + f6 -
    f7 - f8;
108
109       double f_m3_real_cos_tmp = f1 - f4 - f5 + f8;
110       double f_m3_real_sin_tmp = -f2 + f3 + f6 - f7;
111       double f_m3_imag_sin_tmp = f1 + f4 - f5 - f8;
112       double f_m3_imag_cos_tmp = -f2 - f3 + f6 + f7;
113
114       for (int a = 0; a < k; a++)
115       {
116         for (int b = 0; b < k; b++)
```

```
117           {
118             double xx = x + a * delta_xy;
119             double yy = y + b * delta_xy;
120             double rho_temp = sqrt(pow(xx, 2) + pow(yy, 2));
121 
122             f_m0_real[total_pixel_num] = f_m0_real_tmp;
123             f_m0_imag[total_pixel_num] = f_m0_imag_tmp;
124 
125             f_m1_real_cos[total_pixel_num] =
    f_m1_real_cos_tmp;
126             f_m1_real_sin[total_pixel_num] =
    f_m1_real_sin_tmp;
127             f_m1_imag_sin[total_pixel_num] =
    f_m1_imag_sin_tmp;
128             f_m1_imag_cos[total_pixel_num] =
    f_m1_imag_cos_tmp;
129 
130             f_m2_real[total_pixel_num] = f_m2_real_tmp;
131             f_m2_imag[total_pixel_num] = f_m2_imag_tmp;
132 
133             f_m3_real_cos[total_pixel_num] =
    f_m3_real_cos_tmp;
134             f_m3_real_sin[total_pixel_num] =
    f_m3_real_sin_tmp;
135             f_m3_imag_sin[total_pixel_num] =
    f_m3_imag_sin_tmp;
136             f_m3_imag_cos[total_pixel_num] =
    f_m3_imag_cos_tmp;
137 
138             rho[total_pixel_num] = rho_temp;
139             theta[total_pixel_num] = atan2(yy, xx);
140             total_pixel_num++;
141           }
142         }
143       }
144     }
145   }
```

```
146
147  cudaMalloc((void**)&f_m0_real_d, sizeof(double) *
      total_pixel_num);
148  cudaMemcpy(f_m0_real_d, f_m0_real, sizeof(double) *
      total_pixel_num, cudaMemcpyHostToDevice);
149  cudaMalloc((void**)&f_m0_imag_d, sizeof(double) *
      total_pixel_num);
150  cudaMemcpy(f_m0_imag_d, f_m0_imag, sizeof(double) *
      total_pixel_num, cudaMemcpyHostToDevice);
151
152  cudaMalloc((void**)&f_m1_real_cos_d, sizeof(double) *
      total_pixel_num);
153  cudaMemcpy(f_m1_real_cos_d, f_m1_real_cos, sizeof(double) *
       total_pixel_num, cudaMemcpyHostToDevice);
154  cudaMalloc((void**)&f_m1_real_sin_d, sizeof(double) *
      total_pixel_num);
155  cudaMemcpy(f_m1_real_sin_d, f_m1_real_sin, sizeof(double) *
       total_pixel_num, cudaMemcpyHostToDevice);
156  cudaMalloc((void**)&f_m1_imag_sin_d, sizeof(double) *
      total_pixel_num);
157  cudaMemcpy(f_m1_imag_sin_d, f_m1_imag_sin, sizeof(double) *
       total_pixel_num, cudaMemcpyHostToDevice);
158  cudaMalloc((void**)&f_m1_imag_cos_d, sizeof(double) *
      total_pixel_num);
159  cudaMemcpy(f_m1_imag_cos_d, f_m1_imag_cos, sizeof(double) *
       total_pixel_num, cudaMemcpyHostToDevice);
160
161  cudaMalloc((void**)&f_m2_real_d, sizeof(double) *
      total_pixel_num);
162  cudaMemcpy(f_m2_real_d, f_m2_real, sizeof(double) *
      total_pixel_num, cudaMemcpyHostToDevice);
163  cudaMalloc((void**)&f_m2_imag_d, sizeof(double) *
      total_pixel_num);
164  cudaMemcpy(f_m2_imag_d, f_m2_imag, sizeof(double) *
      total_pixel_num, cudaMemcpyHostToDevice);
165
```

