J. Parallel Distrib. Comput. 111 (2018) 104–114

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Efficient optimization approach for fast GPU computation of Zernike moments

Yubo Xuan [a,b], Dayu Li [c,*], Wei Han [b,*]

[a] College of Communication Engineering, Jilin University, Changchun, Jilin, 130012, China
[b] College of Physics, Jilin University, Changchun, Jilin, 130012, China
[c] State Key Laboratory of Applied Optics, Changchun Institute of Optics, Fine Mechanics and Physics, Chinese Academy of Sciences, Changchun, Jilin, 130033, China

HIGHLIGHTS

- The proposed approach significantly reduces computational time of ZMs on a GPU.
- The proposed approach supports the expansion of computation ZMs on multi-GPU.
- Eliminating huge conditional instructions after the image re-layout.

ARTICLE INFO

Article history:
Received 17 September 2016
Received in revised form 8 June 2017
Accepted 30 July 2017
Available online 9 August 2017

Keywords:
Zernike moments
GPU
Reordering image pixels
Addressing diagonal in advance

ABSTRACT

Our study focuses on accelerating the computation of Zernike moments on graphics processing units (GPUs). There are two ideas to achieve the goal. First is to implement a novel re-layout that involves reordering the image pixels and addressing the diagonal pixels in advance, so that computations of all pixels are allocated to an octant effectively. Second is to the leverage the constant memory to store precomputed values used across GPU threads. An in-depth study has been carried out to evaluate the performance in each case and to compare against GPU implementation of other algorithms and to discuss the bottleneck. The result shows that our approach is effective and achieves significant performance improvement compared to other GPU state-of-the-art implementations. Furthermore, our approach is suited for allocating the data flow into multiple GPUs.

1. Introduction

Zernike moments (ZMs) are the mapping of an image onto a set of orthogonal Zernike polynomials. Among all orthogonal moments, they are the best image descriptors because of their unique characteristics, such as minimum information redundancy and invariance to image rotation and scaling [36]. They are used in numerous applications including pattern or object recognition [1,3,10,27,34,35], image reconstruction [31,41], shape matching [16,25], image watermarking [7,15], and image retrieval [19,32,40].

Unfortunately, their computation is very expensive owing to the high complexity of the definition. On the other hand, high-order moments are computationally slow and suffer from errors. Thus real applications like computer vision require a quick response and demand the computation of features at real-time. Several algorithms have been proposed to accelerate the computation of ZMs.

(1) Mukundan and Gu both introduced fast algorithms, namely square-to-circular transformation method to compute ZMs [11,24]. However, their proposed algorithms have limitations. In the case of the contour integration method, it is applicable only for binary images and requires off-line analysis to extract the image boundary points. The accuracy of ZMs will be compromised when the square-to-circular transformation method is used.

(2) Because the factorial terms in the radial polynomials take most of the computational time to derive the ZMs, researchers have tried to remove the factorials by introducing a recurrence relationship. Prata [28] and Kintner [17] proposed a recurrence relation for the radial polynomials of ZMs. It is applicable in cases when ZMs with selected fixed repetition m and order n are required as pattern features. Chong [2] proposed the q-recursive method which uses Zernike radial polynomials of fixed order n with high repetition m to derive a polynomial of low repetition m. These methods do not use any factorial functions, which in turn substantially reduces

* Corresponding authors.
E-mail addresses: lidayu@ciomp.ac.cn (D. Li), 21505211@qq.com (W. Han).

http://dx.doi.org/10.1016/j.jpdc.2017.07.008
0743-7315/© 2017 Elsevier Inc. All rights reserved.

the computational time to derive the ZMs. Among these iteration methods, the Kintner's and q-recursive methods show the best performance.

(3) Hwang proposed a fast computation algorithm that is based on the symmetry of Zernike polynomials [13,14]. It reduces the computational complexity in the generation of Zernike polynomials or aid in obtaining ZMs by projection of the Zernike polynomials onto the image functions. The Zernike polynomials are generated by computing only one of their octant. Therefore, the computational time for this algorithm is approximately one-eighth of the baseline computing method [see Section 4.1].

(4) For real-time implementation of some fixed-sized images, Zernike polynomials are computed in advance and stored in a set of lookup tables [12,16]. Thereafter, ZMs are computed by projecting the image patched onto the pre-computed Zernike polynomials. The entire process works in real-time, but this scenario is valid only for computing fixed images of ZMs. When computing a set of ZMs, the symmetry method can be mixed with other fast methods such as Kintner's or Chong's method, and the combined method was shown to have the best performance on computational time [6,33].

(5) The aforementioned optimization only extensively focused on the algorithm that was applied to reduce the CPU execution time. Graphics processing units (GPUs) are becoming increasingly helpful to parallel data applications over the last few years, because the architecture is composed of large amounts of simple processing units [5,18,23]. The following studies confirm acceleration power of GPUs in ZMs with a high degree of data parallelism. The computation of ZMs by definition on GPUs has achieved observable higher speeds as compared to all algorithms in the CPU. Ujaldon [39] carried out an in-depth research on implementation of ZMs on a GPU. The results on a commodity PC showed up to 5× faster execution times on a Geforce 8800 GTX against that on a Pentium 4CPU. Toharia [38] presented an analysis of a multi-GPU, multi-CPU environment. The analysis was performed on a shot boundary detection application based on ZMs. Martín-Requena [22] executed ZMs of 1 megapixel images on many-cores and clusters of GPUs to attain gains of up to three orders of magnitude when compared with execution on multi-core CPUs of similar age. Meanwhile, utilization of Hwang's symmetric algorithm has been done, but its performance has not improved compared to the baseline implementation on a single GPU [see Table 6 first three columns].

In this study, efforts are made to further accelerate the computation of ZMs over the existing GPU implementation. First, the difficulties of implementing the symmetric method are analyzed. Then, two approaches are presented: (1) a novel re-layout that involves reordering the image pixels and addressing the diagonal pixels in advance, and (2) storing pre-computed factorial terms using lookup tables in the constant memory. An in-depth study is carried out to evaluate the performance and bottlenecks. Furthermore, our scheme supports the expansion on multi-GPU via a simple partitioning mechanism. As a result, our implementation over either different generations of GPUs or multi-GPUs achieves higher performance than the state-of-the-art implementations.

The rest of the paper is organized as follows. The next section describes ZMs and introduces their properties. Section 3 introduces the CUDA programming model and the implementation of ZMs by definition on GPU. In Section 4, we describe a symmetric algorithm and discuss the related problems when porting it to GPUs. Subsequently, we designed a parallel computational approach based on our proposed solution, which is implemented on GPUs to accelerate the computation of ZMs. The experimental results and bottlenecks are discussed in Section 5. Finally, conclusions are given in the last section.

2. ZMs and their properties

The use of ZMs in image analysis was pioneered by Teague [36]. ZMs are features extracted by projecting the input image on a complex set of Zernike polynomials. A Zernike polynomial $V_{nm}(x, y)$ is defined within a unit circle in a complex domain [43], and it can be written in polar form by Euler's formula:

$$V_{nm}(x, y) = R_{nm}(\rho)\cos(m\theta) + jR_{nm}(\rho)\sin(m\theta), \tag{1}$$

where

$$j = \sqrt{-1}, x^2 + y^2 \le 1, \rho = \sqrt{x^2 + y^2}\ \theta = \tan^{-1}\left(\frac{y}{x}\right), \tag{2}$$

and $R_{n,m}(\rho)$ is the Zernike radial polynomial defined as

$$R_{nm}(\rho) = \sum_{s=0}^{\frac{n-|m|}{2}} (-1)^s \frac{(n-s)!}{s!\left(\frac{(n+|m|)}{2} - s\right)!\left(\frac{(n-|m|)}{2} - s\right)!}\rho^{n-2s}. \tag{3}$$

In Eq. (3), order n is a non-negative integer and repetition m is an integer satisfying $n - |m| = (even)$ and $|m| \le n$. s is an integer variable ranging from 0 to $(\frac{n-|m|}{2})$. The Zernike polynomials are orthogonal and satisfy the following equation:

$$\int_0^{2\pi}\int_0^1 V_{nm}^*(\rho, \theta)V_{pq}(\rho, \theta)\rho d\rho d\theta = \begin{cases} \dfrac{\pi}{(n+1)} & \text{if } n = p, m = q \\ 0 & \text{otherwise.} \end{cases} \tag{4}$$

This means that there is no redundancy of information between the moments with different orders and repetitions.

