國立台灣大學電機資訊學院電子工程學系
碩士論文
Graduate Institute of Electronics Engineering
College of Electrical Engineering and Computer Science
National Taiwan University
Master Thesis

利用張量近似法表示多維視覺資料之硬體架構與實現
Hardware Architecture and Implementation of Tensor Approximation for Multi-Dimensional Visual Data

楊其昀
Chi-Yun Yang

指導教授: 盧奕璋博士
Advisor: Yi-Chang Lu, Ph.D.

中華民國 107 年 2 月
February, 2018

doi:10.6342/NTU201800667

國立臺灣大學碩士學位論文
口試委員會審定書

利用張量近似法表示多維視覺資料之硬體架構與實現
Hardware Architecture and Implementation of Tensor Approximation for Multi-Dimensional Visual Data

本論文係楊其昀君（R03943126）在國立臺灣大學電子工程學研究所完成之碩士學位論文，於民國 107 年 2 月 26 日承下列考試委員審查通過及口試及格，特此證明

口試委員：

（指導教授）

系主任、所長

誌謝

用了整整四個年頭，才終於在當兵前最後一個禮拜完成了論文的口試，這幾年在實驗室的研究生生涯也終於要宣告結束。能夠在台大電子所完成學業取得碩士學位，除了自己的努力外，更大一部分必須歸功於周遭的人給我的幫助。

首先必須感謝我的指導教授 ── 盧奕璋老師。老師對於我們不管在研究方向或是人生規劃的決定都給予相當的尊重與支持，也會適時提供寶貴的意見供我們參考。在老師的帶領下，實驗室的研究風氣一直都很自由，成員間的感情也都相當融洽。感謝老師支持我出國交換一年的決定，提供我出國前後在研究時程上的規劃建議，在最後這半年更是為了配合我研發替代役的期限，讓我在論文完成前就安排口試。很高興當初選擇了進入這個實驗室，在老師的指導下做研究，從老師的身上真的學到很多。

感謝博理 430 實驗室的大家。感謝碩一時帶領我們的學長姐，陽明、鈺翔和肖璐，在研究的方法和態度上一直是我的榜樣，讓我在研究上能夠順利的步上軌道。謝謝同屆的東霖、皓瑋和纓喻，一起修課一起熬夜，在研究上也都幫了我很多，是相處時間最長的好夥伴。下一屆的學弟，聖睿、麒銘、冠杰和乃群，雖然比我晚進來實驗室，但在研究上也時常詢問你們的意見，特別感謝聖睿，在最後硬體的設計以及口試上提供我很多重要的資訊。R05 的學弟，岳霖、政彥、茂然、健安、承曄和士軒，以及 R06 的芳宗和韋頤，雖然相處的時間比較少，但有你們在讓實驗室氣氛變得很歡樂，最後在口試的時候也都幫了我很多。最後一定要感謝大學長育誠，總是幫我們解決各種生活上或研究上的疑難雜症，親切又非常值得信賴。就讀研究所的這幾年，實驗室就像是第二個家一樣，每天跟大家一起討論研究內容一起吃飯聊天，讓做研究的過程更加順利也更加愉快，也留下許多寶貴的回憶。

最後也要感謝我的家人，提供我我所擁有的一切也包容我的一切，

doi:10.6342/NTU201800667

讓我能無後顧之憂的出國交換和做研究。也感謝所有在我身邊的朋友，
你們的陪伴和關心也是我能走到這一步不可或缺的部分。

doi:10.6342/NTU201800667

摘要

在電腦視覺與電腦圖學的領域中，常常需要對多維的視覺資料進行分析與處理。隨著使用的的資料量越來越大，用精簡的方式儲存與表示資料也變成一個重要的研究議題。不同於傳統上維度縮減常使用的主成分分析，張量近似可以將資料在維持其原本多維結構的情況下進行維度縮減，更好地利用多維結構中的資料相關性，分別對各個維度進行維度縮減，也提供了在壓縮時的彈性。在需要進行快速影像生成的應用中，資料在經過張量近似壓縮後，重建時的運算量使其無法達到實時生成的需求，因此適用於快速影像生成的張量近似演算法也陸續被提出。

在本篇論文中，我們針對適用於快速影像生成的張量近似演算法進行討論，並且提出了一個硬體加速架構來對其中的分群張量近似演算法進行加速。因為龐大資料量與運算量，張量近似的運算過程相當耗時，利用硬體中平行運算的技巧，可以加快其運算速度。我們使用 10 塊靜態隨機存取記憶體，組成高頻寬的記憶體陣列作為內部儲存區域，在資料輸入時得以取出需要的資料進行平行運算。另外也實作了適用於大尺寸長方形矩陣的 Hestenes-Jacobi 奇異值分解演算法。我們的硬體架構可以對 $128 \times 128 \times 128 \times 128$ 的四維張量進行分群張量近似演算法，使用 TSMC 40nm 製程，運行於 476 MHz 的時脈頻率，運算速度為軟體實作結果的 9.41 倍。完成的晶片面積為 3.151 mm^2，消耗功率為 744.8 mW。

關鍵字：張量近似、資料壓縮、多維資料、硬體設計

doi:10.6342/NTU201800667

doi:10.6342/NTU201800667

Abstract

In the field of computer vision and computer graphics, processing and analyzing of multidimensional visual data are widely used. As the size of data to be processed increases, how to represent and store data in a compact way becomes an important issue. Unlike traditional dimensionality reduction algorithm, like principal component analysis (PCA), tensor approximation is used to analyzes data while the multidimensional structure is retained, which allows the exploitation of spatial redundancy. Dimensionality reduction along each mode also makes the process more flexible. However, for application which requires rapid image rendering, the computational cost of data reconstruction after applying tensor approximation is very high. As a result, several modified tensor approximation algorithms support fast reconstruction have been proposed.