```
166  cudaMalloc((void**)&f_m3_real_cos_d, sizeof(double) *
      total_pixel_num);
167  cudaMemcpy(f_m3_real_cos_d, f_m3_real_cos, sizeof(double) *
       total_pixel_num, cudaMemcpyHostToDevice);
168  cudaMalloc((void**)&f_m3_real_sin_d, sizeof(double) *
      total_pixel_num);
169  cudaMemcpy(f_m3_real_sin_d, f_m3_real_sin, sizeof(double) *
       total_pixel_num, cudaMemcpyHostToDevice);
170  cudaMalloc((void**)&f_m3_imag_sin_d, sizeof(double) *
      total_pixel_num);
171  cudaMemcpy(f_m3_imag_sin_d, f_m3_imag_sin, sizeof(double) *
       total_pixel_num, cudaMemcpyHostToDevice);
172  cudaMalloc((void**)&f_m3_imag_cos_d, sizeof(double) *
      total_pixel_num);
173  cudaMemcpy(f_m3_imag_cos_d, f_m3_imag_cos, sizeof(double) *
       total_pixel_num, cudaMemcpyHostToDevice);
174
175  cudaMalloc((void**)&rho_d, sizeof(double) * total_pixel_num
      );
176  cudaMemcpy(rho_d, rho, sizeof(double) * total_pixel_num,
      cudaMemcpyHostToDevice);
177  cudaMalloc((void**)&theta_d, sizeof(double) *
      total_pixel_num);
178  cudaMemcpy(theta_d, theta, sizeof(double) * total_pixel_num
      , cudaMemcpyHostToDevice);
179
180  // Store Zernike moments real and imaginary part
181  int max_zernike_num = (int((max_order - 1) / 2) + 1) * (int
      ((max_order - 1) / 2) + 1 + ((max_order - 1) % 2));
182  zernike_moments Znm;
183  Znm.real = new double[max_zernike_num];
184  Znm.imag = new double[max_zernike_num];
185
186  double* Znm_real_d;
187  double* Znm_imag_d;
188  cudaMalloc((void**)&Znm_real_d, sizeof(double) *
      max_zernike_num);
```

```
189  cudaMalloc((void**)&Znm_imag_d, sizeof(double) *
      max_zernike_num);
190  cudaMemset((void**)&Znm_real_d, 0, sizeof(double) *
      max_zernike_num);
191  cudaMemset((void**)&Znm_imag_d, 0, sizeof(double) *
      max_zernike_num);
192
193  // GPU to calculate Zernike moments
194  dim3 DimBlock = BLK_SIZE;
195  dim3 DimGrid = ceil(total_pixel_num / float(BLK_SIZE));
196
197  // Calculate Kinterner's M1,M2,M3 coefficient
198  kintner_coefficient k_h[k_c_size];
199
200  for (int m = 0; m < max_order; m++)
201  {
202    // Kinterner's M1,M2,M3 coefficient into constant memory
203    int n_num = int(ceil((max_order - m) / 2.0)); // Get n
     number for each m
204    for (int n = m + 4; n < max_order; n = n + 2)
205    {
206      int n_current_position = (n - m) / 2;
207
208      k_h[n_current_position].M1 = 4 * n * (n - 1) / double((
     n + m) * (n - m));
209      k_h[n_current_position].M2 = -2 * (n - 1) * (n * (n -
     2) + m * m) / double((n + m) * (n - m) * (n - 2));
210      k_h[n_current_position].M3 = -1 * n * (n + m - 2) * (n
     - m - 2) / double((n + m) * (n - m) * (n - 2));
211    }
212    cudaMemcpyToSymbol(k_c, k_h, sizeof(kintner_coefficient)
     * n_num);
213
214    if (m % 4 == 0)
215      calculate_Zernike_m02 << <DimGrid, DimBlock >> > (
     Znm_real_d, Znm_imag_d, f_m0_real_d, f_m0_imag_d, rho_d,
     theta_d, m, max_order, total_pixel_num);
```