ZMs of order n and repetition m are defined by

$$\begin{aligned} Z_{nm} &= \frac{n+1}{\pi}\iint_{x^2+y^2\le 1} f(x, y)V_{nm}^*(x, y)\,dxdy \\ &= \frac{n+1}{\pi}\iint_{x^2+y^2\le 1} R_{nm}(\rho)\left[f(x, y)\cos(m\theta)\right. \\ &\left.- jf(x, y)\sin(m\theta)\right]dxdy. \end{aligned} \tag{5}$$

The symbol $*$ denotes a complex conjugate. ZMs are the projection of the image function $f(x, y)$ onto those orthogonal Zernike polynomials . The procedure for computing ZMs can be understood as the inner product between the image function $f(x, y)$ and the Zernike polynomial. Note that $R_{nm}(\rho) = R_{n(-m)}(\rho)$, and $Z_{n,-m} = Z_{nm}^*$. Hence, only $Z_{n,m}$ $(m \ge 0)$ is considered as the feature. The total number of ZMs can be computed by

$$N_{\text{accumul}} = \sum_{n=0}^{\text{Max}}\left\{\left\lfloor\frac{n}{2}\right\rfloor + 1\right\}, \tag{6}$$

where $\lfloor u \rfloor$ is the integer part of u. Table 1 lists the ZMs of a few selected maximal orders. Obviously, they provide a large number of image features.

The fundamental property of ZMs is their rotational invariance. If an image is rotated by an angle θ, the moments of the original image Z_{nm} and the rotated image Z'_{nm} can be written as

$$Z'_{nm} = Z_{nm}e^{-jm\theta}. \tag{7}$$

We have

$$\left|Z'_{nm}\right| = \left|Z_{nm}e^{-jm\theta}\right| = \left|Z_{nm}\right|, \text{ and} \tag{8}$$

$$\varphi'_{nm} = \varphi_{nm} - \theta, \tag{9}$$

where $|Z_{nm}|$ and φ_{nm} represent the magnitude and phase, respectively. Its derivation process could be found in [35]. The magnitude

Table 1
List of ZMs of different maximal orders.

Order	Moments	No. of moments	Accumulative No.
0	Z_{00}	1	1
1	Z_{11}	1	2
2	Z_{20}, Z_{22}	2	4
…	…	…	…
9	$Z_{91}, Z_{93}, Z_{95}, Z_{97}, Z_{99}$	5	30
10	$Z_{10,0}, Z_{10,2} Z_{10,4}, Z_{10,6} Z_{10,8}, Z_{10,10}$	6	36
11	$Z_{11,1}, Z_{11,3}, Z_{11,5}, Z_{11,7}, Z_{11,9}, Z_{11,11}$	6	42
…	…	…	…
15	$Z_{15,1}, Z_{15,3}, Z_{15,5}, Z_{15,7}, Z_{15,9}, Z_{15,11}, Z_{15,13}, Z_{15,15}$	8	72

remains unchanged while the phase would change with image rotation. A simple experiment was conducted to verify this property. We first take an image of 128 $\times$ 128 pixels, and it is rotated by 45° and 90°. The ZMs for $n = 5$ and $m = 1$ is calculated. The real part and imaginary part of the ZMs change after the image rotation, but the ZM magnitude is a constant [see Fig. 1].

Thus, the magnitude of the ZMs can be considered as a rotational invariant feature of the image. These moment invariant are opted in many analysis and pattern recognition applications [10,16,35]. The choice of the maximal order value n_{max} or some single ZMs will depend on the size of the given image and also on the solution required. It can be observed that the phase of ZMs is computed from the iris images of the same subject [3]. Both the phase and magnitude of the ZMs are utilized in image recognition and retrieval [32,34]. Note that the moments Z_{00} and Z_{11} are not included in the feature construction. These moments are modified to known values during the normalization procedure. For example, Z_{00} is a real constant and represents the average intensity of an input image. It is used to scale a watermark image to a standard size [15] or to denote a feature by using $\frac{|Z_{nm}|}{|Z_{00}|}$ [12].

As Zernike 1-D radial polynomials [see Eq. (3)] contain many factorial terms, they occupy most of the computational time in radial polynomials.

Kintner [17] used polynomials of varying low-order n with a fixed repetition m to compute the radial polynomials. The recurrence relation is

$$R_{n,m}(\rho) = \frac{\left(K_2 \rho^2 + K_3\right) R_{(n-2),m}(\rho) + K_4 R_{(n-4),m}(\rho)}{K_1}, \tag{10}$$

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

$$K_1 = \frac{(n+m)(n-m)(n-2)}{2}, \quad K_2 = 2n(n-1)(n-2),$$
$$K_3 = -m^2(n-1) - n(n-1)(n-2),$$
$$K_4 = -\frac{n(n+m-2)(n-m-2)}{2}.$$

Chong [2] proposed another relation called the q-recursive method which is defined as

$$R_{n,m}(\rho) = H_1 R_{n(m+4)}(\rho) + \left(H_2 + \frac{H_3}{\rho^2}\right) R_{n(m+2)}(\rho), \tag{11}$$

where

$$H_1 = \frac{(m+4)(m+3)}{2} - (m+4) H_2 + \frac{H_3 (n+m+6)(n-m-4)}{8},$$
$$H_2 = \frac{H_3 (n+m+4)(n-m-2)}{4(m+3)} + (m+2),$$
$$H_3 = -\frac{4(m+2)(m+1)}{(n+m+2)(n-m)}.$$

Because recurrence relations are adopted, these methods require computation of extra ZMs even if only a single ZM is required. These methods focus only on computing Zernike radial polynomials and do not attempt to reduce the computational complexity.

Slowdown in speed is caused not only by the factorial terms in the radial polynomials, but also by the evaluation of N^2 image points of the Zernike polynomials. Furthermore, time is taken by the trigonometric functions, $\cos(m\theta)$ and $\sin(m\theta)$, where $m = 0, 1, \ldots, n_{max}$. If both image size N and order n are large, computations involved are significantly large as well.

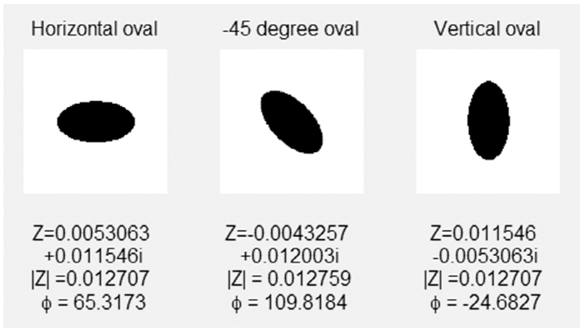


Fig. 1. Example of rotational invariance of ZM.

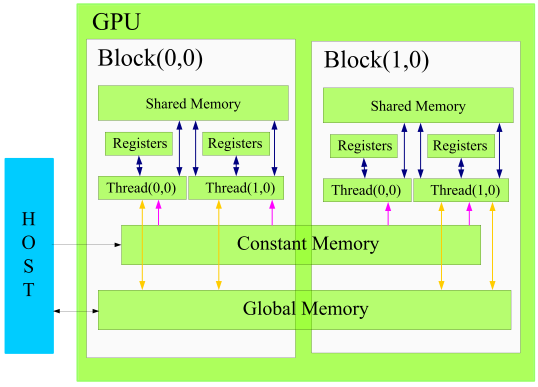


Fig. 2. The CUDA programming model.

3. Computing ZMs by baseline method on GPU

3.1. GPU architecture overview

A GPU is actually an array of *streaming multiprocessors* (SMs), each of which has many *streaming processors* (SP). Massive GPU hardware parallelism is achieved through replication of a common SM architecture. Each SP can execute exactly the same instruction on different data, making it similar to a single instruction multiple thread (SIMT) processor. In 2007, NVIDIA released the Compute Unified Device Architecture (CUDA) GPU programming toolkit [26]. After the release of CUDA, programmers have achieved parallel algorithms through C-like language without the need for any complex graphics programming. Owing to its programmability, GPU can be seen as a device capable of executing a very large number of threads in parallel.

In the CUDA model [see Fig. 2], a host CPU code can launch GPU kernels by calling device functions that execute on the GPU. As the GPU uses a different instruction sets from the host CPU, the CUDA

Table 2
CUDA memory types and characteristics.

