3-D Image Analysis via Jacobi Moments with GPU-Accelerated Algorithms

by

Puwei Wang

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

Department of Applied Computer Science
The University of Winnipeg
Winnipeg, Manitoba, Canada
December 2021

Copyright © 2021 Puwei Wang

Abstract

This research has developed a parallel algorithm to compute 3-Dimensional Jacobi moments with high efficiency and accuracy. The algorithm was implemented in CUDA C. Our developing progress was in the order of Legendre moments, Gegenbauer moments, and Jacobi moments investigated on the 2-D image. Then, we extended research from 2-D to 3-D image. To verify the algorithm's performance, we have implemented image reconstruction from higher orders up to 500 on testing image sized at $512 \times 512 \times 512$. The experiment was deployed on Nvidia Tesla V100, which restrained computational time within 400 milliseconds, and the PSNR value of reconstructed image reached up to 53.6382.

Keywords: Fast moment computing, Jacobi moments, 3-D image reconstruction, GPU acceleration.

Contents

1. Introduction **8**

2. Orthogonal Moments **10**

2.1 Legendre Moments 10

2.2 Gegenbauer Moments 11

2.3 Jacobi Moments 12

2.4 3-D Jacobi Moments 16

3. Improving Accuracy and Efficiency **17**

3.1 $k \times k$ sub-regions 17

3.2 Matrix Multiplication 19

3.3 Matrix-Cuboid Multiplication 22

4. General Purpose of CUDA Computing **27**

4.1 Memory Overview 27

4.2 Memory limitation 28

4.3 Coalesced Memory Access 29

4.4 Barrier Synchronization 30

5. Implementation of GPU-Accelerated Algorithms **32**

5.1 Parallel Polynomial Computation 32

5.2 Parallel Matrix Transpose 34

5.3 Parallel Matrix-Cuboid Multiplication 35

5.4 Implementation on CPU 36
5.5 Performance of GPU-Accelerated Algorithm on 2-D Jacobi Moment Computing 37

6. Experimental Results and Analysis **39**
6.1 3-D Image Reconstruction 40
6.1.1 $\alpha = 0.3$ and $\beta = 0.3$ 42
6.1.2 $\alpha = 0.3$ and $\beta = 0.7$ 44
6.2 Image Slicing and Clipping 46
6.2.1 $\alpha = 0.3$ and $\beta = 0.3$ 47
6.2.2 $\alpha = 0.3$ and $\beta = 0.7$ 49

7. Conclusion and Future Work **51**

List of Tables

1 Summary of Features of different Memory. 28

2 The computational time(ms) of $1,024 \times 1,024$ image reconstructions via the 1000-th order of Jacobi moments with $\alpha = 0.3$ and $\beta = 0.3$ between our GPU-based algorithm. 38

3 The computational time, in millionseconds, of computing Jacobi moments of Figure 13 with $\alpha = 0.3$ and $\beta = 0.3$ and performing the image reconstructions from different maximum orders and $k \times k \times k$ schemes in System II. 42

4 The PSNR values of the reconstructed Figure 13 from applying different maximum orders of Jacobi moments computed by using $\alpha = 0.3$ and $\beta = 0.3$. 42

5 The computational time, in millionseconds, of computing Jacobi moments of Figure 13 with $\alpha = 0.3$ and $\beta = 0.7$ and performing the image reconstructions from different maximum orders and $k \times k \times k$ schemes in System II. 44

6 The PSNR values of the reconstructed Figure 13 from applying different maximum orders of Jacobi moments computed by using $\alpha = 0.3$ and $\beta = 0.7$. 44

List of Figures

1 Example of 3-D matrix multiplication 23

2 Diagram of 3 phases of matrix-cuboid multiplication in moments computing 24

3 An example of 3-D moments cuboid 25

4 Diagram of 3 phases of matrix-cuboid multiplication in image reconstruction 26

5 Overview of GPU memory[11] 27

6 General Comparison between two classic GPU 29

7 Uncoalesced memory access pattern[11] 29

8 Coalesced memory access pattern[11] 30

9 Diagram of Barrier Syncronization 30

10 Diagram of parallel matrix transpose 34

11 Example diagram of multiplication between matrix A and cuboid B with 2 tiling width 35

12 Diagram of computing array $\rho_m^{(\alpha,\beta)}$ 37

13 The tesing image of knee with size $512 \times 512 \times 512$ and 256 gray levels [24]. 40

14 Some reconstructed images of Figure 13 from different maximum orders of Jacobi moments with various $k \times k \times k$ numerical schemes and orders at $\alpha = 0.3$ and $\beta = 0.3$. 43

15 Some reconstructed images of Figure 13 from different maximum orders of Jacobi moments with various $k \times k \times k$ numerical schemes and orders at $\alpha = 0.3$ and $\beta = 0.7$. 45

16 Sliced image from reconstructed 3-D image on $x-y$, $y-z$ and $x-z$ by $M_{max} = 500$ and $k = 23$ at $\alpha = 0.3$ and $\beta = 0.3$. . . . 47

17 Clipped 3-D reconstructed image from Jacobi moments by $M_{max} = 500$ and $k = 23$ at $\alpha = 0.3$ and $\beta = 0.3$. 48

18 Sliced image from reconstructed 3-D image on $x-y$, $y-z$ and $x-z$ by $M_{max} = 500$ and $k = 23$ at $\alpha = 0.3$ and $\beta = 0.7$. . . . 49

19 Clipped 3-D reconstructed image from Jacobi moments by $M_{max} = 500$ and $k = 23$ at $\alpha = 0.3$ and $\beta = 0.7$. 50

List of Code Snippets

1 Polynomial Computing with k scheme Kernels 56

2 Polynomial Computing without k scheme Kernels 61

3 Symmetric Property and Matrix Transpose Kernels 63

4 Matrix-Cuboid Multiplication Kernels 64

5 Computation of $\rho_n^{(\alpha,\beta)}$ 74

Acknowledgements

During the period of study and research, many important people came up and helped me a lot. I take this opportunity to thank every professor, classmate, and staff working in the Department of Applied Computer Science from the University of Winnipeg. It is impossible to complete this thesis without your work.

First, I would like to thank Dr. Simon Liao for giving me this precious opportunity to complete my master's thesis under your supervision. Thanks for your ideas and every valuable suggestion to push this research forward. What's more, your tutoring encouraged me and guided me when I was lost in this research. All your mentoring will inspire me to overcome the difficulty in the rest of my life.

Also, I am willing to thank Dr. Christopher Henry for your excellent course that brought me to the world of GPU computing. The parallel matrix multiplication algorithm that I learned from your class is the basis of completing this thesis.

Then, I am grateful to all the members of my thesis committee, Dr. Blair Jamieson and Dr. Christopher Henry for their precious comments.

Last but not least, I want to thank my parents. Thank you for supporting me to chase the master's degree.

Chapter 1

Introduction

Since Hu first proposed the concept of digital image moments invariant[10], different types of continuous orthogonal moments defined in a rectangular region have been investigated as unique image features in many scientific fields, such as image analysis and pattern recognition. However, some computational issues have obstructed the further development of efficient applications driven by Legendre, Gegenbauer and Jacobi moment based techniques. The objective of this research is to exploit the feasible method that can address the compuational efficiency and accuracy of Jacobi moment.

As the basic one of orthogonal moments set defined in a rectangular region, Legendre moment has been investigated in early research since 1980 [28][29][15]. In recent years, 256×256 image reconstruction via Legendre moments was implemented in 2014[5]. Gegenbauer moments have drawn more attention popular in recent 20 years[4][14][8][13][7][2]. However, the Jacobi moment is studied limitedly due to its complexity[32][26]. In 2018, a CPU-based parallel matrix multiplication algorithm was implemented for image reconstruction via Legendre, Gegenbauer, and Jacobi moments which shortens the computational time within 5 seconds while image size and moments order are up to 1024 and 1000[20]. Although orthogonal moment computation has been developed over multiple decades by many researchers,

the result of applications' computational time remains on the seconds level due to the time-consuming computation.

In previous researches, there has commonly been a dilemma between improving efficiency and improving accuracy. In contrast, our parallel algorithm has addressed this issue. The mathematical approaches we utilized contain recurrent polynomial, $k \times k \times k$ numerical scheme, and symmetric properties. In addition, coalesced memory access, tiled matrix multiplication, and heterogeneous computation are considered for optimizing the performance of computation on GPU. By implementing parallel computation, the computational times of 1024×1024 sized and $512 \times 512 \times 512$ sized image reconstructions via Jacobi moments can be restrained within 20 and 400 milliseconds, respectively.

Chapter 2 will give a mathematical overview of Legendre, Gegenbauer, and Jacobi moments and their corresponding reconstruction functions. $k \times k$ numerical scheme, symmetric property, and matrix-cuboid multiplication approach will be introduced in Chapter 3. The conceptual overview of general purpose GPU computing and its optimization methods are introduced in Chapter 4. Parallel algorithms including polynomial computing, matrix transpose, and tiled matrix-cuboid multiplication, will be presented in Chapter 5. To verify the performance of our program, we have conducted a series of reconstructed images and digitalized verification in Chapter 6. Finally, Chapter 7 will conclude the entire paper.

Chapter 2

Orthogonal Moments

Applying a moment weighting kernel $\psi_{m,n}(x,y)$, the general two-dimensional continuous moment with the $(m+n)$-th order of an image function $f(x,y)$ defined in the rectangular region is given by

$$\Psi_{m,n} = \int_x \int_y \psi_{m,n}(x,y) f(x,y) dx dy, \tag{1}$$

where m, n are non-negative integers. When the kernel function $\psi_{m,n}(x,y)$ is an orthogonal polynomial, $\Psi_{m,n}$ are the set of orthogonal moments.

2.1 Legendre Moments

The n-th order Legendre polynomial is defined in Rodrigues-type format [25]

$$P_n(x) = \frac{1}{2^n n!} \frac{d^n}{dx^n} (x^2 - 1)^n, \tag{2}$$

with the recurrent formula

$$P_{n+1}(x) = \frac{2n+1}{n+1} \, x \, P_n(x) - \frac{n}{n+1} \, P_{n-1}(x), \tag{3}$$

where $P_0(x) = 1$ and $P_1(x) = x$.

The $(m+n)$-th order of Legendre moment of an image function $f(x,y)$

is defined on the square $[-1,1]\times[-1,1]$

$$\lambda_{m,n} = \frac{(2m+1)(2n+1)}{4} \int_{-1}^{+1}\int_{-1}^{+1} f(x,y)\, P_m(x)\, P_n(y)\, dxdy, \tag{4}$$

where $m,n = 0,1,2,....$

2.2 Gegenbauer Moments

The Gegenbauer polynomial, which is also called Ultraspherical polynomial, of degree α and order n is defined in the interval $[-1,1]$ as

$$G_n^{(\alpha)}(x) = \sum_{k=0}^{[n/2]} (-1)^k \frac{\Gamma(n-k+a)(2x)^{n-2k}}{\Gamma(a)k!(n-2k)!}, \quad \alpha > -0.5, \tag{5}$$

where $\Gamma(.)$ is the Gamma function, $[n/2]$ is either $(n-1)/2$ or $n/2$ for odd or even values of n, respectively.

The orthogonal Gegenbauer polynomial $G_n^{(\alpha)}(x)$ obeys the recursive relation

$$G_{n+1}^{(\alpha)}(x) = \frac{2(n+\alpha)}{n+1} x G_n^{(\alpha)}(x) - \frac{(n+2\alpha-1)}{(n+1)} G_{n-1}^{(\alpha)}(x) \tag{6}$$

with $G_0^{(\alpha)}(x) = 1$, and $G_1^{(\alpha)}(x) = 2\alpha x$.

The $(m+n)$-th order of 2-D Gegenbauer moments are defined as

$$A_{m,n} = \frac{1}{C_m^{(\alpha)} C_n^{(\alpha)}} \int_{-1}^{1}\int_{-1}^{1} f(x,y) G_m^{(\alpha)}(x) G_n^{(\alpha)}(y) w^{(\alpha)}(x) w^{(\alpha)}(y) dxdy. \tag{7}$$

where $C_n^{(\alpha)}$ is the normalization constant

$$C_n^{(\alpha)} = \frac{2\pi\Gamma(n+2a)}{2^{2\alpha}n!(n+a)[\Gamma(a)]^2}. \tag{8}$$

2.3 Jacobi Moments

The Jacobi polynomial, occasionally called hypergeometric polynomial, of the n-th order is defined via the hypergeometric function as follow [16]

$$P_n^{\alpha,\beta}(x) = \frac{(\alpha+1)_n}{n!}\,{}_2F_1(-n, 1+\alpha+\beta+n; \alpha+1; \frac{1-x}{2}), \tag{9}$$

where $\alpha, \beta \geq$ -1, the hypergeometric function ${}_2F_1$ is defined as

$${}_2F_1(a,b;c;z) = \sum_{n=0}^{\infty} \frac{(a)_n(b)_n}{(c)_n}\frac{z^n}{n!} = \frac{\Gamma(c)}{\Gamma(a)\Gamma(b)}\sum_{n=0}^{\infty}\frac{\Gamma(a+n)\Gamma(b+n)}{\Gamma(c+n)}\frac{z^n}{n!}, \tag{10}$$

and the Pochhammer symbol $(\alpha)_n$ is

$$(\alpha)_n = \alpha(\alpha+1)(\alpha+2), ..., (\alpha+n-1) = \frac{\Gamma(\alpha+n)}{\Gamma(\alpha)}, \tag{11}$$

with $n = 1, 2, 3, ...,$ and $(\alpha)_0 = 1$.

The Jacobi polynomial can also be written in Rodrigues-type format [32]

$$P_n^{(\alpha,\beta)}(x) = \frac{(-1)^n}{2^n n!}(1-x)^{-\alpha}(1+x)^{-\beta}\frac{d^n}{dx^n}\left[(1-x)^{n+\alpha}(1+x)^{n+\beta}\right]. \tag{12}$$

Likewise, Jacobi polynomial obeys the recurrence relation

$$P_n^{(\alpha,\beta)}(x) = \frac{1}{n(n+\alpha+\beta)}\left[\frac{2n-1+\alpha+\beta}{2}\left((2n+\alpha+\beta)x+\frac{\alpha^2-\beta^2}{2n-2+\alpha+\beta}\right)\right.$$
$$\left. P_{n-1}^{(\alpha,\beta)}(x) - \frac{(n-1+\alpha)(n-1+\beta)(2n+\alpha+\beta)}{2n-2+\alpha+\beta}P_{n-2}^{(\alpha,\beta)}(x)\right], \quad (13)$$

where $P_0^{(\alpha,\beta)}(x) = 1$, $P_1^{(\alpha,\beta)}(x) = \frac{1}{2}\left[\alpha - \beta + (\alpha+\beta+2)x\right]$.