```
216     else if (m % 4 == 1)
217       calculate_Zernike_m13 << <DimGrid, DimBlock >> > (
      Znm_real_d, Znm_imag_d, f_m1_real_cos_d, f_m1_real_sin_d,
      f_m1_imag_sin_d, f_m1_imag_cos_d, rho_d, theta_d, m,
      max_order, total_pixel_num);
218     else if (m % 4 == 2)
219       calculate_Zernike_m02 << <DimGrid, DimBlock >> > (
      Znm_real_d, Znm_imag_d, f_m2_real_d, f_m2_imag_d, rho_d,
      theta_d, m, max_order, total_pixel_num);
220     else if (m % 4 == 3)
221       calculate_Zernike_m13 << <DimGrid, DimBlock >> > (
      Znm_real_d, Znm_imag_d, f_m3_real_cos_d, f_m3_real_sin_d,
      f_m3_imag_sin_d, f_m3_imag_cos_d, rho_d, theta_d, m,
      max_order, total_pixel_num);
222   }
223
224   cudaMemcpy(Znm.real, Znm_real_d, sizeof(double) *
      max_zernike_num, cudaMemcpyDeviceToHost);
225   cudaMemcpy(Znm.imag, Znm_imag_d, sizeof(double) *
      max_zernike_num, cudaMemcpyDeviceToHost);
226
227   for (int m = 0; m < max_order; m++)
228   {
229     for (int n = m; n < max_order; n = n + 2)
230     {
231       int current_nm_position = (int((n - 1) / 2) + 1) * (int
      ((n - 1) / 2) + 1 + ((n - 1) % 2)) + int(m / 2);
232       double coefficient = 4.0 * (n + 1.0) / (CV_PI * pow(
      width_k, 2));
233
234       Znm.real[current_nm_position] *= coefficient;
235       Znm.imag[current_nm_position] *= coefficient;
236     }
237   }
238
239   delete[] f_m0_real;
240   delete[] f_m0_imag;
```

```
  delete[] f_m1_real_cos;
  delete[] f_m1_real_sin;
  delete[] f_m1_imag_sin;
  delete[] f_m1_imag_cos;
  delete[] f_m2_real;
  delete[] f_m2_imag;
  delete[] f_m3_real_cos;
  delete[] f_m3_real_sin;
  delete[] f_m3_imag_sin;
  delete[] f_m3_imag_cos;
  delete[] rho;
  delete[] theta;
  cudaFree((void*)Znm_real_d);
  cudaFree((void*)Znm_imag_d);
  cudaFree((void*)f_m0_real_d);
  cudaFree((void*)f_m0_imag_d);
  cudaFree((void*)f_m1_real_cos_d);
  cudaFree((void*)f_m1_real_sin_d);
  cudaFree((void*)f_m1_imag_sin_d);
  cudaFree((void*)f_m1_imag_cos_d);
  cudaFree((void*)f_m2_real_d);
  cudaFree((void*)f_m2_imag_d);
  cudaFree((void*)f_m3_real_cos_d);
  cudaFree((void*)f_m3_real_sin_d);
  cudaFree((void*)f_m3_imag_sin_d);
  cudaFree((void*)f_m3_imag_cos_d);
  cudaFree((void*)rho_d);
  cudaFree((void*)theta_d);

  return Znm;
}
```

Snippet 2: GPU kernel function for $m = 4k$ and $m = 4k + 2$

```
__global__ void calculate_Zernike_m02(double* Znm_real,
    double* Znm_imag, double* f_real, double* f_imag, double*
    rho, double* theta, int m, int max_order, int
```

```
    total_pixel_num)
{
  int current_pixel_id = blockDim.x * blockIdx.x + threadIdx.
    x;
  if (current_pixel_id < total_pixel_num) // Boundary check
  {
    double R_minus_2; //Register to reduce IO operation
    double R_minus_4;
    double R_Current;
    double rho_current = rho[current_pixel_id];
    double f_real_tmp = f_real[current_pixel_id];
    double f_imag_tmp = f_imag[current_pixel_id];
    double theta_current = theta[current_pixel_id];
    double cos_m_theta_multipiled_f = __cosf(m *
    theta_current) * f_real_tmp;
    double sin_m_theta_multipiled_f = -__sinf(m *
    theta_current) * f_imag_tmp;

    for (int n = m; n < max_order; n = n + 2)
    {
      int current_nm_position = (int((n - 1) / 2) + 1) * (int
    ((n - 1) / 2) + 1 + ((n - 1) % 2)) + int(m / 2);
      int n_current_position = (n - m) / 2;

      if (n == m)
      {
        R_minus_4 = pow(rho_current, n);
        atomicAdd(&Znm_real[current_nm_position], R_minus_4 *
     cos_m_theta_multipiled_f);
        atomicAdd(&Znm_imag[current_nm_position], R_minus_4 *
     sin_m_theta_multipiled_f);
      }

      else if (n == (m + 2))
      {
        R_minus_2 = (m + 2) * R_minus_4 * rho_current *
    rho_current - (m + 1) * R_minus_4;
```