Memory	Location	Access	Scope
Register	On-chip	Read/write	One thread
Shared	On-chip	Read/write	All threads in a block
Global	Off-chip	Read/write	All threads + host
Constant	Off-chip	Read	All threads + host

compilation flow compiles CPU and GPU codes using different compilers targeting different instruction sets. When a GPU kernel is executed by multiple equally-sharp blocks (a block may contain up to 1024 threads), the total number of threads is equal to the number of threads per block times the number of blocks. These threads are mapped onto a hierarchy of hardware resources. Blocks of threads are executed within SMs. The minimum scheduled unit in the SM is the warp, which is 32 threads on the current NVIDIA GPUs. Active threads within a warp run in lock-step within the GPU.

The most significant difference between CPUs and GPUs resides in the memory subsystem. GPUs typically feature a wide memory bus to feed a large number of parallel threads. Table 2 summarizes the characteristics of various CUDA memory spaces. CUDA has both on-chip and off-chip memories. The off-line global memory can be accessed by all SMs across the GPU. It is measured in gigabytes (GB) of memory, which is by far the largest, most commonly used, and slowest memory storage on the GPU. Hence, in general we should reduce the global memory access or use coalesced memory access patterns [9]. The constant memory is a limited off-chip memory space (64KB for GeForce9800GX2 and TeslaK40); it is an excellent way of storing and broadcasting read-only data to all threads. The fastest and most scalable is the highly desirable on-chip memory. These are limited memory stores measured in kilobytes (KB) of storage. A set of threads in the same block can cooperate with one another by sharing data through the shared memory. Two threads from two different blocks can share data, but the speed is much slower because the global memory is placed off-chip. Registers are on-chip resources that are equally divided across active threads.

Though GPUs offer very high computing power at relatively low cost, designing efficient algorithms for the GPUs normally requires additional time and effort, even for experienced programmers. Research has shown that taking advantage of memory spaces can be the key limiting factor in terms of overall cost [8,30]. Application acceleration is highly dependent on being able to utilize the memory subsystem effectively so that all execution units remain busy.

3.2. Baseline implementation on GPU

In this section, we describe a baseline implementation which denotes the computation of ZMs by the definitions on GPU. As Zernike polynomial are defined within a unit circle, the coordinates of the image must be normalized into $[-1, 1]$ by a mapping transform, in which case the pixels located outside the circle are not involved in the calculation. Fig. 3 illustrates an inscribed circle of mapping transform. The number within each grid represents the index of the thread. Accordingly, the discrete form of the ZMs of an image of size $N \times N$ is described as

$$Z_{nm} = \frac{(n+1)}{\lambda_N} \sum_{i=0}^{N-1} \sum_{k=0}^{N-1} f(x, y)\, V_{nm}^{*}(x, y) \tag{12}$$

$$= \frac{(n+1)}{\lambda_N} \sum_{i=0}^{N-1} \sum_{k=0}^{N-1} f(x, y)\, R_{nm}(\rho) \left[\cos(m\theta) - j\sin(m\theta)\right]. \tag{13}$$

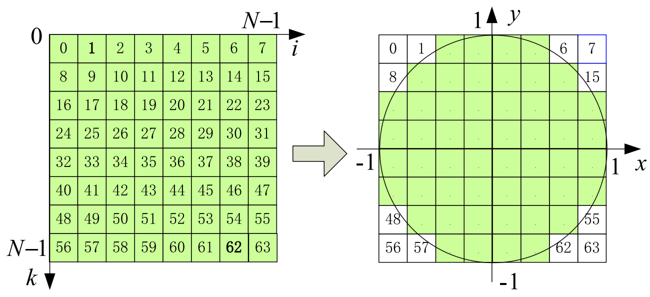


Fig. 3. Mapping transform of an inscribed circle for normalization.

```
FUNCTION   radial_polynomial(ρ, n, m)                          kernel1
radial=0
for s=0 to $\frac{n-m}{2}$
    $c = (-1)^s \frac{(n-s)!}{s!\left(\frac{n+|m|}{2}-s\right)!\left(\frac{n-|m|}{2}-s\right)!}$
radial=radial+ $c \times \rho^{n-2s}$
end for        return radial
FUNCTION  Zernike_moment_map(n,m)
tid=get_thread_id()
i=tid%N                    k=tid/N
x=2(i-N/2+0.5)/N        y=2(k-N/2+0.5)/N
$\rho = \sqrt{x^2 + y^2}$
If $\rho \le 1$
    $\theta = \arctan\left(\frac{y}{x}\right)$
radial=radial_polynomial(ρ, n, m)
    $vr = radial \times \cos(m\theta)$      $vi = radial \times \sin(m\theta)$
zr_map[tid]=f(x,y) ×vr           zi_map[tid]=f(x,y) ×vi
end if   return zr_map,zi_map
Function    sum_zernike_moment_map(zr_map,zi_map)
tid=get_thread_id
kernel2
zr=sum(zr_map[tid])      zi=sum(zi_map[tid])
return $\frac{(n+1)}{\lambda_N}(zr + jzi)$
```

Fig. 4. Pseudocode for computing ZMs by baseline method.

The normalization factor λ_N must be the number of pixels located in the unit circle by the mapping transform. Here $i = \text{tid}\%N$, and $k = \text{tid}/N$, where tid is the index of the thread, and

$$x = \frac{2\left(i - \frac{N}{2} + 0.5\right)}{N}, \tag{14}$$

$$y = \frac{2\left(k - \frac{N}{2} + 0.5\right)}{N}. \tag{15}$$

We use a conditional instruction ($\rho \le 1$) that describes the image properties of the inside unit circle. Hence, the pixels outside the unit circle are excluded from the computational domain established for ZMs.

The procedure to calculate a single Z_{nm}, applying the formulas in Eqs. (13)–(15), is divided into two GPU kernels as shown in Fig. 4. Kernel 1 is assigned to $N \times N$ threads as an input $N \times N$ image, the indexes of threads traverse the image. Under the condition $\rho \le 1$, the radial polynomials $R_{nm}(\rho)$ is calculated using Eq. (3). The real and imaginary parts of the Zernike polynomials (vr and vi) are obtained from $R_{nm}(\rho)\cos(m\theta)$ and $R_{nm}(\rho)\sin(m\theta)$.

Subsequently the real and imaginary parts of the ZM maps (zr_map and zi_map) are derived through image times vr and vi,

Table 3
Polar coordinates and gray values of point P_i $(i = 1, 2, \ldots, 8)$ in Fig. 5.

Index of octant	Point	Radial $0 \le \rho \le 1$	Polar angle $0 \le \theta \le \pi/4$	Gray value of pixel
1	P_1	ρ	θ	h_1
2	P_2	ρ	$\frac{\pi}{2} - \theta$	h_2
3	P_3	ρ	$\frac{\pi}{2} + \theta$	h_3
4	P_4	ρ	$\pi - \theta$	h_4
5	P_5	ρ	$\pi + \theta$	h_5
6	P_6	ρ	$\frac{3\pi}{2} - \theta$	h_6
7	P_7	ρ	$\frac{3\pi}{2} + \theta$	h_7
8	P_8	ρ	$2\pi - \theta$	h_8

respectively. The task of kernel 2 is to generate ZMs by parallel reduction of the ZM map.

4. Implementation of the proposed approach on GPU

4.1. Symmetric algorithm for computation of ZMs

There are eight points in different octants of a unit circle, and the coordinates of the point P_1 in the first octant are (x, y) [see Fig. 5(a)]. The eight points P_i are symmetric with respect to the x-axis, y-axis, origin, and line $y = x$. h_i $(i = 1, 2, \ldots, 8)$ is the gray value of the image at point P_i [see Fig. 5(b)]. The polar radii at points P_i are the same, i.e., ρ. If θ is the polar angle of point P_1 in the first octant, the angles of other symmetric points can be easily expressed in terms of θ and π , as shown in Table 3.