Because of the enormous size of data and computational cost, tensor approximation suffers from long computation time.In this thesis, we propose a hardware accelerator for one of the modified algorithm, clustered tensor approximation (CTA). With parallel processing techniques in hardware implementation, speed-up can be achieved. We utilize 10 SRAMs to compose a memory array with high bandwidth so that all corresponding data of inputs can be fetched and manipulated simultaneously. We also implement a singular value decomposition (SVD) processor based on Hestenes-Jacobi algorithm which is suitable for decomposition of large rectangular matrices. The architecture we propose can apply clustered tensor approximation to a tensor with a dimension of $128 \times 128 \times 128 \times 128$. Using TSMC 40nm technology, the hardware can operate at 476 MHz. The approximation process can be computed 9.41 times faster than the software version. The chip area is 3.151 mm² and the power consumption is 744.8 mW.

Keywords: tensor approximation, data compression, multidimensional data,

doi:10.6342/NTU201800667

hardware architecture

doi:10.6342/NTU201800667

Contents

doi:10.6342/NTU201800667

doi:10.6342/NTU201800667

List of Figures

doi:10.6342/NTU201800667

doi:10.6342/NTU201800667

List of Tables

doi:10.6342/NTU201800667

doi:10.6342/NTU201800667

Chapter 1　緒論

多維視覺資料的處理與分析在電腦視覺 (computer vision) 與電腦圖學 (computer graphics) 的領域的許多應用中是不可或缺的部分。除了許多資料的基本結構就具有多維度的性質，如影像具有二維的結構，體資料 (volume data) 則具有三維的結構，各種影響資料內容的獨立因素也會增加資料組的維度。透過對影響資料的各種因素進行分析，結果可被應用於各種層面。然而，當納入分析的因素以及取樣數量的增加，資料的維度與各維度的大小也隨之增加，需要處理的資料量也因此大幅成長。為了處理大量的多維資料，如何將資料以更精簡的方式儲存與表示，便成為了一個重要的議題。在這篇論文中，我們會就近年來吸引許多注意的多重線性 (multilinear) 維度縮減 (dimensionality reduction) 模型，張量近似，以及其適用於影像生成應用的改良演算法進行討論，並且提出一個張量近似的運算硬體架構來將其運算進行加速。

1.1　多維視覺資料之應用

多維視覺資料的應用相當廣泛。光場 (light field)[1] 資料用四維的架構儲存了影像中光線的方向，並將此資訊應用於後續的影像處理。功能性磁共振造影 (functional Magnetic Resonance Imaging, fMRI) 資料則是由一連串的三維磁共振影像組成的四維資料，可用於醫學分析用途。

而在本篇論文中，我們將使用雙向紋理函數 (Bidirectional Texture Function, BTF)[2] 作為我們討論與實驗時的主要應用。雙向紋理函數包含了一個表面紋理隨空間位置、觀測視角以及光照角度變化的外觀以及光線的反射、散射性質。一個雙向紋理函數具有六個變數，其中兩個變數用於表示觀測視角、兩個變數用於表示光照角度，最後兩個變數則用來代表於紋理表面的位置。有了雙向紋理函數，在電腦圖學上可以更加逼真地描繪出真實世界中的紋理。雙向紋理函數可以透過在不同的光照角度下由不同的角度對一個紋理樣本進行取樣來取得，然而若要更詳盡的表示真實紋理的外觀，取樣的數量也必須隨之增加，資料量相當龐大。

doi:10.6342/NTU201800667

1.2 主成分分析

主成分分析 (principal component analysis) 是最為常用的維度縮減演算法之一。其主要的概念是透過對共變異數矩陣進行特徵值分解來獲得資料的主要成分，也就是特徵向量 (eigenvector)，以及資料對主要成分的投射比重，每一筆資料可以用得到的主成分和比重來表示。若是將其中較不重要的主成分，也就是對應到較小特徵值 (eigenvalue) 的特徵向量除去，也還可以用保留下來的主成分來表示資料，因此可以達到維度縮減的效果。儘管拋棄部分的主成分也會導致資訊的遺失，但是透過去除對應較小特徵值的主成分的方式所得到的低維度表示法也將是遺失資訊最少的最佳解。在實作上，主成分分析也可以利用奇異值分解 (singular value decomposition, SVD) 來達成。而張量近似也可以算是主成分分析於多重線性分析的一般化。張量近似是本論文的核心，相關內容將於第二張中討論。