For $\alpha \geq -1$ and $\beta \geq$ -1, a set of Jacobi polynomials satisfies the orthogonality condition [23]

$$\int_{-1}^{+1} w^{(\alpha,\beta)}(x)P_m^{(\alpha,\beta)}(x)\, P_n^{(\alpha,\beta)}(x)dx = \rho_n^{(\alpha,\beta)}\delta_{mn}, \quad (14)$$

with the weight function

$$w^{(\alpha,\beta)}(x) = (1-x)^\alpha(1+x)^\beta, \quad (15)$$

where δ_{mn} is the Kronecker symbol, and $\rho_n^{(\alpha,\beta)}$ is the normalization constant

$$\rho_n^{(\alpha,\beta)} = \frac{2^{\alpha+\beta+1}}{2n+\alpha+\beta+1}\frac{\Gamma(n+\alpha+1)\Gamma(n+\beta+1)}{\Gamma(n+\alpha+\beta+1)n!}, \quad (16)$$

and $\Gamma(.)$ is the Gamma function.

To simplify the computation, normalization constant $\rho_n^{(\alpha,\beta)}$ is transformed to recurrent formula

$$\rho_{n+1}^{(\alpha,\beta)} = \frac{(n+\alpha+1)(n+\beta+1)(2n+\alpha+\beta+1)}{(n+1)(n+\alpha+\beta+1)(2n+\alpha+\beta+3)}\rho_n^{(\alpha,\beta)}, \quad (17a)$$

$$\rho_0^{(\alpha,\beta)} = 2^{(\alpha+\beta+1)} \frac{\Gamma(\alpha+1)\Gamma(\beta+1)}{\Gamma(\alpha+\beta+2)}, \tag{17b}$$

with $n \geq 0$.

Additionally, the symmetric property of Jacobi polynomial is applied for convenient calculation when $\alpha = \beta$ [27]

$$P_n^{(\alpha,\beta)}(-x) = (-1)^n P_n^{(\beta,\alpha)}(x). \tag{18}$$

As a matter of fact, Jacobi polynomials are a class of classical orthogonal polynomials, which can represent the Legendre and Gegenbauer polynomials as their special cases[3]. For example,

$$P_n(x) = G_n^{(\frac{1}{2})}(x) = P_n^{(0,0)}(x), \tag{19}$$

and

$$G_n^{\lambda}(x) = \frac{\Gamma(\lambda+\frac{1}{2})\Gamma(n+2\lambda)}{\Gamma(2\lambda)\Gamma(n+\lambda+\frac{1}{2})} P_n^{(\lambda-\frac{1}{2},\lambda-\frac{1}{2})}(x). \tag{20}$$

The 2-D orthogonal $(m+n)$-th order of Jacobi moments are defined as

$$J_{m,n} = \frac{1}{\rho_m^{(\alpha,\beta)}\ \rho_n^{(\alpha,\beta)}} \int_{-1}^{+1}\int_{-1}^{+1} f(x,y) P_m^{(\alpha,\beta)}(x)\, P_n^{(\alpha,\beta)}(y) w^{(\alpha,\beta)}(x) w^{(\alpha,\beta)}(y) dx dy, \tag{21}$$

where $m, n = 0, 1, 2, ...$

Due to the orthogonality of the Jacobi moments, the image reconstruction

function of the image $f(x, y)$ from its Jacobi moments can be expressed as

$$f(x, y) = \sum_{m=0}^{\infty} \sum_{n=0}^{m} J_{m-n,n} \, P_{m-n}^{(\alpha,\beta)}(x) \, P_{n}^{(\alpha,\beta)}(y). \tag{22}$$

When Jacobi moments of certain order $\leq M_{max}$ are provided, the image function $f(x, y)$ can be approximated by a truncated series

$$f(x, y) \simeq f_{M_{max}}(x, y) = \sum_{m=0}^{M_{max}} \sum_{n=0}^{m} J_{m-n,n} \, P_{m-n}^{(\alpha,\beta)}(x) \, P_{n}^{(\alpha,\beta)}(y). \tag{23}$$

Since Legendre and Gegenbauer polynomials are the special cases of Jacobi polynomials, the Legendre and Gegenbauer moments can be seen as members of Jacobi moments. Therefore, in this research, we will focus our investigation on Jacobi moments. Furthermore, all results can be extended to Legendre and Gegenbauer moments by choosing the specified values of parameters α and β.

2.4 3-D Jacobi Moments

Referring to (21), the 3-D orthogonal $(m+n+o)$-th order of Jacobi moments are defined as

$$J_{m,n,o} = \frac{1}{\rho_m^{(\alpha,\beta)}\,\rho_n^{(\alpha,\beta)}\rho_o^{(\alpha,\beta)}} \int_{-1}^{+1}\int_{-1}^{+1}\int_{-1}^{+1} f(x,y,z)$$
$$P_m^{(\alpha,\beta)}(x)\,P_n^{(\alpha,\beta)}(y)P_o^{(\alpha,\beta)}(z)w^{(\alpha,\beta)}(x)w^{(\alpha,\beta)}(y)w^{(\alpha,\beta)}(z)dxdydz. \quad (24)$$

Similarly, a 3-D image function $f(x,y,z)$ can be reconstructed from an infinite series of its Jacobi moments

$$f(x,y,z) = \sum_{m=0}^{\infty}\sum_{n=0}^{m}\sum_{o=0}^{n} J_{m-n,n-o,o}\,P_{m-n}^{(\alpha,\beta)}(x)\,P_{n-o}^{(\alpha,\beta)}(y)\,P_o^{(\alpha,\beta)}(z). \quad (25)$$

When certain order $\leq M_{max}$ Jacobi moments are provided, the image function $f(x,y,z)$ can be approximated by a truncated series

$$f(x,y,z) \simeq f_{M_{max}}(x,y,z) = \sum_{m=0}^{M_{max}}\sum_{n=0}^{m}\sum_{o=0}^{n} J_{m-n,n-o,o}$$
$$P_{m-n}^{(\alpha,\beta)}(x)\,P_{n-o}^{(\alpha,\beta)}(y)\,P_o^{(\alpha,\beta)}(z). \quad (26)$$

Chapter 3

Improving Accuracy and Efficiency

For a 2-dimensional digital image sized $M \times N$, the image function $f(x, y)$ becomes its discrete version $f(x_i, y_i)$ defined in $[-1, 1]\times[-1, 1]$ region. Therefore, the double integration in (1) will transform to a formula approximated by double summation. The approximate moment $\widehat{\Psi}_{m,n}$ can be expressed as

$$\widehat{\Psi}_{m,n} = \sum_{x}\sum_{y} \psi_{m,n}(x_i, y_j) f(x_i, y_j)\Delta x \Delta y, \tag{27}$$

where Δx and Δy are the sampling intervals

$$\Delta x = x_i - x_{i-1}, \tag{28a}$$

$$\Delta y = y_j - y_{j-1}, \tag{28b}$$

with the constant values $\Delta x = \frac{2}{M}$ and $\Delta y = \frac{2}{N}$.

3.1 $k \times k$ sub-regions

Suppose the value of $\Delta x \Delta y$ is used directly to approximate the double integration over each image pixel in (27), in that case, significant computing errors of polynomial value in $[-1, 1] \times [-1, 1]$ region will be observed when the moment orders increase[5]. To improve the computational accuracy of

moments in rectangular region, in this research, a more accurate approximate formula is applied [12]

$$\widehat{\Psi}_{m,n} = \sum_{x} \sum_{y} f(x_i, y_j)\, h_{m,n}(x_i, y_j), \tag{29}$$

where

$$h_{m,n}(x_i, y_j) = \int_{x_i - \frac{\Delta x}{2}}^{x_i + \frac{\Delta x}{2}} \int_{y_j - \frac{\Delta y}{2}}^{y_j + \frac{\Delta y}{2}} \psi(x, y)\, dxdy. \tag{30}$$

Thus, the 2-dimensional Jacobi moments defined in (21) can be rewritten as

$$\widehat{J}_{m,n} = \frac{1}{\rho_m^{(\alpha,\beta)} \rho_n^{(\alpha,\beta)}} \sum_{i=1}^{M} \sum_{j=1}^{N} f(x_i, y_j) h_{m,n}(x_i, y_j), \tag{31}$$

where

$$h_{m,n}(x_i, y_j) = \int_{x_i - \frac{\Delta x}{2}}^{x_i + \frac{\Delta x}{2}} \int_{y_j - \frac{\Delta y}{2}}^{y_j + \frac{\Delta y}{2}} f(x, y) P_m^{(\alpha,\beta)}(x)\, P_n^{(\alpha,\beta)}(y) w^{(\alpha,\beta)}(x) w^{(\alpha,\beta)}(y) dxdy. \tag{32}$$

Although several numerical techniques can be applied to calculate the double integrations in (32), in this research, we have adopted the numerical scheme of dividing a pixel into $k \times k$ sub-regions with the same weights to improve the accuracy of moments computing [31].

Since the Jacobi polynomials $P_m^{(\alpha,\beta)}(x)$ and $P_n^{(\alpha,\beta)}(y)$ are independent, the

integrals in (32) can be replaced by

$$h_{m,n}(x_i, y_j) = \frac{4}{k^2MN} \sum_{r=1}^{k} P_m^{(\alpha,\beta)}(x_{i,r}) w^{(\alpha,\beta)}(x_{i,r}) \sum_{s=1}^{k} P_n^{(\alpha,\beta)}(y_{j,s}) w^{(\alpha,\beta)}(y_{j,s}), \tag{33}$$

where

$$x_i = -1 + (i - \frac{1}{2})\frac{\Delta x}{k}, \tag{34a}$$

$$y_j = 1 - (j - \frac{1}{2})\frac{\Delta y}{k}, \tag{34b}$$

$i = 1, 2, 3, ..., kM$ and $j = 1, 2, 3, ..., kN$ for an $M \times N$ image.

3.2 Matrix Multiplication

Although applying the $k \times k$ numerical scheme can improve moment computing accuracy, it will significantly increase the computational time in most situations [12].

To address the issue of computational inefficiency, an approach of applying matrix multiplications on computing moments defined in the rectangular region was proposed[20]. In our research, we have further improved this approach by implementing a parallel algorithm to apply the computational enhancement of GPU.

Substitute (33) into (31), we can express $\widehat{J}_{m,n}$ in the form of matrix multiplications

$$\widehat{J}_{m,n} = \frac{4}{k^2 MN} \frac{1}{\rho_m^{(\alpha,\beta)} \rho_n^{(\alpha,\beta)}} \mathbf{H}_m \mathbf{G} \mathbf{I}_n^T, \tag{35}$$

where $(m+n) \leq M_{max}$. For different m and n, $\widehat{J}_{m,n}$ can be represented in the matrix format

$$\widehat{J}_{m,n} = \begin{bmatrix} J_{0,0} & J_{0,1} & J_{0,2} & \dots & J_{0,M_{max}} \\ \vdots & \vdots & \vdots & \iddots & \vdots \\ J_{M_{max}-2,0} & J_{M_{max}-2,1} & J_{M_{max}-2,2} & \dots & 0 \\ J_{M_{max}-1,0} & J_{M_{max}-1,1} & 0 & \dots & 0 \\ J_{M_{max},0} & 0 & 0 & \dots & 0 \end{bmatrix}, \tag{36}$$

$$\mathbf{G} = \sum_{i=1}^{M} \sum_{j=1}^{N} f(x_i, y_j) = \begin{bmatrix} f_{1,1} & f_{1,2} & f_{1,3} & \dots & f_{1,N} \\ f_{2,1} & f_{2,2} & f_{2,3} & \dots & f_{2,N} \\ f_{3,1} & f_{3,2} & f_{3,3} & \dots & f_{3,N} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ f_{M,1} & f_{M,2} & f_{M,3} & \dots & f_{M,N} \end{bmatrix}, \tag{37}$$

$$\mathbf{H}_m = \sum_{r=1}^{k} P_m^{(\alpha,\beta)}(x_{i,r}) w^{(\alpha,\beta)}(x_{i,r})$$

$$= \begin{bmatrix} P_0^{(\alpha,\beta)}(x_{1,r}) w^{(\alpha,\beta)}(x_{1,r}) & P_0^{(\alpha,\beta)}(x_{2,r}) w^{(\alpha,\beta)}(x_{2,r}) & \dots & P_0^{(\alpha,\beta)}(x_{M,r}) w^{(\alpha,\beta)}(x_{M,r}) \\ P_1^{(\alpha,\beta)}(x_{1,r}) w^{(\alpha,\beta)}(x_{1,r}) & P_1^{(\alpha,\beta)}(x_{2,r}) w^{(\alpha,\beta)}(x_{2,r}) & \dots & P_1^{(\alpha,\beta)}(x_{M,r}) w^{(\alpha,\beta)}(x_{M,r}) \\ P_2^{(\alpha,\beta)}(x_{1,r}) w^{(\alpha,\beta)}(x_{1,r}) & P_2^{(\alpha,\beta)}(x_{2,r}) w^{(\alpha,\beta)}(x_{2,r}) & \dots & P_2^{(\alpha,\beta)}(x_{M,r}) w^{(\alpha,\beta)}(x_{M,r}) \\ \vdots & \vdots & \ddots & \vdots \\ P_{M_{max}}^{(\alpha,\beta)}(x_{1,r}) w^{(\alpha,\beta)}(x_{1,r}) & P_{M_{max}}^{(\alpha,\beta)}(x_{2,r}) w^{(\alpha,\beta)}(x_{2,r}) & \dots & P_{M_{max}}^{(\alpha,\beta)}(x_{M,r}) w^{(\alpha,\beta)}(x_{M,r}) \end{bmatrix}, \tag{38}$$

$$\mathbf{I}_n = \sum_{s=1}^{k} P_n^{(\alpha,\beta)}(y_j, r) w^{(\alpha,\beta)}(y_j, r)$$

$$= \begin{bmatrix} P_0^{(\alpha,\beta)}(y_{1,r}) w^{(\alpha,\beta)}(y_{1,r}) & P_0^{(\alpha,\beta)}(y_{2,r}) w^{(\alpha,\beta)}(y_{2,r}) & \dots & P_0^{(\alpha,\beta)}(y_{N,r}) w^{(\alpha,\beta)}(y_{N,r}) \\ P_1^{(\alpha,\beta)}(y_{1,r}) w^{(\alpha,\beta)}(y_{1,r}) & P_1^{(\alpha,\beta)}(y_{2,r}) w^{(\alpha,\beta)}(y_{2,r}) & \dots & P_1^{(\alpha,\beta)}(y_{N,r}) w^{(\alpha,\beta)}(y_{N,r}) \\ P_2^{(\alpha,\beta)}(y_{1,r}) w^{(\alpha,\beta)}(y_{1,r}) & P_2^{(\alpha,\beta)}(y_{2,r}) w^{(\alpha,\beta)}(y_{2,r}) & \dots & P_2^{(\alpha,\beta)}(y_{N,r}) w^{(\alpha,\beta)}(y_{N,r}) \\ \vdots & \vdots & \ddots & \vdots \\ P_{M_{max}}^{(\alpha,\beta)}(y_{1,r}) w^{(\alpha,\beta)}(y_{1,r}) & P_{M_{max}}^{(\alpha,\beta)}(y_{2,r}) w^{(\alpha,\beta)}(y_{2,r}) & \dots & P_{M_{max}}^{(\alpha,\beta)}(y_{N,r}) w^{(\alpha,\beta)}(y_{N,r}) \end{bmatrix}. \tag{39}$$