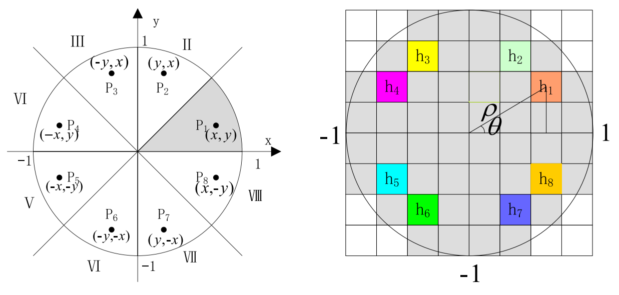
By the symmetry characteristic, ZM can be obtained with an octant of the Zernike polynomials:

$$Z_{nm} = \frac{(n+1)}{\lambda_N} \sum_{i=\frac{N}{2}}^{N-1} \sum_{k=0}^{\frac{N}{2}} R_{nm}(\rho) \left[g_m^r(x, y) - j g_m^i(x, y) \right] \tag{16}$$

where $g_m^r(x, y)$ and $g_m^i(x, y)$ are respectively the variants of $f(x, y)\cos(m\theta)$ and $f(x, y)\sin(m\theta)$ in the first octant $(0 \le y \le x \le 1)$ of a unit circle in Eq. (13). They are further divided into four cases as shown below.

$$\begin{aligned}
g_m^r(x, y) &= \begin{cases} [h_1 + h_2 + h_3 + h_4 + h_5 + h_6 + h_7 + h_8]\cos(m\theta), & m = 4k \\ [h_1 - h_4 - h_5 + h_8]\cos(m\theta) + [h_2 - h_3 - h_6 + h_7]\sin(m\theta), & m = 4k+1 \\ [h_1 - h_2 - h_3 + h_4 + h_5 - h_6 - h_7 + h_8]\cos(m\theta), & m = 4k+2 \\ [h_1 - h_4 - h_5 + h_8]\cos(m\theta) + [-h_2 + h_3 + h_6 - h_7]\sin(m\theta), & m = 4k+3 \end{cases} \\
g_m^i(x, y) &= \begin{cases} [h_1 - h_2 + h_3 - h_4 + h_5 - h_6 + h_7 - h_8]\sin(m\theta), & m = 4k \\ [h_1 + h_4 - h_5 - h_8]\sin(m\theta) + [h_2 + h_3 - h_6 - h_7]\cos(m\theta), & m = 4k+1 \\ [h_1 + h_2 - h_3 - h_4 + h_5 + h_6 - h_7 - h_8]\sin(m\theta), & m = 4k+2 \\ [h_1 + h_4 - h_5 - h_8]\sin(m\theta) + [-h_2 - h_3 + h_6 + h_7]\cos(m\theta), & m = 4k+3. \end{cases}
\end{aligned} \tag{17}$$

The detailed explanation is found in [14]. Eq. (16) only requires the computation of ρ in the first octant. This means that the number of pixels involved in the computation of the Zernike radial polynomials $R_{nm}(\rho)$ can be reduced to one-eighth. Furthermore the pixels of an image in a unit circle are added to Eq. (17). The total number of necessary multiplications is markedly reduced. This results in a successful improvement in computational time. $g_m^r(x, y)$ and $g_m^i(x, y)$ are called octant images in this study.



(a) Form in Cartesian coordinates. (b) Form in polar coordinates.

Fig. 5. Symmetry points corresponding to P_1.

4.2. Optimization strategy

Hwang [14] reported nearly 8× factor gain by using the above symmetry algorithm against the baseline implementation in a CPU. Martín-Requena [22] attempted to optimize the computation of ZMs by a symmetry algorithm on a GPU. However, moving the high-performance algorithm to the GPU is very challenging owing to the complexity of the GPU hardware. The experimental results are shown in the first three columns of Table 6 (see Section 5.2). According to the results, the symmetry method is not competitive at all. The authors analyzed the reasons for the low performance as follows: the implementation requires conditional statements to be placed within the innermost loop of the computation, and this generates a huge numbers of warp divergences in CUDA, which results in poor occupancy for the streaming processors.

Moreover, the valid thread ratio limits the performance of the symmetry method on GPU in our view point. The total index of threads in an $N \times N$ image for the baseline implementation is $N \times N$; the valid area mapping the image to parallel threads is the green area as shown in Fig. 6(a). The outer region of the unit circle is rejected using the conditional instructions,$\rho \le 1$. Correspondingly, the valid thread ratio of computing ZMs is 78.5%.

The total index of threads in an $N \times N$ image for the symmetry method is $\frac{N \times N}{4}$. The valid area mapping the image to parallel threads is a quadrant as shown in Fig. 6(b). Only an octant is required , therefore the valid thread ratio is merely 39.25%.

To eliminate the above factors limiting the performance of the symmetry method, we proposed a new method, which involves re-ordering the image and addressing the diagonal pixels in advance. A 2D image of size $N \times N$ is converted into 8 one-dimensional image information.

4.2.1. Symmetric algorithm is ported to GPU efficiently by re-layout

We present a novel data re-layout to exploit the symmetry of ZMs. An 8 × 8 image illustrates a symmetric characteristic as shown in Fig. 7. The pixels placed in the diagonal have three symmetric partners, where as the remaining ones have seven.

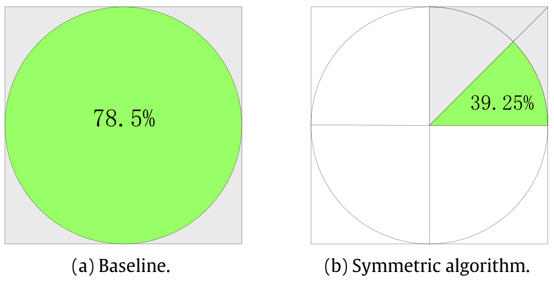


Fig. 6. Valid thread ratio in computing ZMs. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

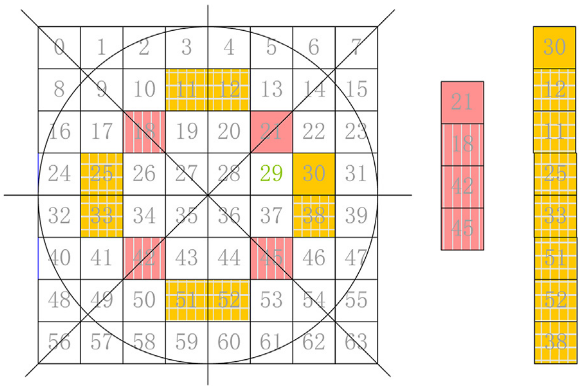


Fig. 7. Two types of symmetric pixels.

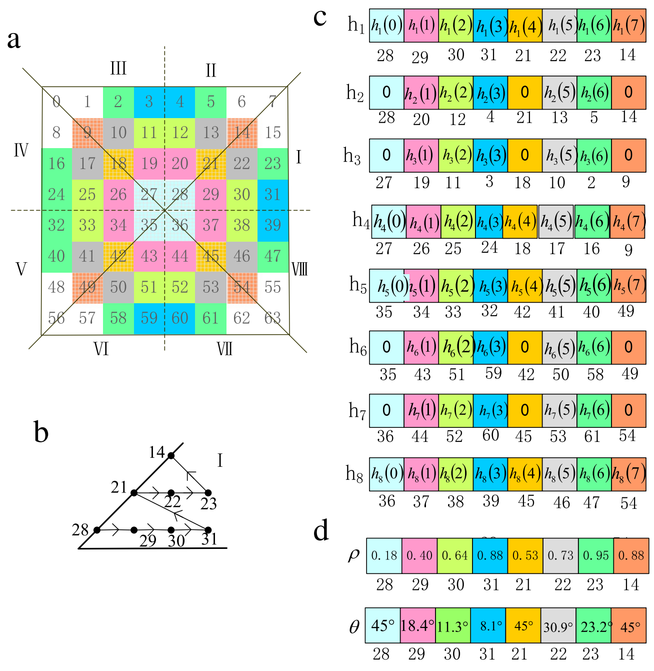


Fig. 8. Re-layout of an 8 × 8 image and coordinates ρ, θ in the first octant. (a) Same colors are pixels symmetric to each other (b) Loaded sequences in first octants (c) Images data are reordered into 8 arrays (d) ρ and θ in the first octant are stored into 2 arrays. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Hence, the diagonal pixels must be considered separately. A common reorder needs to address two types of symmetric pixels using conditional instructions. A better scenario is presented to address symmetric pixels without conditional instructions.

An 8 × 8 image is shown in Fig. 8(a). The numbers (0...63) within each grid represent the addresses of pixels. The pixels placed in the first octant are loaded into a one-dimensional array h_1 [see Fig. 8(c)] following the direction of arrows as shown in Fig. 8(b). Expressions $[h_1(0)\dots h_1(7)]$ within the grids of h_1 are pixel values corresponding to the addresses [28,29,30,31,21,22,23,14]. To ensure that the array h_2, after reordering the pixels in the second octant, still has the symmetric relationship with array h_1, the pixels in the second octants must be loaded following the addresses [28,20,12,4,21,13,5,14] sequentially. The loaded sequences in the other six octants follow the addresses below the arrays $h_3 - h_8$.