1.3 論文架構

本篇論文接下來的部分由四個章節組成。在第二章中，會就張量近似的基本概念做說明，並以雙向紋理函數作為張量近似應用於多維視覺資料的例子做說明，最後則會介紹適用於快速影像生成的張量近似演算法。第三章會就第二章中所討論的演算法以雙向紋理函數進行實驗，並探討各個演算法的效果以及參數的設定。而在第四章中，我們提出了一個硬體加速架構來加速張量近似的運算，並會對硬體內部的設計做詳細的介紹。最後，第五章是結論以及未來的展望。

doi:10.6342/NTU201800667

Chapter 2　以張量近似法表示多維視覺資料

2.1　張量簡介

2.1.1　張量的基本概念

張量 (tensor)，是向量和矩陣在更高階 (order) 的一般化，換言之，向量即是一階的張量，而矩陣則是二階的張量。我們將一個 N 階的張量以 $\mathcal{A} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$ 表示，其元素則記為 $a_{i_1 \ldots i_n \ldots i_N}$。在張量中，一個模 n(n-mode) 的向量是由集合 $\mathcal{A} : \{a_{i_1 \ldots i_n \ldots i_N}\}_{i_n=1}^{I_n}$ 中的元素所組成，若將所有的模 n 向量作為行向量則可以組成矩陣 $uf_n(\mathcal{A}) \in \mathbb{R}^{I_n \times (I_1 \ldots I_{n-1} I_{n+1} \ldots I_N)}$，也就是將張量沿模 n 展開 (unfold) 為一個矩陣，圖 2.1以一個三階張量為例展示了沿著各個模展開為矩陣的方式，其中各個矩陣的行向量即是該模的向量。張量 $\mathcal{A} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$ 也可以與矩陣 $\mathbf{M} \in \mathbb{R}^{J_n \times I_n}$ 進行模 n 的乘法，記為 $\mathcal{A} \times_n \mathbf{M}$，其結果會是一個張量 $\mathcal{B} \in \mathbb{R}^{I_1 \times \ldots \times I_{n-1} \times J_n \times I_{n+1} \ldots \times I_N}$ 張量 $\mathcal{B}$ 的元素則定義為 $b_{i_1 \ldots i_{n-1} j_n i_{n+1} \ldots i_N} = \sum_{i_n} a_{i_1 \ldots i_{n-1} i_n i_{n+1} \ldots i_N} m_{j_n i_n}$，若以展開的矩陣來表示模 n 乘法，可以表示成 $uf_n(\mathcal{B}) = \mathbf{M} uf_n(\mathcal{A})$。在這裡，為了之後的列式能

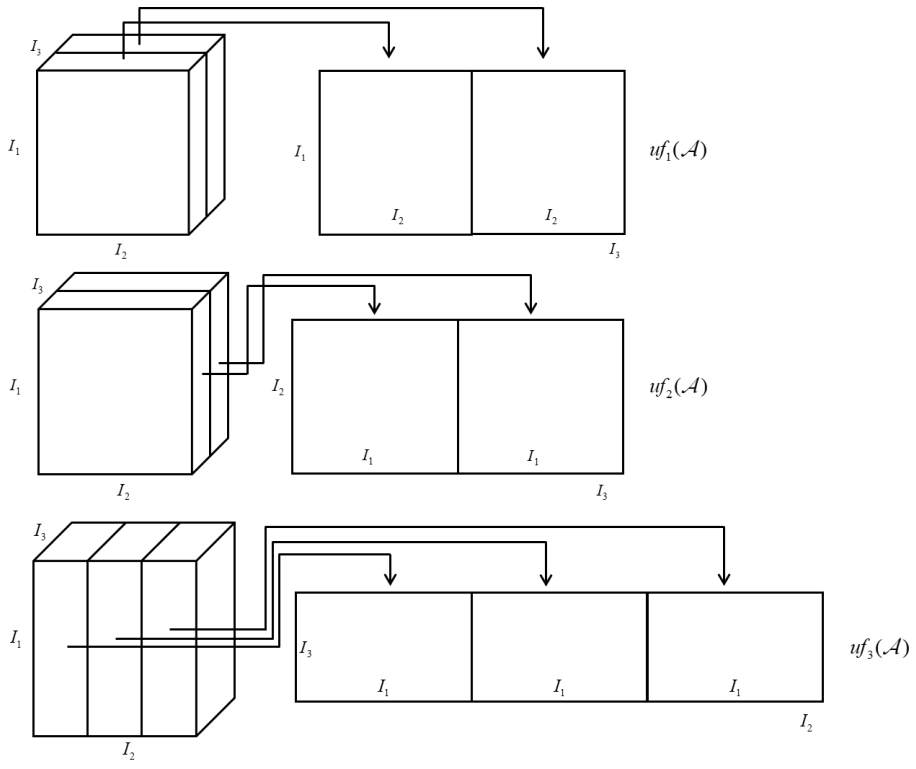


圖 2.1: 三階張量沿各個模展開示意圖

doi:10.6342/NTU201800667

夠更加簡潔，我們定義一連串的模 n 乘法的表示法為：

$$\mathcal{A}\bigtimes_{n=1}^{N}\mathbf{M}_n = \mathcal{A}\times_1\mathbf{M}_1\times_2\mathbf{M}_2\ldots\times_N\mathbf{M}_N \tag{2.1}$$