```
31          atomicAdd(&Znm_real[current_nm_position], R_minus_2 *
    cos_m_theta_multipiled_f);
32          atomicAdd(&Znm_imag[current_nm_position], R_minus_2 *
    sin_m_theta_multipiled_f);
33        }
34
35        else
36        {
37          R_Current = (k_c[n_current_position].M1 * rho_current
    * rho_current + k_c[n_current_position].M2) * R_minus_2 +
    k_c[n_current_position].M3 * R_minus_4;
38          atomicAdd(&Znm_real[current_nm_position], R_Current *
    cos_m_theta_multipiled_f);
39          atomicAdd(&Znm_imag[current_nm_position], R_Current *
    sin_m_theta_multipiled_f);
40          R_minus_4 = R_minus_2;
41          R_minus_2 = R_Current;
42        }
43      }
44    }
45 }
```

Snippet 3: GPU kernel function for $m = 4k + 1$ and $m = 4k + 3$

```
1  __global__ void calculate_Zernike_m13(double* Znm_real,
     double* Znm_imag, double* f_real_cos, double* f_real_sin,
     double* f_imag_sin, double* f_imag_cos, double* rho,
     double* theta, int m, int max_order, int total_pixel_num)
2  {
3    int current_pixel_id = blockDim.x * blockIdx.x + threadIdx.
     x;
4    if (current_pixel_id < total_pixel_num) // Boundary check
5    {
6      double R_minus_2; //Register to reduce IO operation
7      double R_minus_4;
8      double R_Current;
9      double rho_current = rho[current_pixel_id];
```

```
10      double f_real_cos_tmp = f_real_cos[current_pixel_id];
11      double f_real_sin_tmp = f_real_sin[current_pixel_id];
12      double f_imag_cos_tmp = f_imag_cos[current_pixel_id];
13      double f_imag_sin_tmp = f_imag_sin[current_pixel_id];
14      double theta_current = theta[current_pixel_id];
15      double real_part_tmp = __cosf(m * theta_current) *
    f_real_cos_tmp + __sinf(m * theta_current) *
    f_real_sin_tmp;
16      double imag_part_tmp = -__sinf(m * theta_current) *
    f_imag_sin_tmp - __cosf(m * theta_current) *
    f_imag_cos_tmp;
17
18      for (int n = m; n < max_order; n = n + 2)
19      {
20        int current_nm_position = (int((n - 1) / 2) + 1) * (int
    ((n - 1) / 2) + 1 + ((n - 1) % 2)) + int(m / 2);
21        int n_current_position = (n - m) / 2;
22
23        if (n == m)
24        {
25          R_minus_4 = pow(rho_current, n);
26          atomicAdd(&Znm_real[current_nm_position], R_minus_4 *
     real_part_tmp);
27          atomicAdd(&Znm_imag[current_nm_position], R_minus_4 *
     imag_part_tmp);
28        }
29
30        else if (n == (m + 2))
31        {
32          R_minus_2 = (m + 2) * R_minus_4 * rho_current *
    rho_current - (m + 1) * R_minus_4;
33          atomicAdd(&Znm_real[current_nm_position], R_minus_2 *
     real_part_tmp);
34          atomicAdd(&Znm_imag[current_nm_position], R_minus_2 *
     imag_part_tmp);
35        }
36
```