In this case, the diagonal pixels are loaded repetitively. If we ignore the duplicate pixels, the computational result will be wrong. However, judging the duplicate pixels entails additional overhead. To eliminate the effect of repetitive computations, we design an ingenious scheme: the pixel values on the diagonal in arrays $h2$, $h3$, $h6$, and $h7$ are set as zero in advance. Finally, the diagonal pixels are loaded only once, but we ensure that the dimension of all 8 arrays is the same by addressing the diagonals in advance. The image pixels are reordered into eight one-dimensional arrays as shown in Fig. 8(c). If we pay attention to every column, the same indices in the arrays are exactly the correct symmetric points after re-layout. We can take advantage of the same index in the arrays to perform parallel computation using Eqs. (16) and (17) (see Fig. 9).

After data re-layout of the image, the limiting factors mentioned before are eliminated. First, the image pixels are reordered once to avoid wrong repetitive summations. Then, the image data are converted into 8 one-dimensional image information after combining the reordered data with addressing the diagonal in advance. This ensures that the same index of the 8 arrays is naturally symmetric points, so that we do not need any conditional instruction to extract symmetric points in the subsequent calculation. Moreover, the uncalled pixels placed outside the circle [such as 0, 1, 8 in Fig. 8(a)] are avoided.

The reordering sequence is the key to reordering image data. However, reordering is performed in a non-coalesced fashion; hence, it is time consuming (see Section 5.1). Nevertheless, it is executed only once for an image even if just one set of ZMs is obtained. The experiments in Section 5 prove that the cost is worthwhile as compared with the baseline implementation.

4.2.2. *Optimization of Zernike radial polynomials*

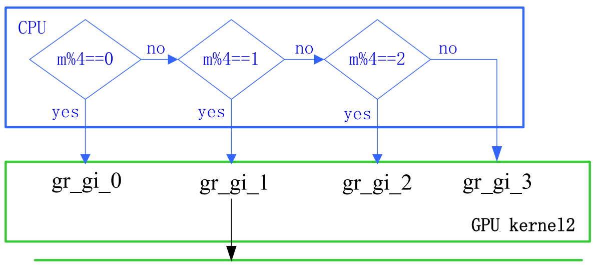
There are massive factorial terms in radial polynomials [see Eq. (3)]. We pre-calculate the value of the factorial function required for computing Eq. (3). The factorial values are stored in the constant memory to ensure fast access from threads. By simply storing the factorial values in the constant memory, we also avoid factorial computations. A high order can lead to more computational time; thus, we save more time via stored factorial values.

4.3. *Implementation of the optimization strategy*

First, we pre-calculated the value of the factorial required for computing Eq. (3). In our experiments, the maximum order for the ZMs is 34; thus, there are 34 values (0!, 1!, 2!, . . . , 34!) in the constant memory.

Second, substituting the values of the address index of the loaded image in the first octant into Eqs. (14) and (15) results in 2 one-dimensional values of ρ and θ. It should be noted that the way of access must correspond to the sequence shown in Fig. 8(b), so that the coordinates of same index in the two arrays are symmetric [as in the 8×8 image shown in Fig. 8(d)]. The required ρ and θ were pre-calculated and stored in the global memory.

Finally, four kernels are implemented for ZM with n order and m repetition in the GPU.



```
FUNCTION gr_gi_1(m, h1, h2, h3, h4, h5, h6, h7, h8, $\theta$)
tid = get_thread_id()
gr[tid] = (h1[tid] - h4[tid] - h5[tid] + h8[tid]) * cos(m * $\theta$[tid])
          + (h2[tid] - h3[tid] - h6[tid] + h7[tid]) * sin(m * $\theta$[tid])
gi[tid] = (h1[tid] + h4[tid] - h5[tid] – h8[tid]) * sin(m * $\theta$[tid])
          + (h2[tid] + h3[tid] - h6[tid] + h7[tid]) * cos(m * $\theta$[tid])
return  gr, gi
```

Fig. 9. Schematic of instruction switching in CPU call to GPU kernel.

```
FUNCTION octant_Zernike_moment_map(n, m, gr, gi, $\rho$)
tid = get_thread_id()
radial = radial_polynomial($\rho$[tid], n, m)
zr_map[tid] = radial * gr[tid]
zi_map[tid] = radial * gi[tid]
return  zr_map, zi_map
```

Fig. 10. Pseudocode for computing ZM map.

Kernel 1: Re-layouting of the image

The gray values of the pixels located in the 8 octants are arranged into $h1$, $h2$,..., $h8$, respectively by the re-layout. The total number NP of the pixels located in the first octant needs to be pre-calculated. It is used to define the array size of h1, ..., h8.

Kernel 2: Computing octant images on GPU

The octant images [see Section 4.1] are divided into 4 cases using Eq. (17). To avoid conditional instruction, we execute the switch control instruction in CPU and call the corresponding GPU kernel to obtain the octant images, gr_gi_0, gr_gi_1, gr_gi_2 and gr_gi_3, where the remainder of repetition m divided by 4 is 0, 1, 2 and 3, respectively. Fig. 9 shows the schematic, the switch control instruction in CPU, and the pseudocode when the remainder of repetition m divided by 4 is 1. Here, $g_m^r(x, y)$ and $g_m^i(x, y)$ are denoted by gr and gi, respectively.

Kernel 3: Computing the ZM map

The ZM map is computed using the octant images and radial polynomials in Eq. (3). It occupies most of the computational time of the ZMs .Fig. 10 shows the pseudocode for computing the ZM map; tid is also the index of a thread . The real and imaginary parts of the ZM map are denoted by zr_map and zi_map, respectively. The pseudocode is simple and efficient owing to the data re-layout. It ensured coalesced memory accesses, while avoiding the use of a conditional instruction .

Kernel 4: Computing the ZMs

Finally, all the computed values of the ZM map have to be accumulated in order to obtain a moment. This type of operator has been thoroughly studied as a parallel reduction for its CUDA implementation. There are two methods for implementing the parallel reduction. The method presented by Mark Harris [21] is the most popular CUDA implementation for parallel reduction. It has been included as project examples in every CUDA SDK release. The thrust API, which in fact is also designed by Mark Harris provides a simple interface that hides all complexities of a reduction, making it both flexible and easy to use. Though the dynamic memory allocation of space slows down the thrust implementation, the difference in execution time between the two methods is just slight [9,37].

Table 4
Summary of hardware features for Geforce 9800GX2 and Tesla K40 during our experimental runs.

Processor	GeForce9800GX2	Tesla K40
CUDA cores	256 (128 per GPU)	2880
Processor clock	1500 MHz	745 MHz
Memory bandwidth (GB/sec)	128 (64 per GPU)	288
Memory size (Type)	1 GB(DDR3)	12 GB(DDR5)
Floating-point arithmetic	32 bits	32 bits /64 bits

5. Experimental results and discussion

We provide a quantitative analysis of computational time in this section. To confirm the proposed method which is valid throughout all generations of GPU, two GPUs from different generations were used to analyze the performance of the proposed method. First, in order to unify comparisons with Martín's strategy as presented in [22], the NVIDIA GeForce 9800GX2 was selected. It is the first generation that support CUDA and it includes two GPU chips distinctively. Because GeForce9800GX2 is a single-precision floating-point , the maximum order for the ZMs is set as 34 because it is the limit imposed by the hardware for the calculation of the factorial numbers that are used within the Zernike polynomials. New generations of GPUs, which support double-precision floating-point, can compute ZMs with the order greater than 34.

Moreover, NVIDIA Tesla K40, which is a popular Kepler architecture GPU, was chosen. Table 4 summarizes its hardware features. We used CUDA 6.5 release for our GPU implementation.

5.1. Reordering cost

As is known to all, to obtain the maximum memory throughput, the global memory access must be coalesced. Coalescing a memory access refers to combining multiple memory accesses into a single transaction. That is, every successive segment memory can be accessed by a warp (the minimum scheduled unit) in a single transaction. In addition, the coalesced memory accesses are sequential and aligned. The proposed reading direction to reorder an image is in a non-coalesced fashion [see Fig. 7]. To reduce the un-coalesced computational time, the reordered addresses ($h_1 - h_8$) are pre-stored in the global memory, which can be searched to avoid complex executions from traversing data to extract symmetric points using conditional statements. This form of lookup table gains a slight improvement against direct image reordering as shown in Table 5.

However, there is a trade-off: we adopt an un-coalesced reordering of the image only once, but we eliminate conditional instruction and achieve memory coalescing sufficiently in the subsequent calculation. It is emphasized that fixed image reordering is executed only once and all our experimental results include a reordering time.