$\mathbf{G}$, $\mathbf{H}_m$, and $\mathbf{I}_n$ are $M \times N$, $(M_{max} + 1) \times M$, and $(M_{max} + 1) \times N$ matrix, respectively. T is the transpose of a matrix. The Jacobi polynomials $P_m^{(\alpha,\beta)}(x_{i,r})$ and $P_n^{(\alpha,\beta)}(y_{j,s})$ in (38) and (39) are the sum of $k \times 1$ vectors.

$$P_m^{(\alpha,\beta)}(x_{i,r})w^{(\alpha,\beta)}(x_{i,r}) = P_m^{(\alpha,\beta)}(x_i,1)w^{(\alpha,\beta)}(x_i,1) + P_m^{(\alpha,\beta)}(x_i,2)w^{(\alpha,\beta)}(x_i,2) + \ ... \ + P_m^{(\alpha,\beta)}(x_{i,k})w^{(\alpha,\beta)}(x_{i,k}) \quad (40)$$

and

$$P_n^{(\alpha,\beta)}(y_{j,s})w^{(\alpha,\beta)}(y_{j,s}) = P_n^{(\alpha,\beta)}(y_{j,1})w^{(\alpha,\beta)}(y_{j,1}) + P_n^{(\alpha,\beta)}(y_{j,2})w^{(\alpha,\beta)}(y_{j,2}) + \ ... \ + P_n^{(\alpha,\beta)}(y_{j,k})w^{(\alpha,\beta)}(y_{j,k}). \quad (41)$$

3.3 Matrix-Cuboid Multiplication

In a 3-D image domain, the 2-D Jacobi moments expressed in (36) can be regarded as the $x - y$ plane of a 3-D moment cuboid, and the 2-D moment matrix extends to the 3-D moment cuboid.

Referring to (35), $\widehat{J}_{m,n,o}$ can be expressed as

$$\widehat{J}_{m,n,o} = \frac{8}{k^3 MNO} \frac{1}{\rho_m^{(\alpha,\beta)} \rho_n^{(\alpha,\beta)} \rho_o^{(\alpha,\beta)}} \mathbf{Z}_o(\mathbf{H}_m \mathbf{G} \mathbf{I}_n^T), \quad (42)$$

where O is the length on z axis and $(m + n + o) \leq M_{max}$.

Figure 2 shows an example of 3-D matrix multiplications in (42), where $M = N = O = 4$, and $M_{max} = 3$.

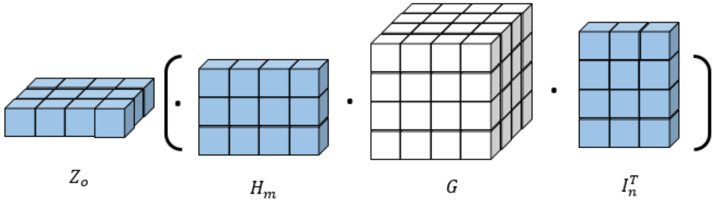

Figure 1: Example of 3-D matrix multiplication

Matrix multiplications in (35) have been extended to matrix-cuboid multiplications in (42). (Note: matrix-cuboid multiplication can be understood as matrix A multiply each matrix in the cuboid B on certain plane.)

Similarly, the polynomial matrix $\mathbf{Z}_o$ on z axis can be expressed as

$$\mathbf{Z}_o = \sum_{l=1}^{k} P_o^{(\alpha,\beta)}(z_q, l) w^{(\alpha,\beta)}(z_q, l)$$
$$= \begin{bmatrix} P_0^{(\alpha,\beta)}(z_{1,l}) w^{(\alpha,\beta)}(z_{1,l}) & P_0^{(\alpha,\beta)}(z_{2,l}) w^{(\alpha,\beta)}(z_{2,l}) & \dots & P_0^{(\alpha,\beta)}(z_{O,l}) w^{(\alpha,\beta)}(z_{O,l}) \\ P_1^{(\alpha,\beta)}(z_{1,l}) w^{(\alpha,\beta)}(z_{1,l}) & P_1^{(\alpha,\beta)}(z_{2,l}) w^{(\alpha,\beta)}(z_{2,l}) & \dots & P_1^{(\alpha,\beta)}(z_{O,l}) w^{(\alpha,\beta)}(z_{O,l}) \\ P_2^{(\alpha,\beta)}(z_{1,l}) w^{(\alpha,\beta)}(z_{1,l}) & P_2^{(\alpha,\beta)}(z_{2,l}) w^{(\alpha,\beta)}(z_{2,l}) & \dots & P_2^{(\alpha,\beta)}(z_{O,l}) w^{(\alpha,\beta)}(z_{O,l}) \\ \vdots & \vdots & \ddots & \vdots \\ P_{M_{max}}^{(\alpha,\beta)}(z_{1,l}) w^{(\alpha,\beta)}(z_{1,l}) & P_{M_{max}}^{(\alpha,\beta)}(z_{2,l}) w^{(\alpha,\beta)}(z_{2,l}) & \dots & P_{M_{max}}^{(\alpha,\beta)}(z_{O,l}) w^{(\alpha,\beta)}(z_{O,l}) \end{bmatrix}, \tag{43}$$

where each of the Jacobi polynomials in (43) is the sum of the $k \times 1$ vector

$$\begin{aligned} P_o^{(\alpha,\beta)}(z_{q,l}) w^{(\alpha,\beta)}(z_{q,l}) = \; & P_o^{(\alpha,\beta)}(z_q, 1) w^{(\alpha,\beta)}(z_q, 1) \\ & + P_o^{(\alpha,\beta)}(z_q, 2) w^{(\alpha,\beta)}(z_q, 2) + \dots + P_o^{(\alpha,\beta)}(z_{q,k}) w^{(\alpha,\beta)}(z_{q,k}). \end{aligned} \tag{44}$$

Figure 2 shows diagrams of 3 phases of matrix-cuboid multiplication in the process of moment calculation. Referring to (42), $\mathbf{H}_m \cdot \mathbf{G}$, $\mathbf{G} \cdot \mathbf{I}_n^T$, and $\mathbf{Z}_o \cdot \mathbf{G}$ are calculated in phases 1, 2, and 3, respectively. Phases 1 and 2 are conducted on the $x - y$ plane, while phase 3 is executed on the $x - z$ plane.

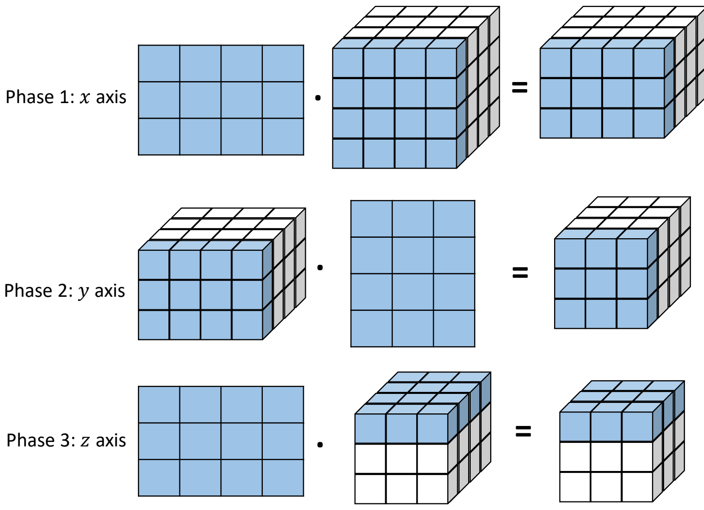

Figure 2: Diagram of 3 phases of matrix-cuboid multiplication in moments computing

After 3 phases of matrix-cuboid multiplication, each element of cuboid executes the rest part of (42) . Then, the element with $(m + n + o) > M_{max}$ are assigned to 0. Figure 3 shows a 3-D moment cuboid with $M_{max} = 7$.

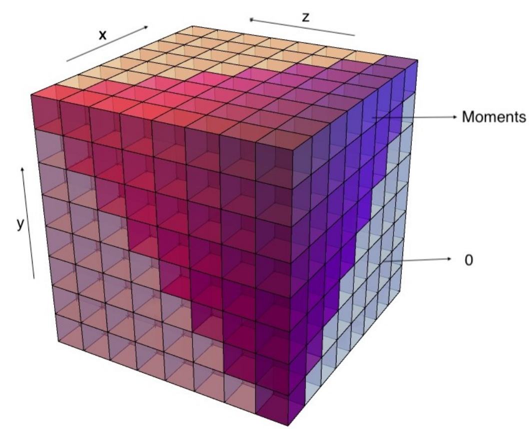

Figure 3: An example of 3-D moments cuboid

The planes of $y-z$ and $x-z$ of moments cuboid can be expressed as

$$\widehat{J}_{m,n,o} = \begin{bmatrix} J_{0,0,M_{max}} & \dots & J_{0,0,2} & J_{0,0,1} & J_{0,0,0} \\ \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & \dots & J_{0,M_{max}-2,2} & J_{0,M_{max}-2,1} & J_{0,M_{max}-2,0} \\ 0 & \dots & 0 & J_{0,M_{max}-1,1} & J_{0,M_{max}-1,0} \\ 0 & \dots & 0 & 0 & J_{0,M_{max},0} \end{bmatrix}, \quad (45)$$

and

$$\widehat{J}_{m,n,o} = \begin{bmatrix} J_{0,0,M_{max}} & 0 & 0 & \dots & 0 \\ J_{0,0,M_{max}-1} & J_{1,0,M_{max}-1} & 0 & \dots & 0 \\ J_{0,0,M_{max}-2} & J_{2,0,M_{max}-2} & J_{2,0,M_{max}-2} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ J_{0,0,0} & J_{1,0,0} & J_{2,0,0} & \dots & J_{M_{max},0,0} \end{bmatrix}. \quad (46)$$

Figure 4 demonstrates diagrams of 3 phases of matrix-cuboid multiplication in the process of image reconstruction.

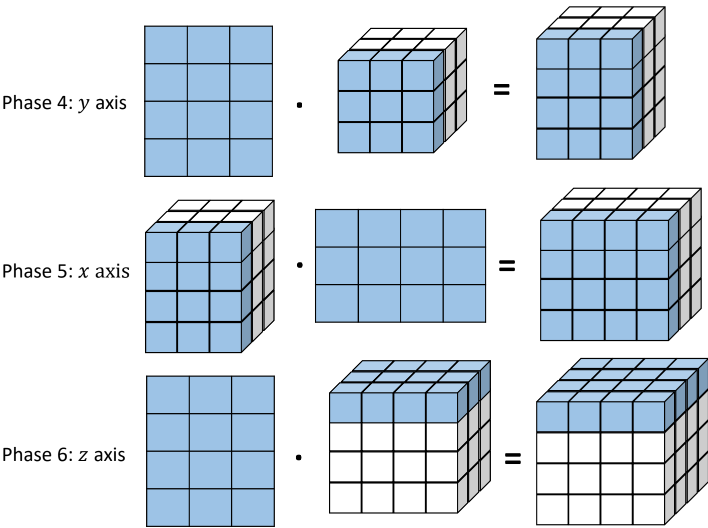

Figure 4: Diagram of 3 phases of matrix-cuboid multiplication in image reconstruction

Chapter 4

General Purpose of CUDA Computing

In this research, our parallel algorithms are implemented using CUDA(Compute Unified Device Architecture) which was introduced by NVIDIA. This chapter will introduce the basic CUDA features that are important in our program, including coalesced memory access, memory allocation, barrier synchronization and memory limitation.

4.1 Memory Overview

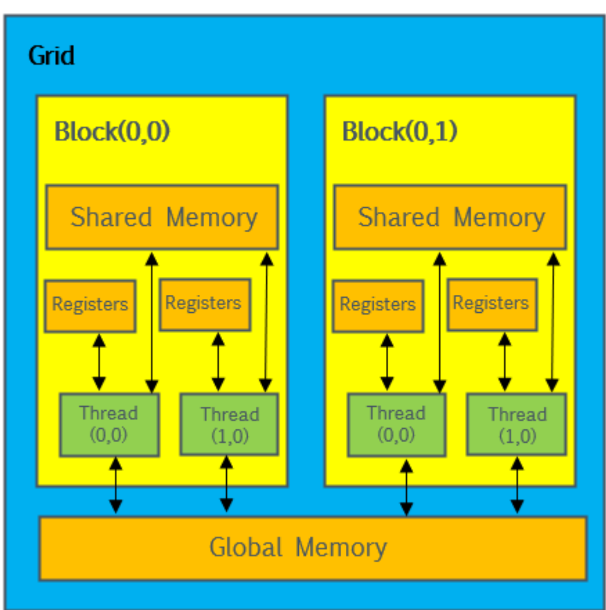

Figure 5: Overview of GPU memory[11]

In modern CUDA-GPU, a block assigned to a Streaming Multiprocessor is divided into 32 threads units called warps. To sufficiently consume each thread of warps, we assign $Tile_width$ to 8 for tiled matrix-cuboid multiplication, so 512 threads per block are allocated.

Figure 5 demonstrates an overview of GPU memory. Table 1 depicts the features of different memory regarding speed, capacity, and scope. Compared with off-chip global memory, shared memory has more bandwidth and is accessible among threads of the same block but holds less storage. Additionally, accessing the register is fast but only accessible to threads, so register is a good technique to fetch data from shared memory.

Table 1: Summary of Features of different Memory.

Memory type	Speed	Capacity	Scope
Global memory	Slow	Large	Grid
Shared memory	Fast	Small	Block
Register	Fast	Small	Thread

4.2 Memory limitation

Block size limitation regarding on-chip memory (e.g., shared memory and registers) is a critical issue determining the feasibility of the CUDA program. When block size is assigned to 8, there will be 64 and 512 threads allocated to each block for 2D and 3D respectively. For single-precision floating point operations, 2D array and 3D array will allocate 256 bytes and 2304 bytes separately in each block. Besides, maximum number of threads per block is 1024 on modern GPU.