此外，模 n 乘法也具有可交換和可結合的性質，如：

$$\begin{aligned}(\mathcal{A}\times_n\mathbf{F})\times_m\mathbf{G} &= (\mathcal{A}\times_m\mathbf{G})\times_n\mathbf{F}\\ (\mathcal{A}\times_n\mathbf{F})\times_n\mathbf{G} &= \mathcal{A}\times_n(\mathbf{F}\cdot\mathbf{G})\end{aligned} \tag{2.2}$$

2.1.2 張量近似

在處理大量資料的時候，維度縮減 (dimensionality reduction) 是一種常用的壓縮方式，在矩陣的主成分分析中，維度縮減的最佳解可以透過截去奇異值分解中對應到較小奇異值的奇異向量來得到。而對於張量的維度縮減，或稱為張量近似，也有著類似的處理模式。

給定一個張量 $\mathcal{A}\in\mathbb{R}^{I_1\times I_2\times\ldots\times I_N}$，張量近似的問題可以描述為尋找一個近似張量 $\tilde{\mathcal{A}}\in\mathbb{R}^{I_1\times I_2\times\ldots\times I_N}$ 滿足下列目標函式：

$$\tilde{\mathcal{A}} = \arg\min_{\hat{\mathcal{A}}}\left\|\mathcal{A}-\hat{\mathcal{A}}\right\|^2 \tag{2.3}$$

這個張量同時也具有較小的秩，即：

$$R_n = Rank_n(\tilde{\mathcal{A}}) \le Rank_n(\mathcal{A}) = Rank(uf_n(\mathcal{A})) \tag{2.4}$$

而求得的結果可以分解為一個核心張量 (core tensor) 和 N 個基底矩陣：

$$\tilde{\mathcal{A}} = \mathcal{Z}\bigtimes_{n=1}^{N}\mathbf{U}_n \tag{2.5}$$

其中 $\mathcal{Z}\in\mathbb{R}^{R_1\times R_2\times\ldots\times R_N}$ 是核心張量，$\mathbf{U}_1\in\mathbb{R}^{I_1\times R_1}$ 是模 1 的基底矩陣，$\mathbf{U}_2\in\mathbb{R}^{I_2\times R_2}$ 是模 2 的基底矩陣，以此類推總共 N 個基底矩陣。也就是說，我們可以用較小的核心向量和基底矩陣來表示原來較大的張量，達到維度縮減的壓縮效果，圖 2.2是以一個三階張量為例，進行張量近似的示意圖。

要達成張量近似，一個較簡易的方式是使用高階奇異值分解 (high order singular value decomposition, HOSVD)，或稱為 N 模奇異值分解 (N-mode SVD)。在

doi:10.6342/NTU201800667

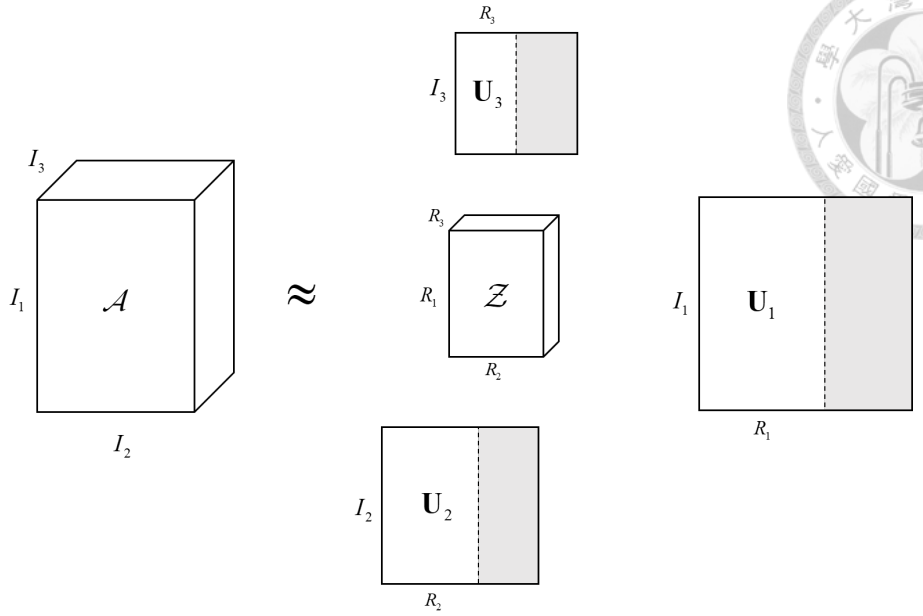


圖 2.2: 三階張量之張量近似示意圖

對矩陣的奇異值分解中，矩陣 $\mathbf{A} \in \mathbb{R}^{I_1 \times I_2}$ 可以被分解成其行向量空間的基底矩陣 $\mathbf{U}_1 \in \mathbb{R}^{I_1 \times I_1}$、包含其奇異值的對角矩陣 $\mathbf{S} \in \mathbb{R}^{I_1 \times I_2}$，以及其列向量空間的基底矩陣 $\mathbf{U}_2 \in \mathbb{R}^{I_2 \times I_2}$，如式 2.6。

$$\mathbf{A} = \mathbf{U}_1 \mathbf{S} \mathbf{U}_2^{\mathrm{T}} \tag{2.6}$$

正如前面提過，矩陣實際上也是一個二階的張量，因此我們可以將式 2.6以張量模 n 乘法的形式表示成：

$$\mathbf{A} = \mathbf{S} \times_1 \mathbf{U}_1 \times_2 \mathbf{U}_2 \tag{2.7}$$