5.2. Computation of ZMs on GetForce9800 GX2

The proposed approach mainly focuses on the optimization of the Zernike radial polynomials and the efficient implementation of the symmetric algorithm. To examine the implementation of symmetric algorithm in detail, we computed Z_{00} on different image sizes by only using our symmetric method through re-layout in the

Table 5
Time of reordering (in μs) for different image sizes on two GPUs(NVIDIA Geforce 9800GX2 and Tesla K40).

Image size	Directly calculated		Lookup table	
	9800	K40	9800	K40
64×64	20	20	13	12
128×128	42	21	26	15
256×256	160	25	117	20
512×512	550	57	416	45
1024×1024	2340	190	1820	140

Table 6
Execution time (in ms) computing a single ZM, Z_{00}, for different image sizes on Geforce 9800GX2. A: our symmetric optimization vs. Martín's baseline; B: our proposed approach vs. Martín's baseline.

Image size	Martín's baseline	Martín's symmetric method	Our symmetric method(A)	Our proposed method(B)
64	0.12	0.20	0.071(1.7×)	0.065(1.8×)
128	0.18	0.28	0.084(2.1×)	0.076(2.3×)
256	0.37	0.58	0.175(2.1×)	0.149(2.5×)
512	1.14	1.73	0.487(2.3×)	0.393(2.9×)
1024	4.19	6.69	1.956(2.1×)	1.537(2.7×)

Table 7
Execution time (in ms) for a 256×256 image when a single ZM, $Z_{n,m}$, is computed on Geforce 9800GX2.

Single ZM	Martín's baseline implementation	Proposed	Proposed vs. Martín's baseline
$Z_{0,0}$	0.37	0.15	2.46×
$Z_{6,2}$	0.47	0.16	2.93×
$Z_{12,0}$	0.69	0.18	3.83×
$Z_{25,13}$	0.71	0.19	3.74×

Table 8
Execution time (in ms) in computing ZMs of all repetitions of a given order for a **1024** ×**1024** image on Geforce 9800GX2. The speed-up is proposed on a single GPU and two GPUs vs. Martín's baseline.

Set of ZMs for a given order	Single GPU		Two GPUs
	Martín's baseline	Proposed	Proposed
$Z_{4,*}$(3repetitions)	12.89	2.35(5.48×)	1.35(9.54×)
$Z_{8,*}$(5repetitions)	25.09	3.02(8.30×)	1.76(14.2×)
$Z_{12,*}$(7 repetitions)	40.97	3.96(10.34×)	2.29(17.9×)
$Z_{16,*}$(9 repetitions)	60.48	5.18(11.67×)	2.97(20.3×)
$Z_{20,*}$(11 repetitions)	83.71	6.63(12.62×)	3.81(21.9×)
$Z_{24,*}$(13 repetitions)	110.56	8.45(13.08×)	4.79(23.1×)
$Z_{28,*}$(15 repetitions)	141.03	10..40(13.56×)	5.91(23.86×)
$Z_{32,*}$(17 repetitions)	175.13	12.74(13.74×)	7.1(24.66×)

first experiment. The execution time against Martín's scenario is shown in Table 6. Our symmetric method for Z_00 provides a speed-up of approximately 2× against Martín's baseline implementation. We did not compare with Martín's symmetric method, because that method does not improve the execution time compared to the baseline implementation. Thus, the data re-layout is effective even for $Z_{0,0}$, which is the minimum amount of the computation in all single ZM calculations. After combining the optimization of the Zernike radial polynomial, we obtained the execution time of the proposed approach. It was found that using constant memory is beneficial.

After comparing the execution time for computing $Z_{0,0}$, we now illustrate optimized computation of a set of single moments for a 256×256 image using the proposed approach. The typical scenario was used in Martín's biomedical application as the vector of features for classifying images. The performance is shown in Table 7. The overhead of reordering 256×256 image is 0.117 ms. Although the overhead is a larger proportion in the overall runtime, the speed-up of the proposed approach against Martín's baseline implementation further increases to approximately 3×.

The next step executes computing for all repetitions of ZMs of a given order for a 1024×1024 image. As the number of CUDA threads is multiplied with the overhead of re-layout only once, significant speed-up performances (speed-up of 5.48 minimum to 13.74 maximum) are reported in the columns for a single GPU as shown in Table 8.

Furthermore, we only need a simple partitioning mechanism to allocate the data flow into multi-GPUs in terms of the re-layout scheme. This is because the same index of arrays are naturally symmetric points in the re-layout arrays. We reordered the image into the two parts using 2 GPUs as shown in Fig. 11. Thus, the image data are allocated in 2 GPUs and the number of CUDA threads reduces to half threads. The execution time obtained by the partition mechanism is in the columns for two GPUs as shown in Table 8. The speed-up of the proposed approach against Martín's baseline implementation further increases to a 9.54 minimum to 24.66 maximum. The results have demonstrated that the simple partitioning mechanism avoids introducing conditionals or dependencies, which are harmful for GPU performance. Therefore, our proposed approach easily supports the expansion of ZM computations on multi-GPUs merely by simple data partitioning.

5.3. Computation of ZMs on K40

In this part, we carry out ZM computations by varying the image size from 64×64 to 1024×1024 on K40, which has 2880 streaming processors. Tables 9 and 10 list the execution time for a set of single moments and moments of all repetitions of ZMs of a given order as the image size increases, respectively.

In order to visually and briefly compare two GPUs from different generations, we plot the execution time of the ZMs, in which the order n is from 4 to 32 on a 1024×1024 image as shown in Fig. 12.

Table 9
Execution time (in μs) for a single ZM, $Z_{n,m}$ for different image sizes on K40.

Single ZMs	Baseline on K40					Proposed on K40(Baseline vs. Proposed)				
	64	128	256	512	1024	64	128	256	512	1024
$Z_{0,0}$	70	78	109	241	448	69(1.01×)	80(1.02×)	86(1.26×)	115(2.1×)	228(2.0×)
$Z_{6,2}$	80	90	133	305	667	77(1.03×)	85(1.05×)	92(1.44×)	124(2.5×)	255(2.6×)
$Z_{12,0}$	93	103	176	447	1090	89(1.04×)	93(1.10×)	99(1.77×)	137(3.3×)	293(3.7×)
$Z_{25,13}$	98	115	203	533	1505	93(1.05×)	97(1.18×)	104(1.95×)	143(3.7×)	302(5.0×)

Table 10
Execution time (in ms) for computing ZMs of all repetitions of a given order for different image sizes on K40.

Set of ZMs for a given order	Baseline on K40					Proposed on K40(Baseline vs. Proposed)				
	64	128	256	512	1024	64	128	256	512	1024
$Z_{4,*}$(3repetitions)	0.21	0.22	0.28	0.59	1.45	0.19(1.1×)	0.20(1.1×)	0.21(1.3×)	0.26(2.3×)	0.42(3.5×)
$Z_{8,*}$(5repetitions)	0.35	0.42	0.51	1.18	3.16	0.32(1.1×)	0.35(1.2×)	0.35(1.2×)	0.43(2.7×)	0.68(4.6×)
$Z_{12,*}$(7 repetitions)	0.53	0.56	0.84	1.85	5.65	0.49(1.1×)	0.50(1.1×)	0.51(1.6×)	0.62(3.0×)	1.00(5.7×)
$Z_{16,*}$(9 repetitions)	0.71	0.78	1.23	2.87	9.10	0.66(1.1×)	0.67(1.2×)	0.67(1.8×)	0.80(3.6×)	1.36(6.6×)
$Z_{20,*}$(11 repetitions)	0.96	1.03	1.71	4.20	13.66	0.83(1.2×)	0.84(1.2×)	0.85(2.0×)	1.03(4.0×)	1.79(7.6×)
$Z_{24,*}$(13 repetitions)	1.31	1.30	2.28	5.84	19.41	1.02(1.2×)	1.03(1.3×)	1.03(2.2×)	1.26(4.6×)	2.27(8.5×)
$Z_{28,*}$(15 repetitions)	1.56	1.63	2.99	7.85	26.52	1.29(1.2×)	1.23(1.3×)	1.21(2.5×)	1.52(5.2×)	2.80(9.4×)
$Z_{32,*}$(17 repetitions)	2.21	2.21	3.79	10.30	35.20	1.40(1.5×)	1.42(1.5×)	1.42(2.7×)	1.79(5.8×)	3.41(10.3×)
$Z_{34,*}$(18 repetitions)	1.93	2.55	4.57	11.69	40.20	1.52(1.3×)	1.50(1.7×)	1.53(3.0×)	1.94(6.0×)	3.76(10.7×)