GPU	Tesla K80	Tesla V100
Archeticture	Kepler	Volta
CUDA cores	2496	5120
Global Memory(GB)	12	16
Shared Memory per Block(bytes)	49152	49152

Figure 6: General Comparison between two classic GPU

4.3 Coalesced Memory Access

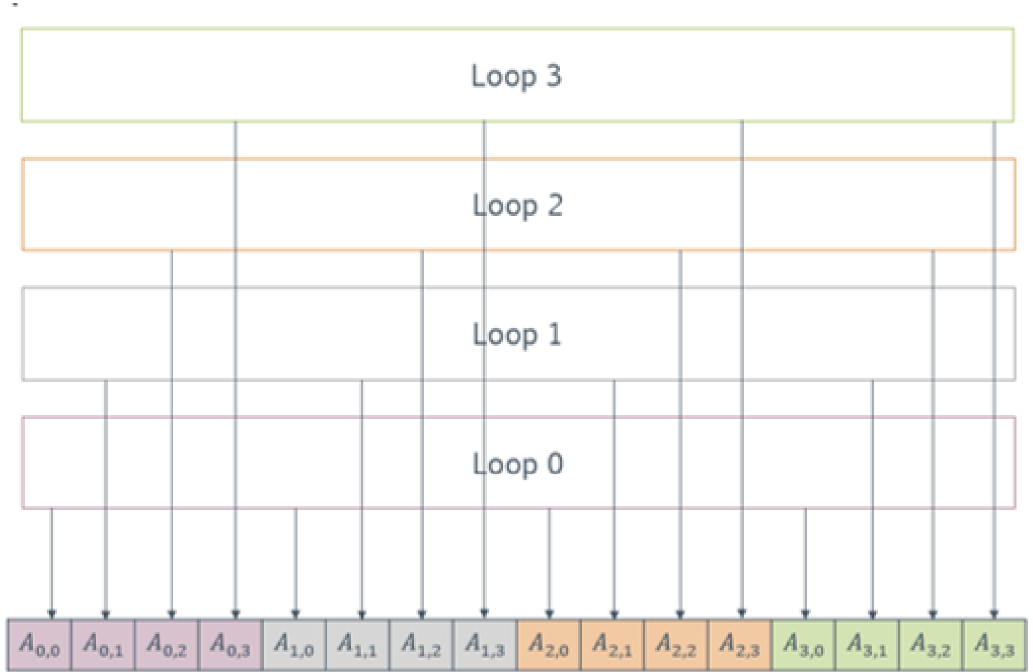

Figure 7: Uncoalesced memory access pattern[11]

Figure 7 and Figure 8 shows the uncoalesced and coalesced memory access respectively. For example, in Figure 8, array A is accessed in the manner as:

$A[ph * width + threadIdx.x]$

Since adjacent threads have consecutive $threadIdx.x$ values, the approach

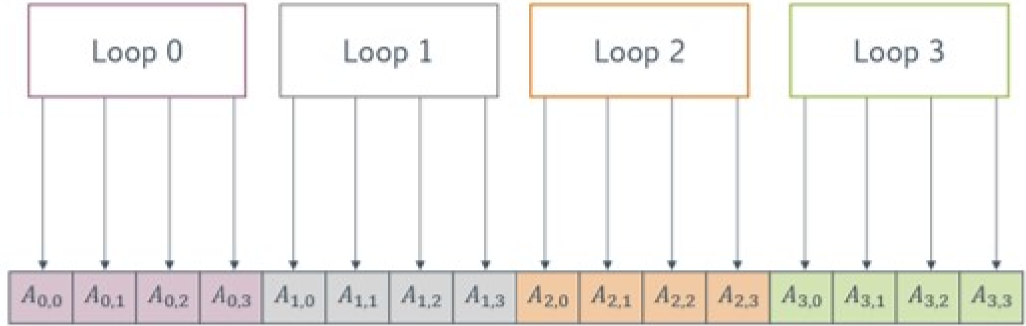

Figure 8: Coalesced memory access pattern[11]

described in Figure 8 will make threads accessible with consecutive addresses(a.k.a., coalesced memory access).

4.4 Barrier Synchronization

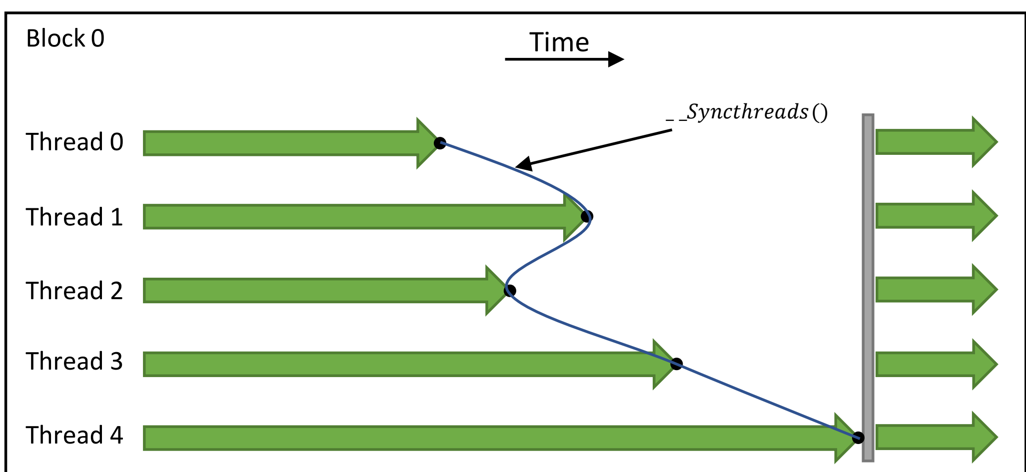

Figure 9: Diagram of Barrier Syncronization

As shown in Figure 9, each thread will not reach the next instruction and start execution at same time within one block. To deal with this issue, we call barrier synchronization statement(i.e., $__syncthreads()$) to synchronize execution, which means threads that arrive at this statement early will wait for threads that reach it late. When all threads finish the instruction and reach the statement, they are ready to concurrently go to the next instruction simultaneously.

Chapter 5

Implementation of GPU-Accelerated Algorithms

In this research, we have developed the GPU-accelerated algorithms for 3-D Jacobi moments computing and image reconstructions. Primarily, three parallel algorithms are developed on the GPU platform, including polynomial computation, matrix transpose, and matrix-cuboid multiplication. In addition, due to its nature of sequential computation, we have also implemented a CPU algorithm to apply (17a) and (17b).

5.1 Parallel Polynomial Computation

Algorithm 1 gives the pseudocode of parallel polynomial computation according to the recurrent Jacobi Polynomial in (13) with k schemes. On-chip memory is utilized in this section. Since on-chip memory will be typically applied in matrix-cuboid multiplication, we will discuss more details in Section 5.3.

CUDA Intrinsic Math Function is a library only available in device code. Compared with normal math function, it features with fast speed but less accuracy[1]. Through our experiments, the PSNR value is not affected when $pow(x, y)$ is changing to $__powf(x, y)$.

Polynomial values utilized in image reconstruction will conduct a similar algorithm, but there is no need to apply the k schemes.

Algorithm 1 Parallel computing of polynomial

1: Define $0th$ and $1st$ order polynomial $0th_x_k$ and $1st_x_k$ with k-scheme
2: Allocate $k \times 32$ to arrays JP_0th_sm, JP_1st_sm and weight w in shared memory.
3: $JP_0th_sm \leftarrow 0th_x_k$
4: $JP_1st_sm \leftarrow 1st_x_k$
5: $w^{(\alpha,\beta)}(x) \leftarrow (1-x)^{\alpha}(1+x)^{\beta}$
6: **for** $h \leftarrow 1$ to k **do**
7: $\quad row_0 \leftarrow row_0 + JP_0th_sm * w^{(\alpha,\beta)}(x)$
8: $\quad row_1 \leftarrow row_1 + JP_1st_sm * w^{(\alpha,\beta)}(x)$
9: **end for**
10: $X_JP[0th_order] \leftarrow row_0$
11: $X_JP[1st_order] \leftarrow row_1$
12: $Previous2nd_poly \leftarrow 0th_x_k$
13: $Previous1st_poly \leftarrow 1st_x_k$
14: **for** $n \leftarrow 2$ to M_{max} **do**
15: $\quad current_poly \leftarrow$ substitute $Previous2nd_poly$ and $Previous1st_poly$
16: $\quad$ Define JP_curr_sm on shared memory
17: $\quad JP_curr_sm \leftarrow current_poly$
18: $\quad row_n \leftarrow 0$
19: $\quad$ **for** $h \leftarrow 1$ to k **do**
20: $\quad\quad row_n \leftarrow row_n + JP_curr_sm * w^{(\alpha,\beta)}(x)$
21: $\quad$ **end for**
22: $\quad X_JP[nth_order] \leftarrow row_n$
23: $\quad Previous2nd_poly \leftarrow Previous1st_poly$
24: $\quad Previous1st_poly \leftarrow JP_curr_sm$
25: **end for**

5.2 Parallel Matrix Transpose

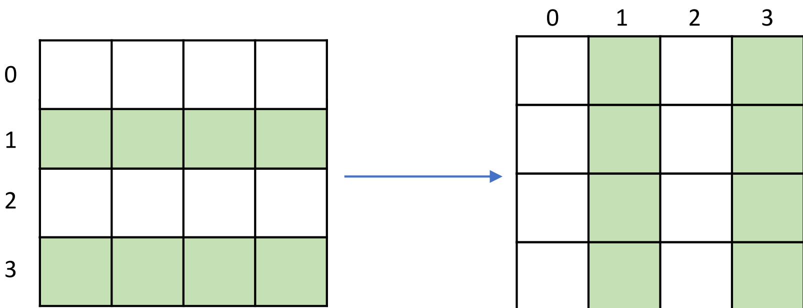

Figure 10: Diagram of parallel matrix transpose

Figure 10 shows the diagram of parallel matrix transpose. Referring to (42), $\mathbf{H}_m$ yielded from Algorithm 1 is transposed. The odd columns of the transposed matrix times -1 to apply the symmetric property in (18) and yields to $\mathbf{I}_n^T$.

Similar to $\mathbf{H}_m$, $\mathbf{Z}_o$ is polynomials of the positive axis as well. Thus, $\mathbf{H}_m$ can substitute $\mathbf{Z}_o$ and index on the $x - z$ plane to execute corresponding matrix-cuboid multiplication.

5.3 Parallel Matrix-Cuboid Multiplication

As expressed in 42, we have extended the matrix multiplication to matrix-cuboid multiplication for 3-D Jacobi moments computation and image reconstruction. Since the matrix-cuboid multiplication algorithm holds the majority calculation amount of the whole program, we adopted the tiled matrix-cuboid multiplication method to exploit the on-chip memory to optimize the computational efficiency.

Algorithm 2 depicts the pseudocode of the tiled matrix-cuboid multiplication, and Figure 11 shows the diagram of multiplication between matrix A and cuboid B with 2 tiling width.

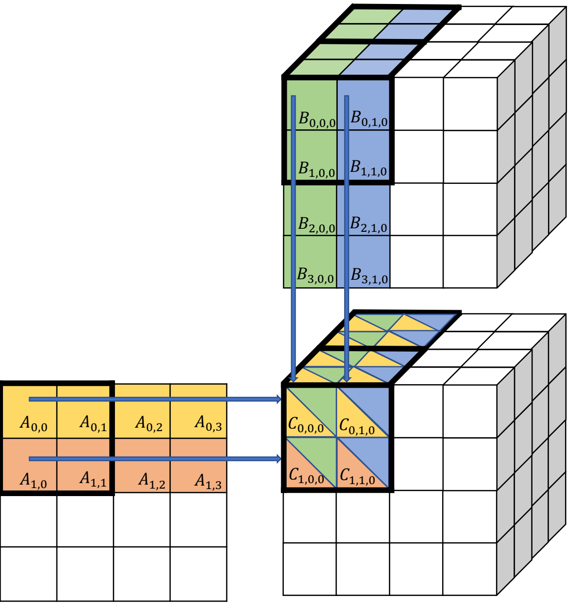

Figure 11: Example diagram of multiplication between matrix A and cuboid B with 2 tiling width

Algorithm 2 Tiled Matrix-Cuboid Multiplication

1: Define A_sm and B_sm in shared memory with $Tile_width$
2: $temp \leftarrow 0$
3: **for** $ph \leftarrow 0$ to $(width/Tile_width)$ **do**
4: Loading A into A_sm
5: Loading B into B_sm
6: $__syncthreads()$
7: **for** $i \leftarrow 0$ to $Tile_width$ **do**
8: $temp \leftarrow$ temp $+A_sm[threadIdx.y][i] \times B_sm[threadIdx.z][i][threadIdx.x]$
9: **end for**
10: $__syncthreads()$
11: **end for**
12: $Result_cuboid \leftarrow temp$

5.4 Implementation on CPU

Generally, GPU programming features the advantage of parallel computation but the disadvantage of high memory latency, which can speed up programs remarkably when the algorithm is highly parallel so that memory latency is tolerable. By contrast, the CPU has higher efficiency on sequential computation with low memory latency. Thus, modern GPU development often advocates being deployed on heterogeneous platforms. Algorithm 3 demonstrates the pseudocode of our CPU algorithm to compute (17a) and (17b). As shown in Figure 12, starting from $\rho_0^{(\alpha,\beta)}$, each $\rho_n^{(\alpha,\beta)}$ is calculated via $\rho_{n-1}^{(\alpha,\beta)}$ and assigned to 1-D array ρ_array iteratively. Therefore, such sequential computation optimally takes advantage of the mechanism of CPU.

Algorithm 3 Computation of coefficient $\rho_{M_{max}}^{(\alpha,\beta)}$

1: $\rho_0^{(\alpha,\beta)} \leftarrow$ (17b)
2: **for** $n \leftarrow 1$ to M_{max} **do**
3: $\quad \rho_temp \leftarrow$ substitute $\rho_{n-1}^{(\alpha,\beta)}$ into 17a
4: $\quad \rho_{n-1}^{(\alpha,\beta)} \leftarrow \rho_temp$
5: $\quad \rho_n^{(\alpha,\beta)} \leftarrow \rho_temp$
6: $\quad$ Assign $\rho_n^{(\alpha,\beta)}$ to ρ_array
7: **end for**
8: **return** ρ_array

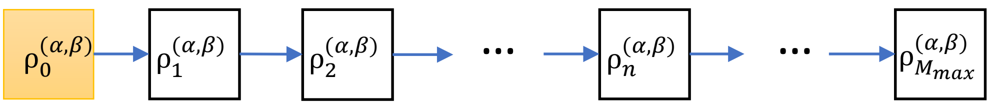

Figure 12: Diagram of computing array $\rho_m^{(\alpha,\beta)}$

5.5 Performance of GPU-Accelerated Algorithm on 2-D Jacobi Moment Computing

To verify our newly proposed GPU-accelerated parallel algorithm, we have performed the image reconstructions via the 2-dimensional Jacobi moments applying our GPU methodology for a general comparison with those of the CPU-main algorithm proposed in [20], where the computational time is 4.8221 seconds when $M_{max} = 1000, k = 23$ and $\alpha = \beta = 0.3$. The following system is employed to perform this experiment.