這個概念延伸到高階張量就能夠得到如式 2.5的形式，核心張量就類比於矩陣奇異值分解中的奇異值對角矩陣，包含著各個模的基底矩陣之間的交互關係。

而在做法上，我們將張量 $\mathcal{A}$ 沿著各模展開為 $uf_n(\mathcal{A})$，接著將展開後的矩陣做一般的矩陣奇異值分解，再將得到的行向量空間基底矩陣作為式 2.5中的 $\mathbf{U}_n$。如此一來便可得到各個模的基底矩陣。因為這些基底矩陣都是正交矩陣 (orthogonal matrix)，具有轉置矩陣即為逆矩陣的性質，可以用式 2.8的方式來得到核心張量。

$$\mathcal{Z} = \mathcal{A} \bigtimes_{n=1}^{N} \mathbf{U}_n^T \tag{2.8}$$

若是將各個模的基底向量，先依照需要壓縮的量截去對應到較小奇異值的奇異向量，再以這些經過截切 (truncation) 的基底矩陣計算核心張量，就可以得到維度經過縮減的核心張量，達到壓縮的目的。

doi:10.6342/NTU201800667

2.2 張量近似應用於多維視覺資料

多維視訊資料通常都具有資料量龐大的特性，因此壓縮演算法也就顯得特別重要，主成分分析在這方面廣泛的被應用，而近年來，張量近似在這方面也逐漸受到重視。相較於傳統的主成分分析在做法上會將所有資料轉化成矩陣在進行分析，張量近似可以直接將資料依據其特性把不同的影響要素分配到各個維度上。如此一來在維度縮減上便可以更加有彈性，根據不同的應用，使用者可以決定在那些維度上做較多的縮減。因此只要有適當的維度分配，在相同壓縮率的情況下，張量近似通常能夠有更符合需求的近似結果。另一方面，因為張量近似會產生多個基底矩陣，在資料的表示上也較為精簡。在本篇論文中，我們將以雙向紋理函數 (bidirectional texture function, BTF) 作為主要的應用，進行實驗與討論。

為獲得特定紋理的雙向紋理函數，常見的做法是對該紋理的樣本從不同的視角，以及不同的光照角度下擷取該紋理的影像。以這些取樣得到的影像來建構張量，不但可以透過張量近似對大量的影像資訊作壓縮，近似後的結果也可以作為一個生成模型 (generative model)，用於生成 (render) 紋理在任何視角和任何光照角度下的樣貌。

在張量的建構方面，若我們對一個 $I_x \times I_y$ 的二維紋理樣本從 I_v 個不同的視角進行擷取，並於每一個視角給予 I_l 個不同角度的光照，我們可以將這些影像資料建構成一個四階張量 $\mathcal{A}_{BTF} \in \mathbb{R}^{I_x \times I_y \times I_v \times I_l}$。值得注意的是，在一般的分析方法中，習慣將二維的影像展開成一個一維的向量，然而如此一來便忽略了影像資料中鄰近像素通常擁有相近值的特性，在近似的過程中也就沒辦法有效的將這些冗餘資訊 (redundancy) 去除。而由於張量是一個高階的結構，我們可以直接將影像以二維的方式存於張量中，避免掉這樣的情況。另外，雖然視角以及光照的角度也可以根據其表示參數分配到多個維度，這些維度的大小通常很小，在近似時較難達到更高的壓縮倍率，因此在這裡不採用這樣的作法。

完成了張量的建構之後，便可以利用 2.1.2節的方式達到雙向紋理函數的張量近似，如前面所提到，我們可以用張量 $\tilde{\mathcal{A}}_{BTF} \in \mathbb{R}^{I_x \times I_y \times I_v \times I_l}$ 來近似原本的張量 $\mathcal{A}_{BTF}$，其又可以分解為：

$$\tilde{\mathcal{A}}_{BTF} = \mathcal{Z}_{BTF} \times_1 \mathbf{U}_x \times_2 \mathbf{U}_y \times_3 \mathbf{U}_v \times_4 \mathbf{U}_l \tag{2.9}$$

其中 $\mathcal{Z}_{BTF} \in \mathbb{R}^{R_x \times R_y \times R_l \times R_v}$，$\mathbf{U}_x \in \mathbb{R}^{I_x \times R_x}$，$\mathbf{U}_y \in \mathbb{R}^{I_y \times R_y}$，$\mathbf{U}_v \in \mathbb{R}^{I_v \times R_v}$，$\mathbf{U}_l \in \mathbb{R}^{I_l \times R_l}$。我們可以透過理解這些矩陣和張量在雙向紋理函數的應用中所代表的意

doi:10.6342/NTU201800667

義，對它們在張量近似中所扮演的角色有更進一步的了解。舉例來說，$\mathbf{U}_v$ 的行向量展開了一個視角的向量空間，可以透過線性組合組成一個包含特定像素於特定光照角度下從 I_v 個視角觀測到的亮度值的向量。$\mathbf{U}_v$ 的列向量則包含將張量沿模 v 展開後的列向量組合成定對應視角所需的係數。其他基底矩陣的行向量與列向量也同樣有著類似的特性，而核心張量則是包含了各個基底矩陣之間的交互關係。另外，若我們將 $\mathbf{U}_x$ 和 $\mathbf{U}_y$ 沿著各自的模與核心向量做模 n 乘法如：

$$\mathcal{T} = \mathcal{Z}_{BTF} \times_1 \mathbf{U}_x \times_2 \mathbf{U}_y \tag{2.10}$$