```
37        else
38        {
39          R_Current = (k_c[n_current_position].M1 * rho_current
      * rho_current + k_c[n_current_position].M2) * R_minus_2 +
      k_c[n_current_position].M3 * R_minus_4;
40          atomicAdd(&Znm_real[current_nm_position], R_Current *
      real_part_tmp);
41          atomicAdd(&Znm_imag[current_nm_position], R_Current *
      imag_part_tmp);
42          R_minus_4 = R_minus_2;
43          R_minus_2 = R_Current;
44        }
45      }
46    }
47  }
```

References

[1] Chee-Way Chong, P Raveendran, and Ramakrishnan Mukundan. A comparative analysis of algorithms for fast computation of zernike moments. *Pattern Recognition*, 36(3):731–742, 2003.

[2] Mark Harris. Optimizing Parallel Reduction in CUDA. `https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf`.

[3] Heloise Hse and A Richard Newton. Sketched symbol recognition using zernike moments. In *Proceedings of the 17th International Conference on Pattern Recognition, 2004. ICPR 2004.*, volume 1, pages 367–370. IEEE, 2004.

[4] Sun-Kyoo Hwang and Whoi-Yul Kim. Fast and efficient method for computing art. *IEEE Transactions on Image Processing*, 15(1):112–117, 2005.

[5] Sun-Kyoo Hwang and Whoi-Yul Kim. A novel approach to the fast computation of zernike moments. *Pattern Recognition*, 39(11):2065–2076, 2006.

[6] Ismail A Ismail, Mohamed A Shouman, Khalid M Hosny, and Hayam M Abdel Salam. Invariant image watermarking using accurate zernike moments 1. 2010.

[7] Hyung Shin Kim and Heung-Kyu Lee. Invariant image watermark using zernike moments. *IEEE transactions on Circuits and Systems for Video Technology*, 13(8):766–775, 2003.

[8] Eric C Kintner. On the mathematical properties of the zernike polynomials. 1976.

[9] David B Kirk and W Hwu Wen-Mei. *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.

[10] AH Kulkarni, HM Rai, KA Jahagirdar, PS Upparamani, et al. A leaf recognition technique for plant classification using rbpnn and zernike moments. *International Journal of Advanced Research in Computer and Communication Engineering*, 2(1):984–988, 2013.

[11] Yogesh Kumar, Ashutosh Aggarwal, Shailendra Tiwari, and Karamjeet Singh. An efficient and robust approach for biomedical image retrieval using zernike moments. *Biomedical Signal Processing and Control*, 39:459–473, 2018.

[12] Shan Li, Moon-Chuen Lee, and Chi-Man Pun. Complex zernike moments features for shape-based image retrieval. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 39(1):227–237, 2008.

[13] Simon Liao and George A Papakostas. Accuracy analysis of moment functions. *Moments and Moment Invariants-Theory and Applications, edited by George A. Papakostas*, 1:33–56, 2014.

[14] Marko Lukic, Eva Tuba, and Milan Tuba. Leaf recognition algorithm using support vector machine with hu moments and local binary patterns. In *2017 IEEE 15th international symposium on applied machine intelligence and informatics (SAMI)*, pages 000485–000490. IEEE, 2017.

[15] Petr Novotnỳ and Tomáš Suk. Leaf recognition of woody species in central europe. *Biosystems Engineering*, 115(4):444–452, 2013.

[16] Chhaya Patel. Different optimization strategies and performance evaluation of reduction on multicore cuda architecture. *International Journal of Engineering Research & Technology (IJERT)*, 3(4), 2014.

[17] Aluizio Prata and WVT Rusch. Algorithm for computation of zernike polynomials expansion coefficients. *Applied Optics*, 28(4):749–754, 1989.

[18] Manuel Jesús Martín Requena, Pablo Moscato, and Manuel Ujaldón. Efficient data partitioning for the gpu computation of moment functions. *Journal of Parallel and Distributed Computing*, 74(1):1994–2004, 2014.

[19] Chandan Singh, Neerja Mittal, and Ekta Walia. Face recognition using zernike and complex zernike moment features. *Pattern Recognition and Image Analysis*, 21(1):71–81, 2011.

[20] Chandan Singh and Ekta Walia. Fast and numerically stable methods for the computation of zernike moments. *Pattern Recognition*, 43(7):2497–2506, 2010.

[21] Michael Reed Teague. Image analysis via the general theory of moments. *Josa*, 70(8):920–930, 1980.

[22] Zernike von F. Beugungstheorie des schneidenver-fahrens und seiner verbesserten form, der phasenkontrastmethode. *physica*, 1(7-12):689–704, 1934.

[23] Xiaoyu Wang and Simon Liao. Image reconstruction from orthogonal fourier-mellin moments. In *International Conference Image Analysis and Recognition*, pages 687–694. Springer, 2013.

[24] Stephen Gang Wu, Forrest Sheng Bao, Eric You Xu, Yu-Xuan Wang, Yi-Fan Chang, and Qiao-Liang Xiang. A leaf recognition algorithm for plant classification using probabilistic neural network. In *2007 IEEE international symposium on signal processing and information technology*, pages 11–16. IEEE, 2007.

[25] Yongqing Xin, Simon Liao, and Miroslaw Pawlak. Circularly orthogonal moments for geometrically robust image watermarking. *Pattern Recognition*, 40(12):3740–3752, 2007.

[26] Yubo Xuan, Dayu Li, and Wei Han. Efficient optimization approach for fast gpu computation of zernike moments. *Journal of Parallel and Distributed Computing*, 111:104–114, 2018.

[27] F Zernike. Diffraction theory of the knife-edge test and its improved version, the phase-contrast method. *Physica*, 1:689–704, 1934.