- System I: an Amazon Web Service (AWS) instance equipped with NVIDIA K80 GPU, 61 GB RAM and 12-core Intel Xeon E5-2686 2.93

GHz;

Table 2 shows the computational time, in milliseconds, of image reconstructions on a testing image sized at $1,024 \times 1,024$ from Jacobi moments applying our new GPU-accelerated parallel algorithm. Compared with the experimental results reported in [20], the GPU-based algorithm has substantially improved the computational time over the CPU algorithm.

Table 2: The computational time(ms) of $1,024 \times 1,024$ image reconstructions via the 1000-th order of Jacobi moments with $\alpha = 0.3$ and $\beta = 0.3$ between our GPU-based algorithm.

k/M_{max}	100	200	400	600	800	1000
1×1	5.4285	11.9645	29.2900	53.3869	87.1518	125.0830
3×3	5.5239	11.9465	29.6824	53.7598	87.7610	125.0563
7×7	5.6209	12.2677	30.0813	54.3206	88.5738	126.5641
11×11	5.8006	12.3777	30.7504	54.7838	89.4475	126.4578
15×15	5.9706	12.6044	30.8305	55.9036	90.5527	128.4940
19×19	6.1716	12.9281	31.6762	56.5531	91.2469	117.4786
23×23	6.3164	13.2126	32.0236	57.6629	92.6617	131.7593

Chapter 6

Experimental Results and Analysis

In this research, for a more comprehensive study, we have conducted our experimental tests on System II.

- System II: an Amazon Web Service (AWS) instance equipped with NVIDIA Tesla V100 GPU, 61 GB RAM and 8-core Intel Xeon X5670 2.30GHz.

To assess the accuracy and efficiency of our GPU-based parallel method regarding 3-D Jacobi moments, we have performed the image reconstructions with $k \times k \times k$ schemes from Jacobi moments up to orders of 500. An image sized at $512 \times 512 \times 512$ with 256 gray levels is utilized as the testing image, which is shown in Figure 13.

To evaluate the quality of a reconstructed image, we have adopted Peak Signal-to-Noise Ratio (PSNR) as the measurement. The PSNR is the ratio between the maximum power of the signal and the affecting noise, and is defined as

$$PSNR = 10log_{10}\frac{G_{max}^2}{MSE}, \tag{47}$$

where G_{max}^2 is the maximum gray level of an image function, and MSE is the Mean Square Error between the original $M \times N \times O$ image function

Figure 13: The tesing image of knee with size $512 \times 512 \times 512$ and 256 gray levels [24].

$f(x_i, y_j, z_t)$ and its reconstructed version $\widehat{f}(x_i, y_j, z_t)$

$$MSE = \frac{1}{MNO} \sum_{i=1}^{M} \sum_{j=1}^{N} \sum_{t=1}^{O} [\, f(x_i, y_j, z_t) - \widehat{f}(x_i, y_j, z_t)\,]^2. \tag{48}$$

In general, the higher PSNR values indicate that the better image reconstruction performances have been conducted.

6.1 3-D Image Reconstruction

In this section, a series of 3-D knee images reconstructed from Jacobi moments with $\alpha = 0.3$ and $\beta = 0.3$ are shown in Figure 14. Since the symmetric property can not apply to the algorithm of Jacobi moments computing and image reconstruction via Jacobi moments when $\alpha \neq \beta$, we demonstrate a set

of 3-D knee images reconstructed from Jacobi moments with $\alpha = 0.3$ and $\beta = 0.7$ in Figure 15. Both experiments are deployed on System II.

To inspect the efficiency of our GPU-based algorithm, Table 3 and Table 5 demonstrates the computational times of computing the Jacobi moments of Figure 13 and performing the image reconstructions from different orders of Jacobi moments with various $k \times k \times k$ numerical schemes. As shown in Table 3 and Table 5, the total computational times to compute the 500-th order of Jacobi moments with coefficients $\alpha = 0.3$, $\beta = 0.3$ and $\alpha = 0.3$, $\beta = 0.7$, when $k = 23$, and perform the 3-D image reconstruction on an image sized at $512 \times 512 \times 512$ are 387.36 ms and 382.22, respectively.

To measure the accuracy of our GPU-based algorithm, Table 4 and Table 6 shows the PSNR values of the reconstructed Figure 13 from different orders of Jacobi moments of $\alpha = 0.3$, $\beta = 0.3$ and $\alpha = 0.3$, $\beta = 0.7$, with diverse $k \times k \times k$ numerical schemes. When the order of Jacobi moments is 500, and the $23 \times 23 \times 23$ scheme is applied, the PSNR values of the reconstructed images are 53.6382 and 52.2096 for both experiments.

In Figure 14 and Figure 15, we adopt M_{max} orders from 100 to 500 and $k \times k \times k$ numerical schemes of $k = 3, 15$, and 23. When M_{max} and k are rising, the errors of reconstructed images are visually decreasing.

6.1.1 $\alpha = 0.3$ and $\beta = 0.3$

Table 3: The computational time, in millionseconds, of computing Jacobi moments of Figure 13 with $\alpha = 0.3$ and $\beta = 0.3$ and performing the image reconstructions from different maximum orders and $k \times k \times k$ schemes in System II.

k/M_{max}	100	200	300	400	500
1	34.14	86.05	156.49	260.32	381.16
3	31.64	79.75	151.34	264.50	382.56
7	34.14	82.66	143.23	264.17	395.09
11	34.14	76.64	157.07	262.18	392.44
15	31.58	86.12	154.35	263.35	391.03
19	34.18	82.98	159.79	241.76	391.46
23	34.19	83.56	151.32	261.98	387.36

Table 4: The PSNR values of the reconstructed Figure 13 from applying different maximum orders of Jacobi moments computed by using $\alpha = 0.3$ and $\beta = 0.3$.

k/M_{max}	100	200	300	400	500
1	25.2091	28.0826	27.4947	24.6565	23.1343
3	25.3882	31.1975	35.8204	32.8885	32.9276
7	25.4009	31.3042	37.9227	45.6986	47.1008
11	25.4029	31.3189	38.0050	46.4260	51.8814
15	25.4035	31.3231	38.0389	46.6932	52.7366
19	25.4037	31.3250	38.0530	46.7840	53.4047
23	25.4038	31.3261	38.0599	46.8488	53.6382

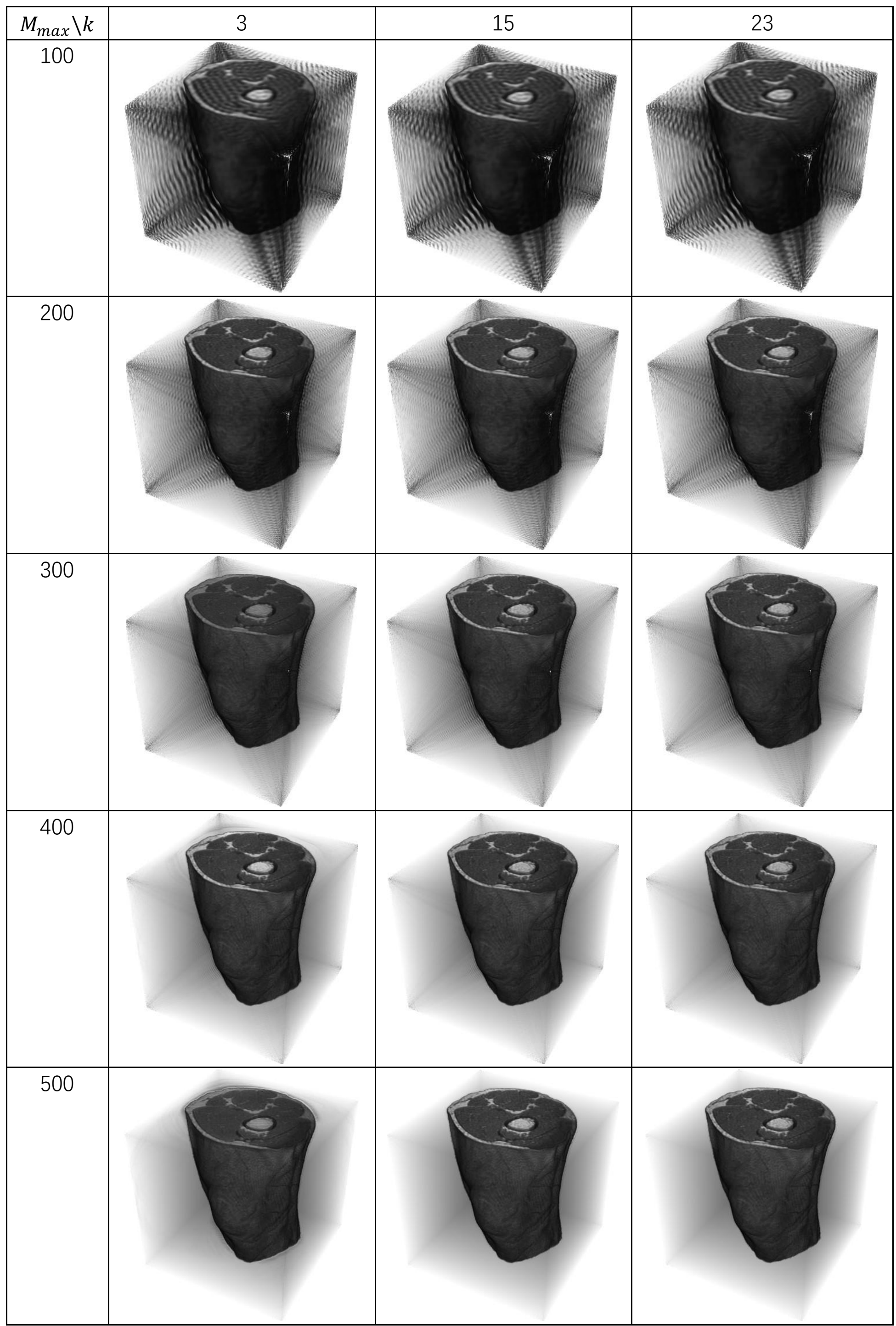

Figure 14: Some reconstructed images of Figure 13 from different maximum orders of Jacobi moments with various $k \times k \times k$ numerical schemes and orders at $\alpha = 0.3$ and $\beta = 0.3$.

6.1.2 $\alpha = 0.3$ and $\beta = 0.7$

Table 5: The computational time, in millionseconds, of computing Jacobi moments of Figure 13 with $\alpha = 0.3$ and $\beta = 0.7$ and performing the image reconstructions from different maximum orders and $k \times k \times k$ schemes in System II.

k/M_{max}	100	200	300	400	500
1	34.77	87.31	154.77	263.90	388.54
3	34.73	87.19	161.93	244.22	390.33
7	31.87	86.14	151.55	266.56	372.75
11	34.53	81.08	148.74	244.32	388.13
15	34.79	87.30	145.37	266.43	371.74
19	34.83	84.28	160.94	260.52	387.55
23	34.63	86.79	159.83	247.81	382.22

Table 6: The PSNR values of the reconstructed Figure 13 from applying different maximum orders of Jacobi moments computed by using $\alpha = 0.3$ and $\beta = 0.7$.

k/M_{max}	100	200	300	400	500
1	22.3991	26.3	27.1229	24.754	23.1449
3	22.5118	28.1887	33.2822	32.3055	33.193
7	22.5224	28.267	34.3804	41.9403	47.6815
11	22.5242	28.28	34.4314	42.2188	50.961
15	22.5248	28.2849	34.4512	42.3068	51.6275
19	22.5252	28.2874	34.4601	42.3463	52.0355
23	22.5254	28.2889	34.4651	42.3697	52.2096

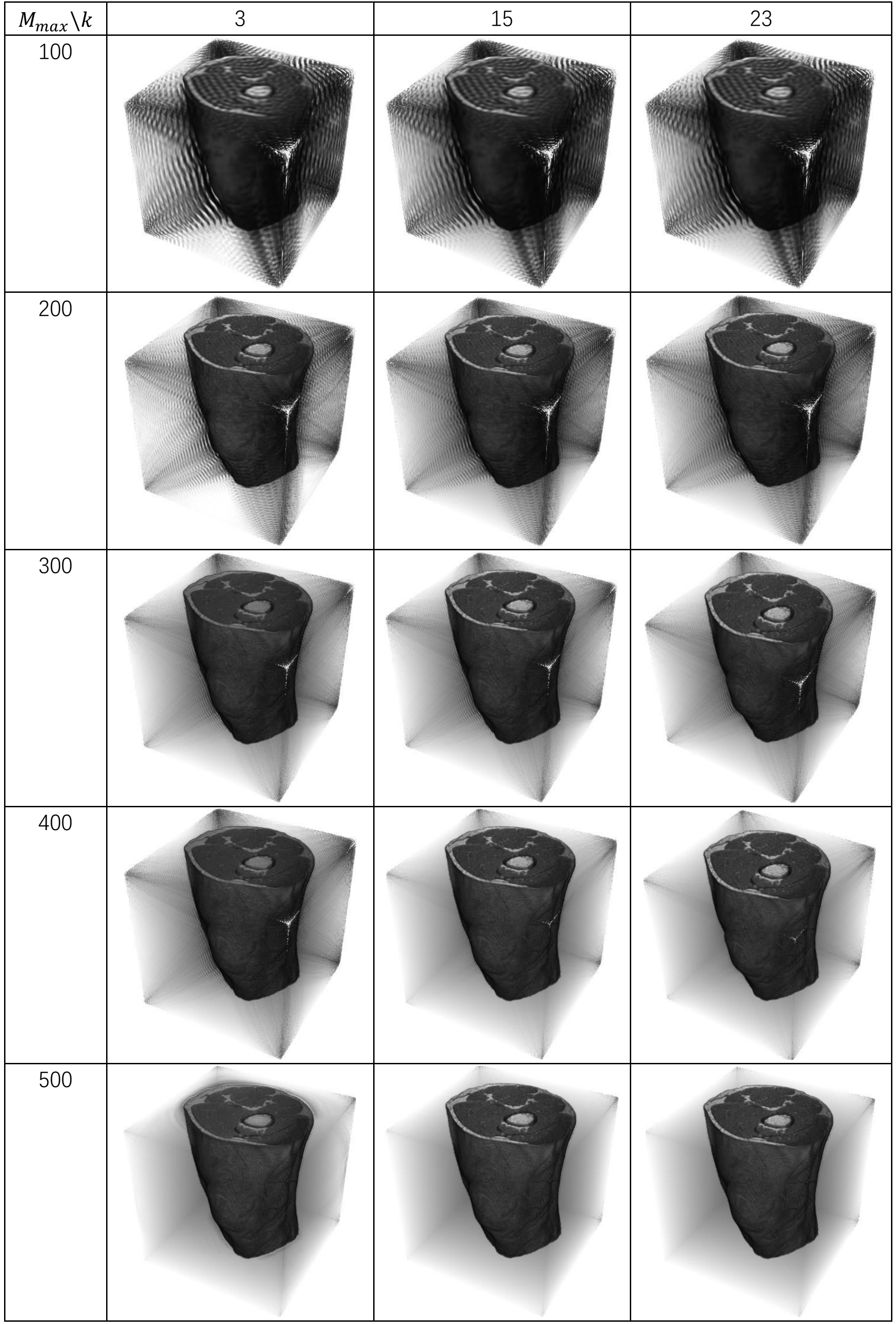

Figure 15: Some reconstructed images of Figure 13 from different maximum orders of Jacobi moments with various $k \times k \times k$ numerical schemes and orders at $\alpha = 0.3$ and $\beta = 0.7$.