所得的到 $\mathcal{T} \in \mathbb{R}^{I_x \times I_y \times R_v \times R_l}$ 便包含著一組沿著視角與光照角度兩個模變動的基底影像，如圖 2.3所示，透過對這些基底影像進行線性組合，就可以生成出任何視角在任何光照角度下的紋理影像。有了 $\mathcal{T}$ 之後，在生成需要的紋理影像 $\mathbf{A}$ 時，只需

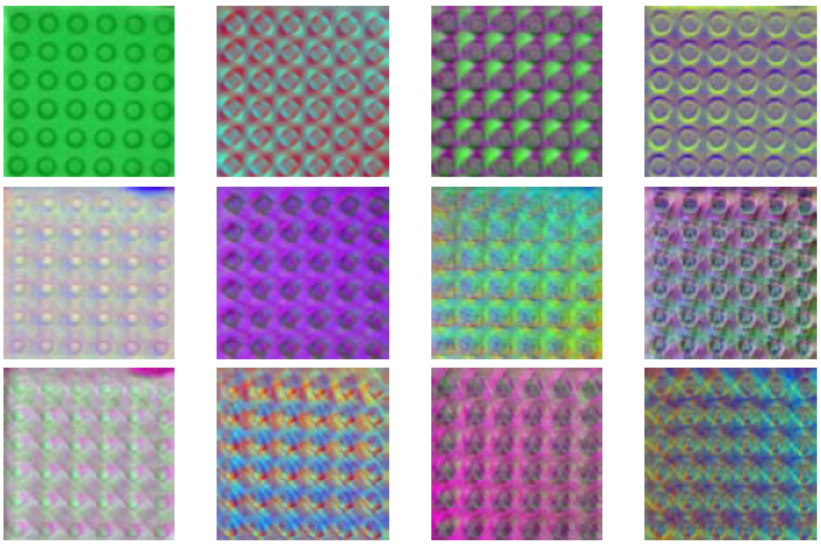


圖 2.3: 沿視角與光照角度兩個模變動之基底影像 [3]

要從 $\mathbf{U}_v$ 和 $\mathbf{U}_l$ 中取出對應視角及對應光照角度的列向量 $\mathbf{v}$ 和 $\mathbf{l}$，並計算：

$$\mathbf{A} = \mathcal{T} \times_3 \mathbf{v} \times_4 \mathbf{l} \tag{2.11}$$

即可得到在該視角及光照角度下得到的影像。要生成原本的樣本中沒有的視角或光照角度的影像時，則可以從視角及光照角度的取樣半球面上選取最近的三個視角或光照角度，並利用其重心座標 (barycetric coordinate) 將代表那三個視角或光照角度的列向量做凸性組合 (convex combination)，便可以得到生成新的視角或光照角度影像所需的向量，如圖 2.4所示。從這裡也可以看出，張量近似可以比主成分

doi:10.6342/NTU201800667

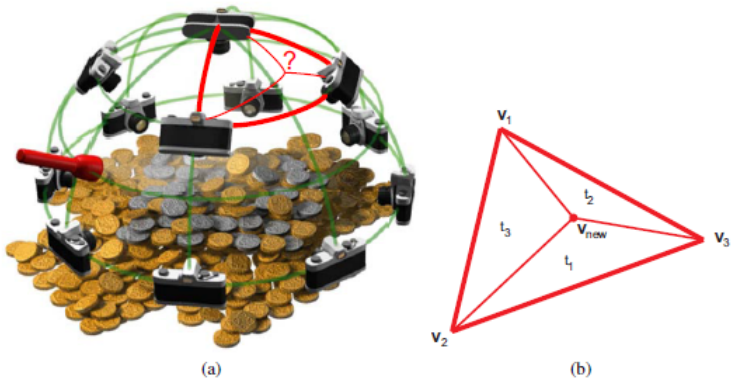


圖 2.4: 利用重心座標及凸性組合得到新視角或光照角度之生成向量 [4]

分析更加精簡的表示資料。在這個例子中，在不做維度縮減的情況下，使用主成分分析會得到 $I_v \times I_l$ 個長度為 $I_x \times I_y$ 的基底向量，若要重建出特定的紋理影像，我們需要用 $I_v \times I_l$ 個係數來將這些基底向量依特定比例做組合，也就是說每個紋理影像必須使用 $I_v \times I_l$ 個係數來做表示。而使用張量近似則時，如式 2.10 所示，僅需要兩個長度分別為 I_v 和 I_l 的向量來重建出特定紋理影像，因此每個紋理影像只需要用 $I_v + I_l$ 個係數來做表示即可。

2.3 交替最小平方法 (Alternating Least Squares algorithm)