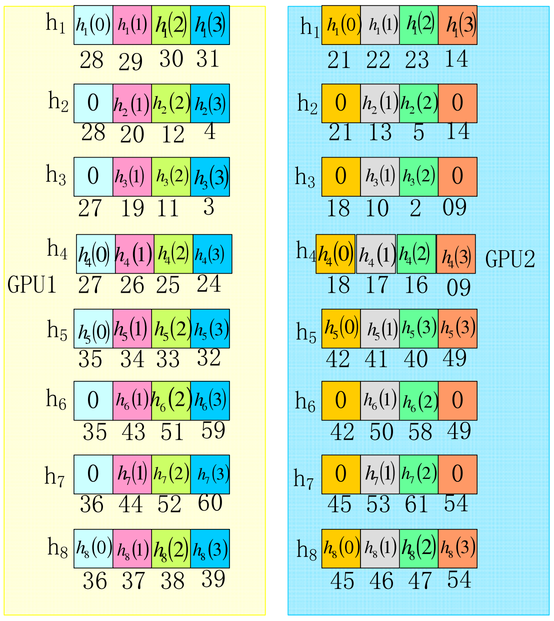


Fig. 11. An 8×8 image data partition on 2 GPUs of Geforce 9800GX2.

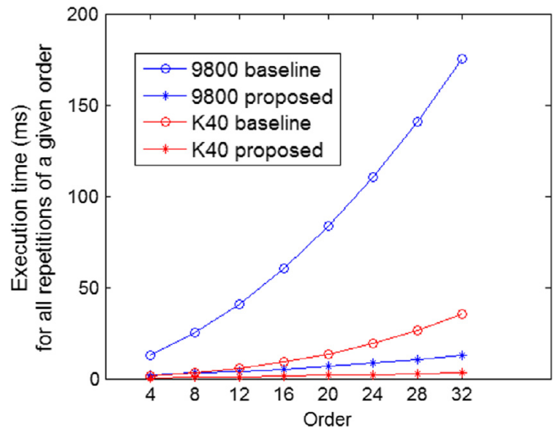


Fig. 12. Execution time of ZMs for a 1024 × 1024 image on two GPUs from different generations from order 4 to 32.

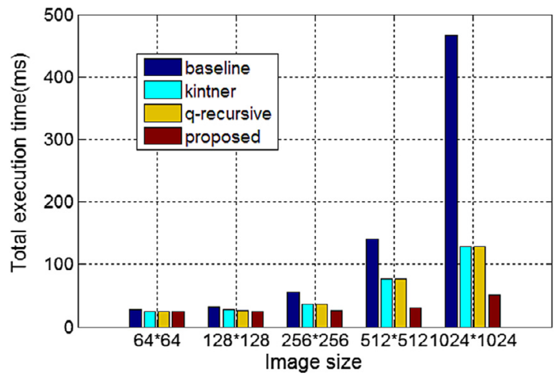


Fig. 13. Total running time for computing ZMs of all repetitions within order 34 on different image sizes on K40.

It is clear that the advanced K40 hardware leads to a reduction of the overall elapsed times against Geforce 9800GX2. There is a significant acceleration even for the baseline implementation on K40. We notice that the proposed approach almost does not offer an obvious improvement on K40 when the order n is low. The reason is the basic cost, which includes the data reading and reordering overhead. The ratio of the basic cost to the overall runtime greatly increases when the order n is low.

In the end, we implement Kintner's [17] and Chong's [2] methods, which are widely used in accelerating ZMs. Their algorithms eliminate the factorial operation by recurrence relations. In order to visually analyze the performance of all the methods on different image sizes on K40, we draw a bar graph [see Fig. 13], which contains the total running time for computing ZMs for all repetitions within order 34. Our proposed approach is significantly better than Kintner's and Chong's when the size of input image is 512 and 1024. Simultaneously, we also observe that all the methods only offer slight improvements against the baseline when the size of the input image is less than 512. The reason can be explained by the occupancy, which is the ratio of the number of active threads per multiprocessor to the maximum number of possible active threads. A low occupancy always interferes with the ability to hide memory latency, resulting in performance degradation [4].

There are 45 multiprocessors in K40, and each multiprocessor at most performs concurrently 2048 threads, corresponding to

Table 11
Number of threads and occupancy on K40 using our proposed approach for different images.

Image size	Total threads	Active threads per multiprocessor	Occupancy
64 × 64	4 × 128	128	6.25%
128 × 128	16 × 128	128	6.25%
256 × 256	64 × 128	256	12.5%
512 × 512	256 × 128	768	37.5%
1024 × 1024	1024 × 128	2048	100%

occupancy of 100%. Typically, once an occupancy of 50% has been reached, additional increases in occupancy do not translate into improved performance [4]. Therefore, each multiprocessor must at least perform concurrently 1024 threads if we need to reach an occupancy larger than of 50%. In our proposed approach, 128 threads per block is used. Hence the number of active threads per multiprocessor should be a multiple of 128 threads. The occupancy of the proposed approach on K40 is shown in Table 11. It can be observed that occupancy is low when the image size is less than 512. The total number of blocks in a grid is less than the numbers of multiprocessors, particularly when the images sizes are 64 or 128; thus, some multiprocessors do not even get a block resulting in waiting and idle time. In summary, low occupancy with smaller images becomes the bottleneck of acceleration. On the other hand, the occupancy of a 512 × 512 image has been significantly increased and tend to be 50%, while as speed-up of 4.7× is reached against the baseline. An occupancy of 100% with a 1024 × 1024 image means that the GPU is kept busy and the speed-up of 9.1× is obtained.

6. Conclusion

In this study, our work focused on the acceleration of GPU execution time for computing ZMs. Its major contributions are (1) efficiently exploiting the symmetry of the ZMs by data re-layout, which involves reordering of image pixels and addressing the diagonal pixels in advance and (2) leveraging the constant memory to store the pre-computed factorial values.

The implemented re-layout maximized the GPU performance: it coalesced the access to the global memory and avoided huge thread divergence that extracts symmetric points. The reordering sequence is the key to image data reordering. Although it is non-coalesced fashion and requires additional overhead, the overall obtained speed-up is sufficiently good. In particular, in practice, one set of ZMs are necessary to descript image features. In that case, the speed-up is multiplied, as a result of which, the execution time reduces significantly because the re-layout is executed only once.

Our programs run on Geforce 9800GX2. The experimental results demonstrated that the scheme of data re-layout efficiently exploited the symmetry of the ZMs. Then, we fully compared our experiments with the GPU implementation of Martin. The results indicated that our proposed method is significantly superior to that of Martin. Hence, our approach is applicable even to low performance computing hardware like Geforce 9800GX2. Moreover, NVIDIA embedded platforms such as Jetson TK1 have lower cost and power consumption advantages compared with other desktop GPU cards. High performance computing application of ZMs in embedded platforms can also be used for our approach [29,42].

Simultaneously, we carry out the implementation of our proposed approach, baseline, Kintner's, and q-recursive on K40, which is a high performance computing hardware. Our proposed method achieved a higher performance than the above mentioned methods when the input image size is larger than 256.

We analyzed the bottleneck in the implementation for small images and found that low occupancy leads to the small improvement. Future work is to research an approach that can recovers some of the lost performance and wasted resources in small-size images. That is, concurrent execution of GPU kernels should be exploited to keep all GPU resources busy. A packing-image strategy might be extremely useful for increasing occupancy. A similar approach is mentioned in the new generation NVIDIA GPUs by Hyper-Q technology, which allows 32 independent, hardware-managed work queues [20].

Regarding the methodology applied, there are two remarkable contributions. One is that it is of paramount importance to reorder the image as one-dimensional array if the image processing area is fan-shaped or other irregular shapes. The advantage of this image re-layout is that it effectively eliminates the conditional statements required when special pixels are extracted. Image reordering also achieves memory coalescing in the subsequent calculations.

Another comment is that computational time can be saved by exploiting the GPU memory space. The data that can be calculated in advance are stored in the GPU memory, which can later be extracted by searching in the form of lookup tables. This largely increases the computational efficiency. The constant memory storing strategy is particularly efficient in minimizing the repeated execution of many highly frequent computations, saving huge amounts of computational time. In short, it is a strategy that trades memory space for computation time.

Acknowledgments

This work is supported by the National Natural Science Foundation of China with Grant Numbers of 61405194, 11604327 and 21571080.

The project was supported by the State Key Laboratory of Applied Optics, Changchun Institute of Optics, Fine Mechanics and Physics, Chinese Academy of Sciences.