6.2 Image Slicing and Clipping

Base on the highly satisfied results of 3-D image reconstructions from Jacobi moments, we can perform slicing and clipping to conduct image analysis.

Firstly, by employing the 3-D Jacobi moments of orders up to 500, $23 \times 23 \times 23$ numerical scheme, and coefficients sets of $\alpha = \beta = 0.3$ and $\alpha = 0.3, \beta = 0.7$, we have conducted various slicing positions on $x - y$, $y - z$ and $x - z$ planes. With 20 distance, slicing positions are ranged from 190 to 270 shown in Figure 16 and Figure 18.

Figure 17 and Figure 19 demonstrates some of the clipped reconstructed images from the 500-th order of 3-D Jacobi moments, $\alpha = \beta = 0.3$ and $\alpha = 0.3, \beta = 0.7$ respectively, with $k = 23$. The 3-D images are clipped by the planes shown above each of images. The clipping planes in each column are parallel to others with 100 distance on the x axis.

The computational times to reconstruct all of the sliced image shown in Figure 16 and Figure 18 are negligible, and our algorithm is able to provide the real-time performance with the highly satisfied accuracy.

All 3-D image visualization in our experiments are implemented on Mathematica 12. Although the background of original Figure 13 is black, for the convenience of 3-D image visualization, we have adjusted the background of 3-D images to white in Figure 13, Figure 14, Figure 15, Figure 17, and Figure 19.

6.2.1 $\alpha = 0.3$ and $\beta = 0.3$

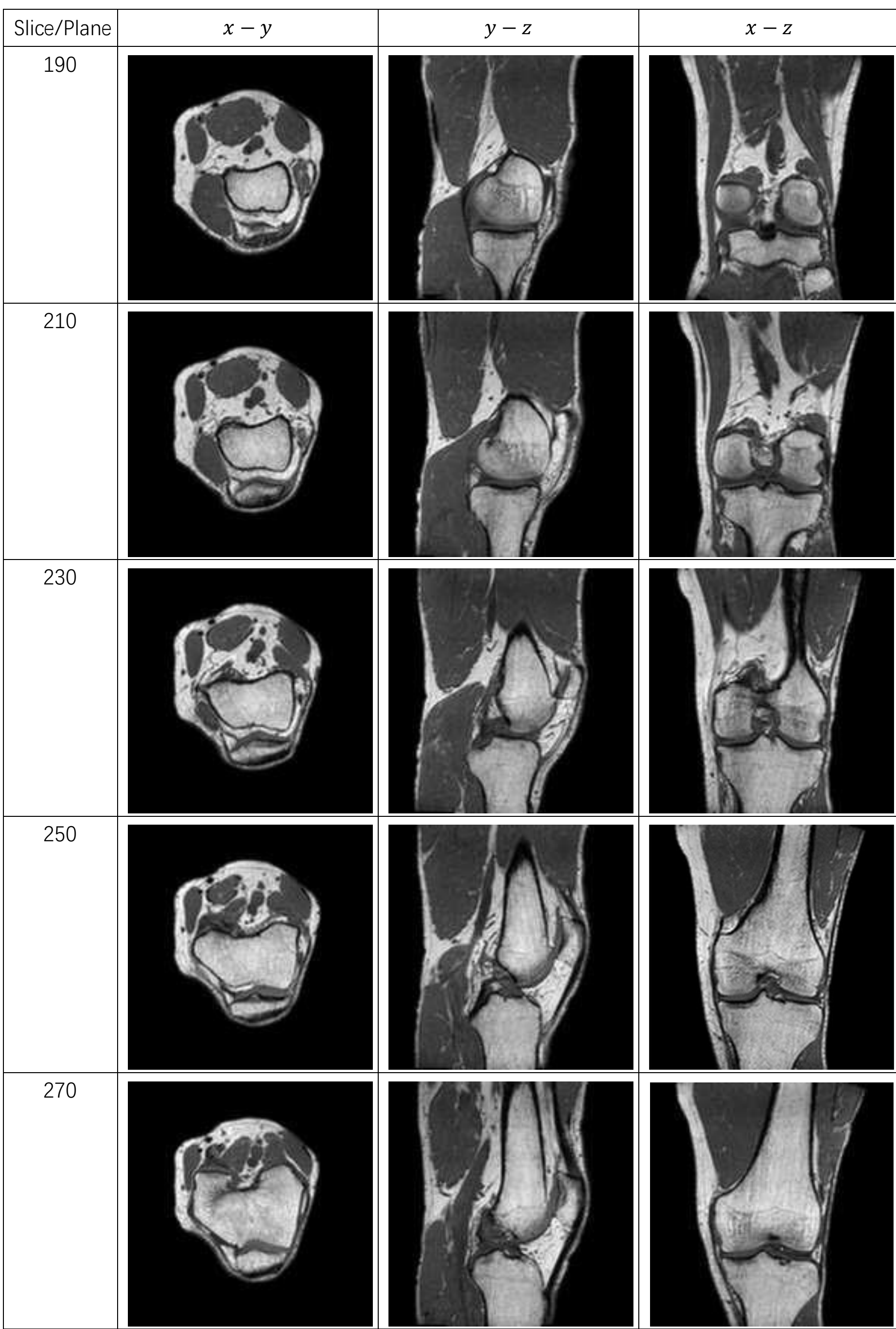

Figure 16: Sliced image from reconstructed 3-D image on $x - y$, $y - z$ and $x - z$ by $M_{max} = 500$ and $k = 23$ at $\alpha = 0.3$ and $\beta = 0.3$.

Figure 17: Clipped 3-D reconstructed image from Jacobi moments by $M_{max} = 500$ and $k = 23$ at $\alpha = 0.3$ and $\beta = 0.3$.

6.2.2 $\alpha = 0.3$ and $\beta = 0.7$

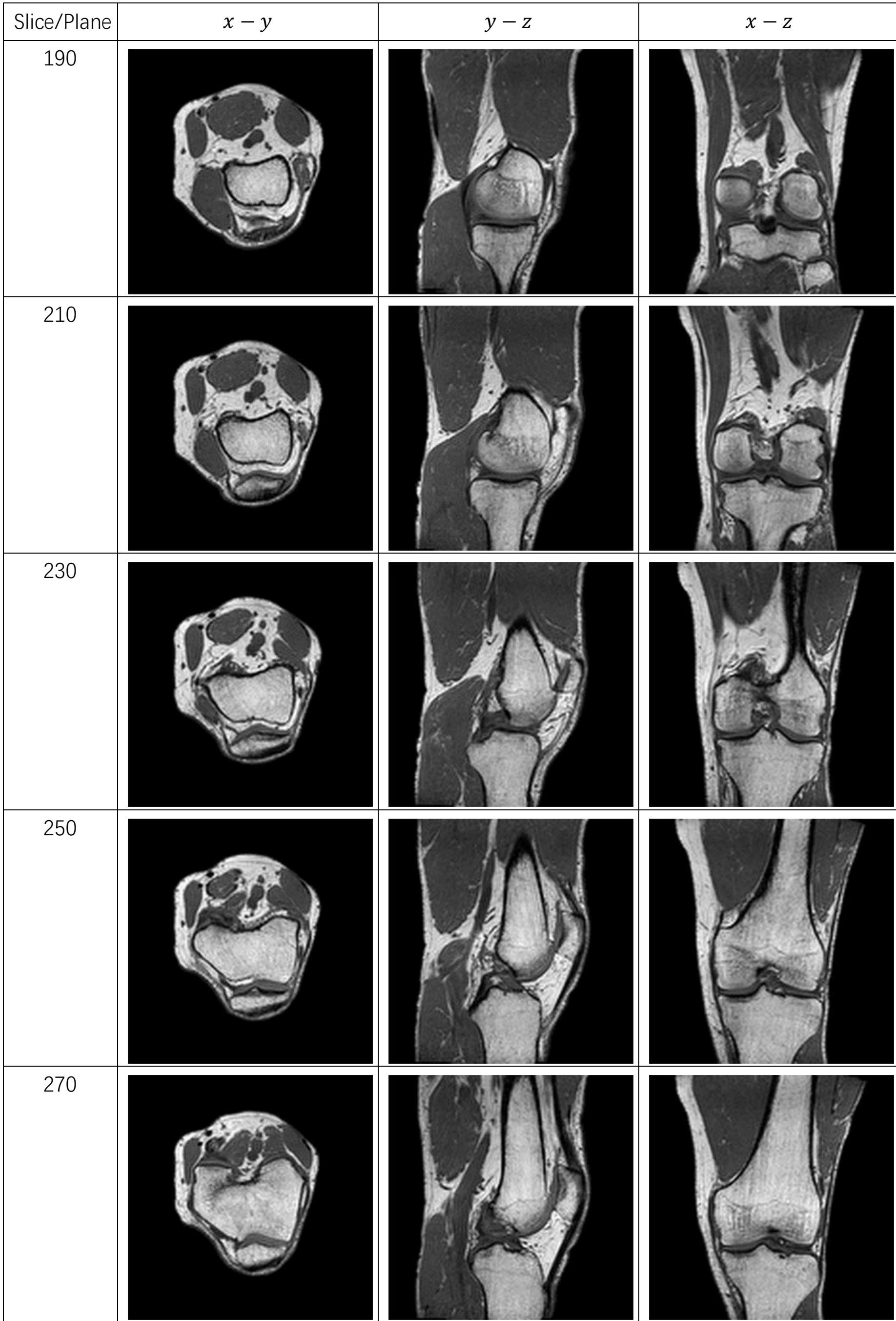

Figure 18: Sliced image from reconstructed 3-D image on $x - y$, $y - z$ and $x - z$ by $M_{max} = 500$ and $k = 23$ at $\alpha = 0.3$ and $\beta = 0.7$.

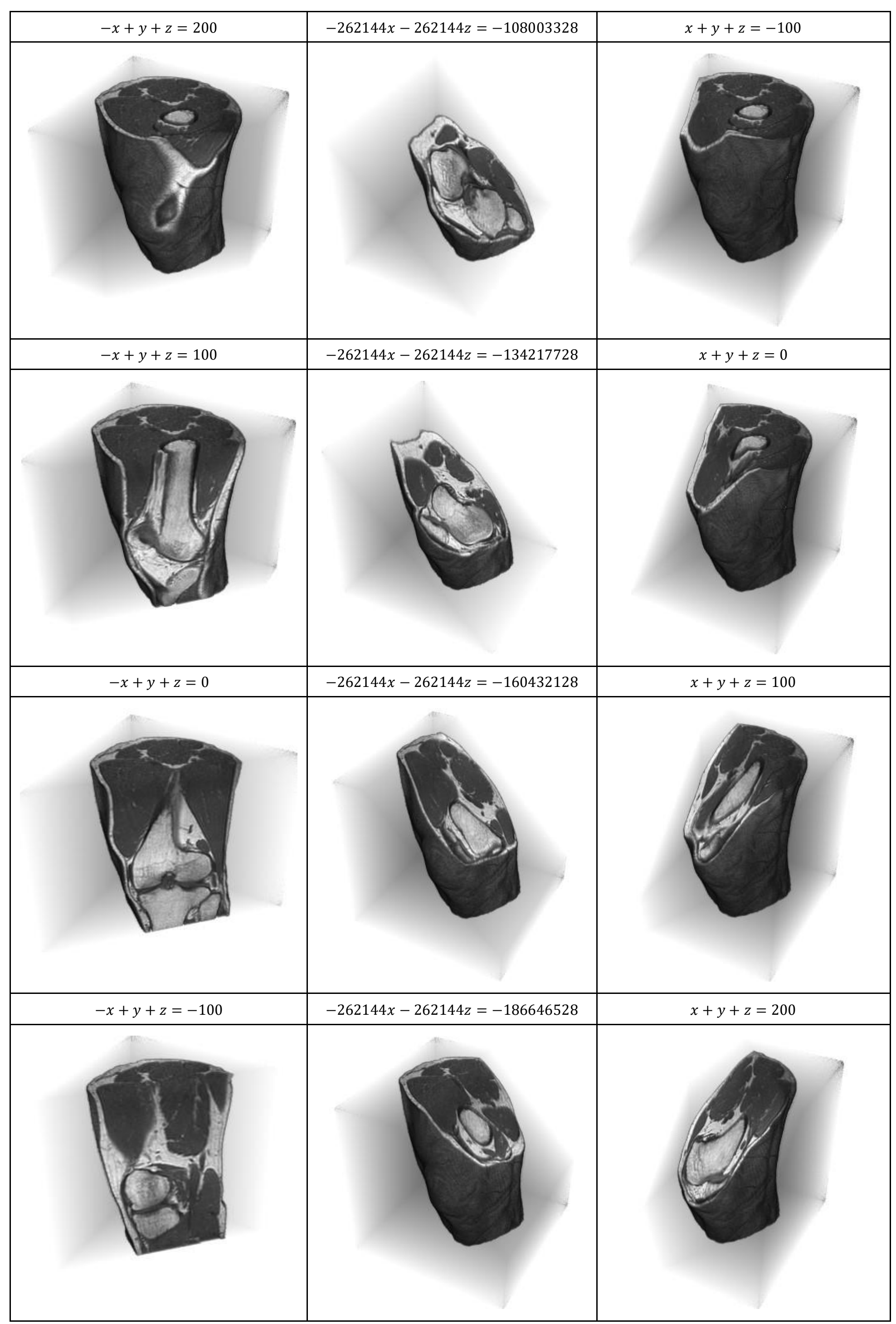

Figure 19: Clipped 3-D reconstructed image from Jacobi moments by $M_{max} = 500$ and $k = 23$ at $\alpha = 0.3$ and $\beta = 0.7$.