不同於矩陣的維度縮減，單純的截切通常沒有辦法在張量的維度縮減上得到最佳解。[5] 中提出了一種迭代式 (iterative) 的交替最小平方 (alternating least squares, ALS) 演算法以達到更好的結果。交替最小平方法在每一次的迭代中僅針對張量的其中一個模的基底矩陣做最佳化，其他模的基底矩陣則保持不變。舉例來說，在對模 n 做最佳化的過程中，為了得到新的模 n 基底矩陣 $\mathbf{U}_n^{(j+1)} \in \mathbb{R}^{I_n \times R_n}$，會先固定現有的其他模的基底矩陣 $\mathbf{U}_1^{(j+1)}, ...\mathbf{U}_{n-1}^{(j+1)}, \mathbf{U}_{n+1}^{(j)}, ..., \mathbf{U}_N^{(j)}$，以式 2.8的方式得到在除 n 以外其他模上得到維度縮減的張量 $\mathcal{B} \in \mathbb{R}^{R_1 \times ... \times R_{n-1} \times I_n \times R_{n+1} ... \times R_N}$，接著將所得的張量做模 n 的高階奇異值分解，最後再擷取主要的 R_n 個奇異向量即可得到新的模 n 基底張量並在其他模的最佳化流程中使用。這樣的最佳化過程會輪流用於各個模，並且重複直到結果收斂。其中在初始值的部分，根據 [5] 提出的方法，可以直接對目標張量做各個模的高階奇異值分解，再以截切後的基底矩陣作為初始值。然而，這樣的方式必須要對原本未經過縮減的張量先做一次高階奇異值分解，以模 n 為例，為了取得初始的模 n 基底矩陣，必須對

doi:10.6342/NTU201800667

$uf_n(\mathcal{A}) \in \mathbb{R}^{I_n \times (I_1 \ldots I_{n-1} I_{n+1} \ldots I_N)}$ 做奇異值分解，相較於之後對模 n 進行最佳化時是對 $uf_n(\mathcal{B}) \in \mathbb{R}^{I_n \times (R_1 \ldots R_{n-1} R_{n+1} \ldots R_N)}$ 做奇異值分解，所需的資源及時間都更多。為了解決這個問題，可以使用 $[\mathbf{I}_{R_n}\mathbf{0}]^T$ 作為替代，其中 $\mathbf{I}_{R_n}$ 是一個 $R_n \times R_n$ 的單位矩陣，或是使用隨機產生的矩陣作為初始值，如此一來可以簡化運算量，且在結果上也沒有明顯的影響。交替最小平方法的流程可以見演算法 1所示。

演算法 1 交替最小平方演算法流程 [5]

給定: $\mathcal{A}, \{R_n\}_{n=1}^N$

求: $\mathcal{Z}, \{\mathbf{U}_n\}_{n=1}^N$

初始化 $\{\mathbf{U}_n^{(0)}\}_{n=1}^N$

$j \leftarrow 0$

repeat

for $n \leftarrow 1$ to N **do**

$\mathcal{B} \leftarrow \mathcal{A} \times_1 \mathbf{U}_1^{(j+1)T} \ldots \times_{n-1} \mathbf{U}_{n-1}^{(j+1)T} \times_{n+1} \mathbf{U}_{n+1}^{(j)T} \ldots \times_N \mathbf{U}_N^{(j)T}$

$\mathbf{U} \leftarrow$ SVD($uf_n(\mathcal{B})$) 所得行向量基底矩陣

$\mathbf{U}_n^{(j+1)} \leftarrow [\mathbf{u}_1 \mathbf{u}_2 \ldots \mathbf{u}_{R_n}]$

end for

$\mathcal{Z} \leftarrow \mathcal{A} \bigtimes_{n=1}^{N} \mathbf{U}_n^{(j+1)T}$

$j \leftarrow j + 1$

until $\mathcal{Z}, \{\mathbf{U}_n\}_{n=1}^N$ 收斂

2.4 適合快速影像生成之張量近似演算法

如同先前提到的雙向紋理函數，張量分析時常用於壓縮多維視覺資料並作為影像生成時的模型，而在很多的應用中，影像的生成必須要能達到實時生成 (real-time rendering) 的速度。雖然透過張量近似已經可以對大量的多維資料達到相當程度的壓縮，但要達到實時生成的目標，影像重建時所需的運算量通常還是太大了。為了降低運算量，最直觀的方法便是將維度縮減的幅度增加，但這樣的方式也相對提高了近似的誤差。因此，[6] 及 [7] 分別提出了分群張量近似演算法 (clustered tensor approximation, CTA) 和 K 分群張量近似演算法 (K-clustered tensor approximation)，將張量沿著其中一個模分成幾個群集 (cluster) 後再進行近似，如此一來在生成影像時便可以只對有需要的群集進行重建運算，減少影像生成時的

doi:10.6342/NTU201800667

演算法 2 分群張量近似演算法流程 [6]

給定: $\mathcal{A}, \{R_n\}_{n=1}^{N}, C$

求: $\left\{\mathcal{Z}_c, \{\mathbf{U}_{n,c}\}_{n=1}^{N}\right\}_{c=1}^{C}$

初始化 $\{\mathcal{A}_c\}_{c=1}^{C}$

repeat

for $c \leftarrow 1$ to C **do**

$\mathcal{Z}_c \leftarrow ALS(\mathcal{A}_c)$ 所得核心向量

$\{\mathbf{U}_{n,c}\}_{n=1}^{N} \leftarrow ALS(\mathcal{A}_c)$ 所得基底矩陣

$\mathbf{V}_{m,c} \leftarrow (uf_m(\mathcal{Z}_c)uf_m(\mathcal{Z}_c)^T)^{-\frac{1}{2}}uf_m(\mathcal{Z}_c)$

end for

for $i \leftarrow 1$ to I_m **do**

for $c \leftarrow 1$ to C **do**

$\mathcal{A}_{m_i,c} \leftarrow \mathcal{A}_{m_i} \bigtimes_{\substack{n=1 \\ n\neq m}}^{N} \mathbf{U}_{n,c}^T$