References

[1] A. Bouziane, Y. Chahir, Unified framework for human behaviour recognition: An approach using 3D Zernike moments, Neurocomputing 100 (2013) 107–116.
[2] C. Chong, P. Raveendran, R. Mukundan, A comparative analysis of algorithms for fast computation of Zernike moments, Pattern Recognit. 36 (2003) 731–742.
[3] Chun-WeiTan, Accurate iris recognition at a distance using stabilized iris encoding and zernike moments phase features, IEEE Trans. Image Process. 23 (2014) 3962–3972.
[4] CUDA C Best Practices Guide, http://docs.nvidia.com/pdf/CUDA_C_Best_Practices_Guide.pdf.
[5] Gili Dardikman, Video-rate processing in tomographic phase microscopy of biological cells using CUDA, Opt. Express 24 (2016) 1839–11854.
[6] An-Wen Deng, Stable, fast computiation of high-order Zernike moments using a recursive method, Pattern Recognit. 56 (2016) 16–25.
[7] S.M. Elshoura, D.B. Megherbi, Analysis of noise sensitivity of Tchebichef and Zernike moments with application to image watermarking, J. Vis. Commun. Image Represent 24 (2013) 567–578.
[8] Exploiting Byunghyun Jang, Exploiting memory access patterns to improve memory performance in data-parallel architectures, IEEE Trans. Parallel Distrib. Syst. 22 (2011) 105–118.
[9] Rob Farber, CUDA Application Design and Development, Morgan Kaufmann, 2011, pp. 111–131, 133–145.
[10] Sajad Farokhi, Near infrared face recognition by combining Zernike moments and un-decimated discrete wavelet transform, Digit. Signal Process. 31 (2014) 13–27.

[11] J. Gu, H.Z. Shu, C. Toumoulin, L.M. Luo, A novel algorithm for fast computation of Zernike moments, Pattern Recognit. 35 (12) (2002) 2905–2911.
[12] S.K. Hwang, Mark Billinghurst, Local descriptor by Zernike Moments for real-time keypint matching, Congr. Image Signal Process. (2008) 781–785.
[13] S.K. Hwang, W.Y. Kim, A fast and efficient method for computing ART, IEEE Trans. Image Process. 15 (2006) 112–116.
[14] Hwang, W. Kim, A Novel approach to the fast computation of Zernike moments, Pattern Recognit. 39 (2006) 2065–2076.
[15] H.S. Kim, H.K. Lee, Invariant image watermark using Zernike moments, IEEE Trans. Circuits Syst. Video Technol. 13 (2003) 766–775.
[16] W.Y. Kim, Y.S. Kim, A region-based shape descriptor using Zernike moments, Signal Process., Image Commun. 16 (2000) 95–102.
[17] E.C. Kintner, On the mathematical properties of the Zernike polynomials, Opt. Acta 8 (23) (1976) 679–680.
[18] Dayu Li, Lifa Hu, Wavefront processor for liquid crystal adaptive optics system based on graphics processing unit, Opt. Commun. 316 (2014) 211–216.
[19] S. Li, M.-C. Lee, P. Chi-Man, Complex Zernike moments features for shape based image retrieval, IEEE Trans. Syst. Man Cybern. 39 (2009) 227–237.
[20] Ryan S. Luley, Effective utilization of CUDA Hyper-Q for improved power and performance efficiency, in: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops, pp. 1160–1168.
[21] Mark Harris, Optimizing Parallel Reduction in CUDA http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf.
[22] Martín-Requena, P. Moscato, M. Ujaldón, Efficient data partitioning for the GPU computation of moment functions, J. Parallel Distrib. Comput. 74 (2014) 1994–2004.
[23] Jarno Mielikainen, GPU compute unified device architecture (CUDA)-based parallelization of the RRTMG shortwave rapid radiative transfer model, IEEE J. Sel. Top. Appl. Earth Obs. Remote Sens. (2016) 921–931.
[24] R. Mukundan, K.R. Ramakrishnan, Fast computation of Legendre and Zernike moments, Pattern Recognit. 28 (9) (1995) 1433–1442.
[25] M. Novotni, R. Klein, Shape retrieval using 3D Zernike descriptors, Computer Aided Des. 36 (2004) 1047–1062.
[26] NVidia, CUDA C Programming Guide, http://docs.nvidia.com/cuda/index.html.
[27] G.A. Papakostas, Y.S. Boutalis, D.A. Karras, B.G. Mertzios, A new class of Zernike moments for computer vision applications, Inform. Sci. 177 (2007) 2802–2819.
[28] A. Prata, W.V.T. Rusch, Algorithm for computation of Zernike polynomials expansion coe3cients, Appl. Opt. 28 (1989) 749–754.
[29] Xuan Qi, Chen Liu, Stephanie Schuckers, Key-frame analysis for face related video on GPU-accelerated embedded platform, in: International Conference on Computational Science and Computational Intelligence, 2016, pp. 682–687.
[30] Paul Rosen, A visual approach to investigating shared and global memory behavior of CUDA kernels, in: Eurographics Conference on Visualization, EuroVis 2013, vol. 32, pp. 161–170.
[31] C. Singh, Improved quality of reconstructed images using floating point arithmetic for moment calculation, Pattern Recognit. 39 (2006) 2047–2064.
[32] C. Singh, Pooja, Improving image retrieval using combined features of Hough transform and Zernike moments, Opt. Lasers Eng. 49 (2011) 384–1396.
[33] Chandan Singh, Fast and accurate method for high order Zernike moments computation, Appl. Math. Comput. 218 (2012) 7759–7773.
[34] Chandan Singh, Neerja Mittal, Ekta Walia, Face recognition using zernike and complex zernike moment features, Pattern Recognit. Image Anal. 21 (2011) 71–81.
[35] C. Singh, E. Walia, Rotation invariant complex Zernike moments features and their applications to human face and character recognition, IET Comput. Vis. 5 (5) (2011) 255–266.
[36] M.R. Teague, Image analysis via the general theory of moments, J. Opt. Soc. Amer. 70 (8) (1980) 920–930.
[37] Thrust_Quick_Start_Guide, http://docs.nvidia.com/cuda/pdf/Thrust_Quick_Start_Guide.pdf.
[38] Toharia, Shot boundary detection using Zernike moments in multi-GPU multi-CPU architectures, J. Parallel Distrib. Comput. 72 (2012) 1127–1133.
[39] Ujaldon, GPU acceleration of Zernike moments for large-scale images, in: 23rd IEEE International Parallel and Distributed Processing Symposium, IEEE Computer Society, 2009, pp. 23–29.
[40] A. Wiliem, V.K. Madasu, W. Boles, P. Yarlagadda, A face recognition approach using Zernike moments for video surveillance, in: Proc. RNSA Security Tech. Conf. Melbourne, Australia, 28 September 2007, pp. 341–355.
[41] Yongqing Xin, Image reconstruction with polar Zernike moments, in: S. Singh, et al. (Eds.), ICAPR 2005, in: LNCS, vol. 3687, 2005, pp. 394–403.
[42] Kuan-Yu Yeh, Constructing a GPU cluster platform based on multiple NVIDIA Jetson TKI, in: IEEE International Conference on Bioinformatics and Biomedicine, 2016, pp. 917–922.
[43] F. Zernike, Beugungstheorie des schneidenverfahrens und seiner verbesserten form, der phasenkontrastmethode, Physica 1 (1934) 689–704.

Yubo Xuan is a lecturer at College of Communication Engineering, Jilin University, China, since 2007. She received her B.S. degree from Northeast Normal University and the M.S. degree from Jilin University (China) in 2002 and 2005 respectively. Her current research interests are parallel computer system and image processing.

Dayu Li is an Associate Professor at State Key Laboratory of Applied Optics, Changchun Institute of Optics, Fine Mechanics and Physics, Chinese Academy of Sciences, China, since 2010. He received his B.S. degree from Jilin University and the Ph.D. degree from Chinese Academy of Sciences in 2002 and 2007 respectively (China). His current research interests are GPU high performance computing and wavefront processor of adaptive optics system.

Wei Han is a Professor at College of Physics, Jilin University, China, since 2001. He received his B.S degree from Jilin University in 1989 (China) and the Ph.D. degree from Tomsk Polytechnic University in 1997(Russia). His current research field includes: application of nanomaterial in green energy technologies, electronics, control and parallel computer system. He has published about 100 papers in international referred journals, including Advanced Energy Materials, ACS Nano and Nanoscale.