Chapter 7

Conclusion and Future Work

In conclusion, we have developed a parallel algorithm to compute 3-D Jacobi moments in a rectangular region. The proposed algorithm mathematically improves the efficiency by utilizing recurrent polynomial, symmetry properties, and k sub-regions. Also, the optimization related to GPU programming includes coalesced memory access, tiled matrix-cuboid multiplication, and heterogeneous computation.

Regarding the performance verification of our algorithm, we implement the image reconstruction from higher orders up to 500 and k scheme up to 23 on the test image sized at $512 \times 512 \times 512$, and the PSNR value between reconstructed images and the original image can reach around 53 at maximum. We also conducted a series of image clipping and slicing on the optimally reconstructed image. Besides, we have addressed the dilemma, which rising k will barely increase the computational time.

Since matrix multiplication is the most expensive algorithm in this research. CUDA library *cuBLAS* is considered to be adopted to optimize the matrix multiplication for future work.

References

[1] Cuda toolkit documentation, April 2021.

[2] Zaineb Bahaoui, Hakim El Fadili, Khalid Zenkouar, and Hassan Qjidaa. Image analysis by efficient gegenbauer moments computation for 3d objects reconstruction. In *Information Technology for Organizations Development (IT4OD), 2016 International Conference on*, pages 1–6. IEEE, 2016.

[3] William Wallace Bell. *Special functions for scientists and engineers.* Courier Corporation, 2004.

[4] A Chiang, S Liao, Q Lu, and M Pawlak. Gegenbauer moment-based applications for chinese character recognition. In *Electrical and Computer Engineering, 2002. IEEE CCECE 2002. Canadian Conference on*, volume 2, pages 908–911. IEEE, 2002.

[5] Amy Chiang and Simon Liao. Image analysis with legendre moment descriptors. *Journal of Computer Science*, 11(1):127–136, 2014.

[6] Jan Flusser, Barbara Zitova, and Tomas Suk. *Moments and moment invariants in pattern recognition.* John Wiley & Sons, 2009.

[7] Khalid Hosny. New set of gegenbauer moment invariants for pattern recognition applications. *Arabian Journal for Science & Engineering (Springer Science & Business Media BV)*, 39(10), 2014.

[8] Khalid M Hosny. Image representation using accurate orthogonal gegenbauer moments. *Pattern Recognition Letters*, 32(6):795–804, 2011.

[9] Khalid M. Hosny, Ahmad Salah, Hassan I. Saleh, and Mahmoud Sayed. Fast computation of 2d and 3d legendre moments using multi-core cpus and gpu parallel architectures. *Journal of Real-Time Image Processing*, 16(6):2027–2041, 2019.

[10] Ming-Kuei Hu. Visual pattern recognition by moment invariants. *information Theory, IRE Transactions on*, 8(2):179–187, 1962.

[11] D.B. Kirk and W.W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach.* Morgan Kaufmann, 3 edition, 2016.

[12] Simon Liao. Accuracy analysis of moment functions. *Gate to Computer Sciece and Research*, 2014.

[13] Simon Liao and Jing Chen. Object recognition with lower order gegenbauer moments. *Lecture Notes on Software Engineering*, 1(4):387, 2013.

[14] Simon Liao, Amy Chiang, Qin Lu, and Miroslaw Pawlak. Chinese character recognition via gegenbauer moments. In *Pattern Recognition, 2002. Proceedings. 16th International Conference on*, volume 3, pages 485–488. IEEE, 2002.

[15] Simon Liao and Miroslaw Pawlak. On image analysis by moments. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(3):254–266, 1996.

[16] Abramowitz M. and Stegun I. A. Handbook of mathematical functions with formulas, graphs, and mathematical tables. 1965.

[17] Wolfram Mathmatica. Segmentation of a Knee Bone in 3D . https://www.wolfram.com/mathematica/new-in-10/enhanced-3d-image-processing/segmentation-of-a-knee-bone-in-3d.html.

[18] R Mukundan and KR Ramakrishnan. Fast computation of legendre and zernike moments. *Pattern recognition*, 28(9):1433–1442, 1995.

[19] Ramakrishnan Mukundan and KR Ramakrishnan. *Moment functions in image analysis: theory and applications*, volume 100. World Scientific, 1998.

[20] Marcel Nwali and Simon Liao. A new fast algorithm to compute moments defined in a rectangular region. *Pattern Recognition*, 89, 05 2019.

[21] George A Papakostas. Moments and moment invariants: Theory and applications. *Science Gate*, 2014.

[22] Miroslaw Pawlak. Image analysis by moments: reconstruction and computational aspects. *Oficyna Wydawnicza Politechniki Wrocławskiej*, 2006.

[23] Koekoek R. and Swarttouw R. F. *The Askey-scheme of hypergeometric orthogonal polynomials and its q-analogue*. 1998.

[24] Wolfram Research. ExampleData. `https://reference.wolfram.com/language/ref/ExampleData.html`, 2016.

[25] G Sansone. *Orthogonal functions*, volume 9. Dover Publications, 1991.

[26] NVS Sree, Rathna Lakshmi, and C Manoharan. An automated system for classification of micro calcification in mammogram based on jacobi moments. *International Journal of Computer Theory and Engineering*, 3(3):431, 2011.

[27] Gabor Szego. *Orthogonal Polynomials*. Colloquium Publications, 1975.

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

[29] C-H Teh and Roland T. Chin. On image analysis by the methods of moments. *IEEE Transactions on pattern analysis and machine intelligence*, 10(4):496–513, 1988.

[30] Bo Pang Tianpeng Xia and Simon Liao. An efficient approach to computer bessel-fourier moments with gpu-accelerated algorithm. *Elsevier*, 2020.

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

[32] Pew-Thian Yap and Reveendran Paramesran. Jacobi moments as image feature. In *TENCON, 2004 IEEE Region 10 Conference*. IEEE, 2004.

Code Snippets

This chapter depicts five kernels, including polynomial computing with k scheme, polynomial computing without k scheme, symmetric property and matrix transpose, and computation of $\rho_n^{(\alpha,\beta)}$. Also, their corresponding function calls are listed in Snippet **??**.

Apart from the function call, memory should be defined and allocated on the host side before kernel launch. And necessary data need to transfer from host to device via $cudaMemcpy()$(i.e., the test image in this program). After kernel callings are completed, the computing result also needs to transfer back to the host(i.e., reconstructed image in this program). Finally, we need to free the memory that is allocated initially.

Snippet 1: Polynomial Computing with k scheme Kernels

```
1 __global__ void PositveIndex(float *kXArray, float *
    kXArray_1st, float *kXArray_0th, float *d_xJacobiP, const
    long d_k, const long order, float alpha, float beta)
2 {
3   int row = blockIdx.y * blockDim.y + threadIdx.y;
4   int col = blockIdx.x * blockDim.x + threadIdx.x;
5   float delta = 2. / IMG_SIZE;
```

```
6   if ((row < d_k) && (col < IMG_SIZE))
7   {
8     kXArray[row * IMG_SIZE + col] = -1.f + (col * d_k + row +
      1.f) * delta / (float)d_k;
9     kXArray_1st[row * IMG_SIZE + col] = 0.5 * (alpha - beta +
      (alpha + beta + 2) * kXArray[row * IMG_SIZE + col]);
10    kXArray_0th[row * IMG_SIZE + col] = 1.f;
11  }
12 }
13
14 __global__ void PolynomialMatrix2(float *kXArray, float *
     kXArray_1st, float *kXArray_0th, float *d_xJacobiP, const
     long d_k, const long order, float alpha, float beta)
15 {
16   int row = blockIdx.y * blockDim.y + threadIdx.y;
17   int col = blockIdx.x * blockDim.x + threadIdx.x;
18   float kXArray_1 = 1 - kXArray[row * IMG_SIZE + col];
19   float kXArrayp_1 = 1 + kXArray[row * IMG_SIZE + col];
20   __shared__ float JP_0th_sm[k][BLOCK_SIZE];
21   __shared__ float JP_1st_sm[k][BLOCK_SIZE];
22   __shared__ float weight_s[k][BLOCK_SIZE];
23   if ((row < d_k) && (col < IMG_SIZE))
24   {
25     JP_0th_sm[threadIdx.y][threadIdx.x] = kXArray_0th[row *
     IMG_SIZE + col];
26     JP_1st_sm[threadIdx.y][threadIdx.x] = kXArray_1st[row *
     IMG_SIZE + col];
```

```
27
28     weight_s[threadIdx.y][threadIdx.x] = alpha == 0 && beta
    == 0 ? 1 : __powf(kXArray_1, alpha) * __powf(kXArrayp_1,
    beta);
29
30     if (row == 0)
31     {
32       float row_0 = 0.f;
33       float row_1 = 0.f;
34       for (int h = 0; h < d_k; ++h)
35       {
36         row_0 += JP_0th_sm[h][threadIdx.x] * weight_s[h][
    threadIdx.x];
37         row_1 += JP_1st_sm[h][threadIdx.x] * weight_s[h][
    threadIdx.x];
38         __syncthreads();
39       }
40
41       d_xJacobiP[0 * IMG_SIZE + col] = row_0;
42       d_xJacobiP[1 * IMG_SIZE + col] = row_1;
43     }
44   }
45 }
46
47 __global__ void PolynomialMatrix3(float *kXArray, float *
    kXArray_1st, float *kXArray_0th, float *kXArray_curr,
    float *d_xJacobiP, const long d_k, const long order,
```

```
    unsigned int n, float *kXArray_prev, float alpha, float
    beta)
48 {
49   int row = blockIdx.y * blockDim.y + threadIdx.y;
50   int col = blockIdx.x * blockDim.x + threadIdx.x;
51   if ((row < d_k) && (col < IMG_SIZE))
52   {
53     float kX_pre2, kX_pre1, kX = kXArray[row * IMG_SIZE + col
    ];
54     if (n == 2)
55     {
56       kX_pre2 = kXArray_0th[row * IMG_SIZE + col];
57       kX_pre1 = kXArray_1st[row * IMG_SIZE + col];
58     }
59     if (n == 3)
60     {
61       kX_pre2 = kXArray_1st[row * IMG_SIZE + col];
62       kX_pre1 = kXArray_curr[row * IMG_SIZE + col];
63     }
64     if (n > 3)
65     {
66       kX_pre2 = kXArray_prev[row * IMG_SIZE + col];
67       kX_pre1 = kXArray_curr[row * IMG_SIZE + col];
68     }
69     kXArray_prev[row * IMG_SIZE + col] = kX_pre1;
70     kXArray_curr[row * IMG_SIZE + col] = (1.f / (n * (n +
    alpha + beta))) * (((2.f * n - 1.f + alpha + beta) / 2.f)
```

```
    * ((2.f * n + alpha + beta) * kX + ((alpha * alpha - beta
    * beta) / (2.f * n + alpha + beta - 2.f))) * kX_pre1 - (((
    n - 1.f + alpha) * (n - 1.f + beta) * (2.f * n + alpha +
    beta) * kX_pre2) / (2.f * n - 2.f + alpha + beta)));
71   }
72 }
73
74 __global__ void PolynomialMatrix4(float *kXArray_curr, float
    *d_xJacobiP, const long d_k, const long order, unsigned
    int n, float *kXArray, float alpha, float beta)
75 {
76   int row = blockIdx.y * blockDim.y + threadIdx.y;
77   int col = blockIdx.x * blockDim.x + threadIdx.x;
78   float row_n;
79   float kXArray_1 = 1 - kXArray[row * IMG_SIZE + col];
80   float kXArrayp_1 = 1 + kXArray[row * IMG_SIZE + col];
81   __shared__ float JP_curr_sm[k][BLOCK_SIZE];
82   __shared__ float weight_s[k][BLOCK_SIZE];
83   if ((row < d_k) && (col < IMG_SIZE))
84   {
85     JP_curr_sm[threadIdx.y][threadIdx.x] = kXArray_curr[row *
     IMG_SIZE + col];
86     weight_s[threadIdx.y][threadIdx.x] = alpha == 0 && beta
    == 0 ? 1 : __powf(kXArray_1, alpha) * __powf(kXArrayp_1,
    beta);
87     if (row == 0)
88     {
```

```
89        row_n = 0.f;
90        for (int h = 0; h < d_k; h++)
91        {
92          row_n += JP_curr_sm[h][threadIdx.x] * weight_s[h][
    threadIdx.x];
93          __syncthreads();
94        }
95        d_xJacobiP[n * IMG_SIZE + col] = row_n;
96      }
97    }
98  }
99
100 //Host Side Calling function
101 PositveIndex<<<grid0, block0>>>(kXArray, kXArray_1st,
    kXArray_0th, d_xJacobiP, k, Mmax, alpha, beta);
102 PolynomialMatrix2<<<grid0, block0>>>(kXArray, kXArray_1st,
    kXArray_0th, d_xJacobiP, k, Mmax, alpha, beta);
103 for (unsigned int n = 2; n <= Mmax; n++)
104 {
105   PolynomialMatrix3<<<grid0, block0>>>(kXArray, kXArray_1st,
    kXArray_0th, kXArray_curr, d_xJacobiP, k, Mmax, n,
    kXArray_prev, alpha, beta);
106   PolynomialMatrix4<<<grid0, block0>>>(kXArray_curr,
    d_xJacobiP, k, Mmax, n, kXArray, alpha, beta);
107 }
```

Snippet 2: Polynomial Computing without k scheme Kernels

```
1 __global__ void IMGrec1(float *xlPoly, const long order,
```

```
    float alpha, float beta)
{
     int row = blockIdx.y * blockDim.y + threadIdx.y;
     int col = blockIdx.x * blockDim.x + threadIdx.x;

     if ((row == 0) && (col < IMG_SIZE))
     {
          float delta = 2. / IMG_SIZE;
          float row_0;
          float row_1;
          float row_index;
          row_index = -1.0f + 0.5f * delta + col * delta;
          row_0 = 1.0f;
          row_1 = 0.5 * (alpha - beta + (alpha + beta + 2) *
    row_index);

          xlPoly[0 * IMG_SIZE + col] = row_0;
          xlPoly[1 * IMG_SIZE + col] = row_1;
          float row_temp = 0.0f;
          for (unsigned int n = 2; n <= order; n++)
          {
               row_temp = (1. / (n * (n + alpha + beta))) *
    (((2. * n - 1. + alpha + beta) / 2.) * ((2. * n + alpha +
    beta) * row_index + ((alpha * alpha - beta * beta) / (2. *
     n + alpha + beta - 2.))) * row_1 - (((n - 1. + alpha) * (
    n - 1. + beta) * (2. * n + alpha + beta) * row_0) / (2. *
    n - 2. + alpha + beta)));
```

```
22             xlPoly[n * IMG_SIZE + col] = row_temp;
23             row_0 = row_1;
24             row_1 = row_temp;
25         }
26     }
27 }
```

Snippet 3: Symmetric Property and Matrix Transpose Kernels

```
1 __global__ void PolyTranspose(float *d_xJacobiP, float *
    d_yJacobiPT, const long order)
2 {
3   int row = blockIdx.y * blockDim.y + threadIdx.y;
4   int col = blockIdx.x * blockDim.x + threadIdx.x;
5   if (col < IMG_SIZE && row < (order + 1))
6     d_yJacobiPT[col * (order + 1) + row] = row % 2 == 0 ?
    d_xJacobiP[col + row * IMG_SIZE] : -1 * d_xJacobiP[col +
    row * IMG_SIZE];
7 }
8
9 __global__ void IMGrec1Transpose(float *xlPoly, float *
    xlPolyT, float *ylPoly, const long order)
10 {
11   int row = blockIdx.y * blockDim.y + threadIdx.y;
12   int col = blockIdx.x * blockDim.x + threadIdx.x;
13
14   if ((col < IMG_SIZE) && (row < (order + 1)))
15   {
16     xlPolyT[col * (order + 1) + row] = xlPoly[col + row *
```

```
    IMG_SIZE];
17   }
18   if ((row < (order + 1)) && (col < IMG_SIZE))
19     ylPoly[row * IMG_SIZE + col] = row % 2 == 0 ? xlPoly[row
    * IMG_SIZE + col] : -1 * xlPoly[row * IMG_SIZE + col];
20 }
```

Snippet 4: Matrix-Cuboid Multiplication Kernels

```
1  /*MatMul1: X-axis*/
2  __global__ void MatMul1(float *Left_Mat, float *Right_Cube,
    float *Result_Cube, const long Order_p1)
3  {
4      int row = blockIdx.y * blockDim.y + threadIdx.y;
5      int col = blockIdx.x * blockDim.x + threadIdx.x;
6      int plane = blockIdx.z * blockDim.z + threadIdx.z;
7
8      int tx = threadIdx.x;
9      int ty = threadIdx.y;
10     int tz = threadIdx.z;
11     __shared__ float ds_A[BLOCKDIM_8][BLOCKDIM_8];
12     __shared__ float ds_B[BLOCKDIM_8][BLOCKDIM_8][BLOCKDIM_8
    ];
13     float temp = 0.0f;
14     for (int ph = 0; ph < ceil(IMG_SIZE / (float)BLOCKDIM_8);
     ++ph)
15     {
16         if ((row < Order_p1) && (ph * BLOCKDIM_8 + tx) <
    IMG_SIZE)
```

```
17        {
18            ds_A[ty][tx] = Left_Mat[row * IMG_SIZE + ph *
   BLOCKDIM_8 + tx];
19        }
20        if ((ph * BLOCKDIM_8 + ty) < IMG_SIZE && col <
   IMG_SIZE && plane < IMG_SIZE)
21        {
22            ds_B[tz][ty][tx] = Right_Cube[plane * IMG_SIZE *
   IMG_SIZE + (ph * BLOCKDIM_8 + ty) * IMG_SIZE + col];
23        }
24        __syncthreads();
25
26        for (int i = 0; i < BLOCKDIM_8; ++i)
27        {
28            temp += ds_A[ty][i] * ds_B[tz][i][tx];
29        }
30        __syncthreads();
31    }
32
33    if ((row < Order_p1) && (col < IMG_SIZE) && (plane <
   IMG_SIZE))
34    {
35        Result_Cube[plane * IMG_SIZE * Order_p1 + row *
   IMG_SIZE + col] = temp;
36    }
37 }
38
```

```
39 /*MatMul2: Y-axis*/
40 __global__ void MatMul2(float *Left_Cube, float *Right_Mat,
     float *Result_Cube, const long Order_p1)
41 {
42     int row = blockIdx.y * blockDim.y + threadIdx.y;
43     int col = blockIdx.x * blockDim.x + threadIdx.x;
44     int plane = blockIdx.z * blockDim.z + threadIdx.z;
45     int tx = threadIdx.x;
46     int ty = threadIdx.y;
47     int tz = threadIdx.z;
48     __shared__ float ds_A[BLOCKDIM_8][BLOCKDIM_8][BLOCKDIM_8
     ];
49     __shared__ float ds_B[BLOCKDIM_8][BLOCKDIM_8];
50     float temp = 0.0f;
51 
52     for (int ph = 0; ph < ceil(IMG_SIZE / (float)BLOCKDIM_8);
     ++ph)
53     {
54         if ((row < Order_p1) && (ph * BLOCKDIM_8 + tx) <
     IMG_SIZE && plane < IMG_SIZE)
55         {
56             ds_A[tz][ty][tx] = Left_Cube[plane * Order_p1 *
     IMG_SIZE + row * IMG_SIZE + ph * BLOCKDIM_8 + tx];
57         }
58         if ((ph * BLOCKDIM_8 + ty) < IMG_SIZE && col <
     Order_p1)
59         {
```

```
60             ds_B[ty][tx] = Right_Mat[(ph * BLOCKDIM_8 + ty) *
    Order_p1 + col];
61         }
62         __syncthreads();
63 
64         for (int i = 0; i < BLOCKDIM_8; ++i)
65         {
66             temp += ds_A[tz][ty][i] * ds_B[i][tx];
67         }
68         __syncthreads();
69     }
70 
71     if (row < Order_p1 && col < Order_p1 && plane < IMG_SIZE)
72         Result_Cube[plane * Order_p1 * Order_p1 + row *
    Order_p1 + col] = temp;
73 }
74 /*MatMul3: Z-axis*/
75 
76 __global__ void MatMul3(float *Left_Mat, float *Right_Cube,
    float *Result_Cube, const long Order_p1, float *rho_in)
77 {
78     int row = blockIdx.y * blockDim.y + threadIdx.y;
79     int col = blockIdx.x * blockDim.x + threadIdx.x;
80     int plane = blockIdx.z * blockDim.z + threadIdx.z;
81 
82     int tx = threadIdx.x;
83     int ty = threadIdx.y;
```

```
84     int tz = threadIdx.z;
85     __shared__ float ds_A[BLOCKDIM_8][BLOCKDIM_8];
86     __shared__ float ds_B[BLOCKDIM_8][BLOCKDIM_8][BLOCKDIM_8
    ];
87 
88     float temp = 0.0f;
89     for (int ph = 0; ph < ceil(IMG_SIZE / (float)BLOCKDIM_8);
     ++ph)
90     {
91         if ((plane < Order_p1) && (ph * BLOCKDIM_8 + tx) <
    IMG_SIZE)
92         {
93             ds_A[tz][tx] = Left_Mat[plane * IMG_SIZE + ph *
    BLOCKDIM_8 + tx];
94         }
95         if (row < Order_p1 && col < Order_p1 && (ph *
    BLOCKDIM_8 + tz) < IMG_SIZE)
96         {
97             ds_B[tz][ty][tx] = Right_Cube[(ph * BLOCKDIM_8 +
    tz) * Order_p1 * Order_p1 + row * Order_p1 + col];
98         }
99         __syncthreads();
100 
101         for (int i = 0; i < BLOCKDIM_8; ++i)
102         {
103             temp += ds_A[tz][i] * ds_B[i][ty][tx];
104         }
```

```
105         __syncthreads();
106     }
107 
108     float mult1;
109     mult1 = 8. / (k * k * k * IMG_SIZE * IMG_SIZE * IMG_SIZE
        * rho_in[row] * rho_in[col] * rho_in[plane]);
110     Result_Cube[plane * Order_p1 * Order_p1 + row * Order_p1
        + col] = row + col + plane < Order_p1 ? temp * mult1 : 0.f
        ;
111 }
112 
113 /*MatMul4: Image reconstruction on Y-axis*/
114 __global__ void MatMul4(float *Left_Mat, float *Right_Cube,
        float *Result_Cube, const long Order_p1)
115 {
116     int row = blockIdx.y * blockDim.y + threadIdx.y;
117     int col = blockIdx.x * blockDim.x + threadIdx.x;
118     int plane = blockIdx.z * blockDim.z + threadIdx.z;
119 
120     int tx = threadIdx.x;
121     int ty = threadIdx.y;
122     int tz = threadIdx.z;
123     __shared__ float ds_A[BLOCKDIM_8][BLOCKDIM_8];
124     __shared__ float ds_B[BLOCKDIM_8][BLOCKDIM_8][BLOCKDIM_8
        ];
125     float temp = 0.0f;
126 
```

```
127     for (int ph = 0; ph < ceil(Order_p1 / (float)BLOCKDIM_8);
    ++ph)
128     {
129         if ((row < IMG_SIZE) && (ph * BLOCKDIM_8 + tx) <
    Order_p1)
130         {
131             ds_A[ty][tx] = Left_Mat[row * Order_p1 + ph *
    BLOCKDIM_8 + tx];
132         }
133         if ((ph * BLOCKDIM_8 + ty) < Order_p1 && col <
    Order_p1 && plane < IMG_SIZE)
134         {
135             ds_B[tz][ty][tx] = Right_Cube[plane * Order_p1 *
    Order_p1 + (ph * BLOCKDIM_8 + ty) * Order_p1 + col];
136         }
137         __syncthreads();
138
139         for (int i = 0; i < BLOCKDIM_8; ++i)
140         {
141             temp += ds_A[ty][i] * ds_B[tz][i][tx];
142         }
143         __syncthreads();
144     }
145
146     if ((row < IMG_SIZE) && (col < Order_p1) && (plane <
    IMG_SIZE))
147     {
```

```
148         Result_Cube[plane * IMG_SIZE * Order_p1 + row *
    Order_p1 + col] = temp;
149     }
150 }
151 
152 /*MatMul5: Image reconstruction on X-axis*/
153 __global__ void MatMul5(float *Left_Cube, float *Right_Mat,
    float *Result_Cube, const long Order_p1)
154 {
155     int row = blockIdx.y * blockDim.y + threadIdx.y;
156     int col = blockIdx.x * blockDim.x + threadIdx.x;
157     int plane = blockIdx.z * blockDim.z + threadIdx.z;
158     int tx = threadIdx.x;
159     int ty = threadIdx.y;
160     int tz = threadIdx.z;
161     __shared__ float ds_A[BLOCKDIM_8][BLOCKDIM_8][BLOCKDIM_8
    ];
162     __shared__ float ds_B[BLOCKDIM_8][BLOCKDIM_8];
163     float temp = 0.0f;
164     for (int ph = 0; ph < ceil(Order_p1 / (float)BLOCKDIM_8);
     ++ph)
165     {
166         if ((row < IMG_SIZE) && (ph * BLOCKDIM_8 + tx) <
    Order_p1 && plane < IMG_SIZE)
167         {
168             ds_A[tz][ty][tx] = Left_Cube[plane * IMG_SIZE *
    Order_p1 + row * Order_p1 + ph * BLOCKDIM_8 + tx];
```

```
169         }
170         if ((ph * BLOCKDIM_8 + ty) < Order_p1 && col <
    IMG_SIZE)
171         {
172             ds_B[ty][tx] = Right_Mat[(ph * BLOCKDIM_8 + ty) *
     IMG_SIZE + col];
173         }
174         __syncthreads();
175 
176         for (int i = 0; i < BLOCKDIM_8; ++i)
177         {
178             temp += ds_A[tz][ty][i] * ds_B[i][tx];
179         }
180         __syncthreads();
181     }
182 
183     if ((row < IMG_SIZE) && (col < IMG_SIZE) && (plane <
    Order_p1))
184     {
185         Result_Cube[plane * IMG_SIZE * IMG_SIZE + row *
    IMG_SIZE + col] = temp;
186     }
187 }
188 
189 /*MatMul6: Image reconstruction on Z-axis*/
190 __global__ void MatMul6(float *Left_Mat, float *Right_Cube,
    float *Result_Cube, const long Order_p1)
```

```
191 {
192      int row = blockIdx.y * blockDim.y + threadIdx.y;
193      int col = blockIdx.x * blockDim.x + threadIdx.x;
194      int plane = blockIdx.z * blockDim.z + threadIdx.z;
195      int tx = threadIdx.x;
196      int ty = threadIdx.y;
197      int tz = threadIdx.z;
198      __shared__ float ds_A[BLOCKDIM_8][BLOCKDIM_8];
199      __shared__ float ds_B[BLOCKDIM_8][BLOCKDIM_8][BLOCKDIM_8
    ];
200      float temp = 0.0f;
201      for (int ph = 0; ph < ceil(Order_p1 / (float)BLOCKDIM_8);
     ++ph)
202      {
203          if ((plane < IMG_SIZE) && (ph * BLOCKDIM_8 + tx) <
    Order_p1)
204          {
205              ds_A[tz][tx] = Left_Mat[plane * Order_p1 + ph *
    BLOCKDIM_8 + tx];
206          }
207          if (row < IMG_SIZE && col < IMG_SIZE && (ph *
    BLOCKDIM_8 + tz) < Order_p1)
208          {
209              ds_B[tz][ty][tx] = Right_Cube[(ph * BLOCKDIM_8 +
    tz) * IMG_SIZE * IMG_SIZE + row * IMG_SIZE + col];
210          }
211          __syncthreads();
```

```
212
213         for (int i = 0; i < BLOCKDIM_8; ++i)
214         {
215             temp += ds_A[tz][i] * ds_B[i][ty][tx];
216         }
217         __syncthreads();
218     }
219
220     if ((row < IMG_SIZE) && (col < IMG_SIZE) && plane <
    IMG_SIZE)
221     {
222         Result_Cube[plane * IMG_SIZE * IMG_SIZE + row *
    IMG_SIZE + col] = temp;
223     }
224 }
```

Snippet 5: Computation of $\rho_n^{(\alpha,\beta)}$

```
1 float *CoeforJacobi(int n, float alpha, float beta)
2 {
3     float rho_0 = pow(2., (alpha + beta + 1)) * ((tgamma(
    alpha + 1.) * tgamma(beta + 1.)) / tgamma(alpha + beta +
    2.));
4     static float rho_array[Mmax + 1];
5     rho_array[0] = rho_0;
6     float rho_temp;
7     for (int m = 1; m <= n; m++)
8     {
9         rho_temp = (((m + alpha) * (m + beta) * (2.f * m +
```

```
    alpha + beta - 1.f)) / (m * (m + alpha + beta) * (2.f * m
    + alpha + beta + 1.f))) * rho_0;
10          rho_0 = rho_temp;
11          rho_array[m] = rho_temp;
12      }
13      return rho_array;
14  }
```