$\mathbf{W}_{m_i,c} \leftarrow uf_m(\mathcal{A}_{m_i,c})\mathbf{V}_{m,c}^T$

end for

$\mathcal{A}_{m_i}$ 分配至群集 $\arg\max_c \|\mathbf{W}_{m_i,c}\|_F^2$

end for

until $\{\mathcal{A}_{m_i}\}_{i=1}^{I_m}$ 皆分配到原群集

運算成本。

2.4.1 分群張量近似 (Clustered Tensor Approximation)

分群張量近似演算法的演算法如演算法 2所示，主要分成三個階段，初始化階段、近似階段以及分群階段。在初始化階段，先將目標張量 $\mathcal{A} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$ 沿著欲分群的模 m 切為 I_m 個子張量 (sub-tensor)$\mathcal{A}_{m_i} \in \mathbb{R}^{I_1 \times \ldots \times I_{m-1} \times 1 \times I_{m+1} \times \ldots \times I_N}$ ，其中 $\mathcal{A}_{m_i}$ 表示沿模 m 所得到的第 i 的子張量。接著給定一個初始的分群，將子張量分配到 C 個群集中，並將群集中的子張量組合為 $\mathcal{A}_c \in \mathbb{R}^{I_1 \times \ldots \times I_{m-1} \times I_{m,c} \times I_{m+1} \times \ldots \times I_N}$ ，其中 $I_{m,c}$ 為分配到群集 c 中的子張量個數。

在近似階段中，會以 2.3節所提到的交替最小平方法對每個群集張量進行近似，並分解得 $\mathcal{Z}_c \in \mathbb{R}^{R_1 \times R_2 \times \ldots \times R_N}$ 以及 $\mathbf{U}_{1,c} \in \mathbb{R}^{I_1 \times R_1}$，$\mathbf{U}_{2,c} \in \mathbb{R}^{I_2 \times R_2}$，……，$\mathbf{U}_{m,c} \in \mathbb{R}^{I_{m,c} \times R_m}$，……，$\mathbf{U}_{N,c} \in \mathbb{R}^{I_N \times R_N}$。然而，透過這樣的近似每個子張量只能用所屬群集分解後的 R_m 個基底進行重建，在相近壓縮率下為原本交替最小平方

doi:10.6342/NTU201800667

法縮減後基底數的 $1/C$，因此重建時的誤差便會增加。為了使降低這樣的誤差，必須將各個子張量分配到它們最能被完整的重建的群集中。在分群階段中，各個子張量將依據重建誤差重新被分配到各個群集中，透過重複近似與分群的階段，來達到區域最佳解 (local optimum)。

為了找到重建誤差最小的群集，在分群階段我們首先對每個群集分別計算：

$$\mathbf{V}_{m,c} = (uf_m(\mathcal{Z}_c)uf_m(\mathcal{Z}_c)^T)^{-\frac{1}{2}}uf_m(\mathcal{Z}_c) \tag{2.12}$$

因為 $uf_m(\mathcal{Z}_c)$ 的列向量具有正交的性質，式 2.12可以理解為對 $uf_m(\mathcal{Z}_c)$ 列向量的正規化 (normalize)，接著對每個子張量便可以計算：

$$\begin{aligned} \mathcal{A}_{m_i,c} &= \mathcal{A}_{m_i} \bigtimes_{\substack{n=1 \\ n\neq m}}^{N} \mathbf{U}_{n,c}^T \\ \mathbf{W}_{m_i,c} &= uf_m(\mathcal{A}_{m_i,c})\mathbf{V}_{m,c}^T \end{aligned} \tag{2.13}$$

得到的 $\mathbf{W}_{m_i,c} \in \mathbb{R}^{R_m}$ 代表的是子張量 $\mathcal{A}_{m_i}$ 投射到群集 c 基底張量的比重，比重越大，代表子張量越能完整的投射到 c 的基底張量，利用這些基底進行重建時便能夠有更少的近似誤差。因此，我們在分群階段會將子張量分配到投射比重最大的群集：

$$\arg\max_c \|\mathbf{W}_{m_i,c}\|_F^2 \tag{2.14}$$

所有的子張量都被分配到新的群集後，新的群集將在近似階段中重新計算出新的核心張量即基底矩陣，整個流程將會重複直到分群的結果不再變動為止。

2.4.2 K 分群張量近似 (K-Clustered Tensor Approximation)

分群張量近似透過將張量分配到多個小群集中以減少影像生成時的計算成本。然而因為每個子張量只能使用一個群集的基底進行重建，儘管透過重複分配達到了區域最佳解，仍然遺失了其他群集中相關的資訊。此外，分群張量近似的近似結果受到初始分群相當大的影響，在沒有適當的初始分群下，便會造成更多的資訊遺失。K 分群張量近似演算法透過將子張量分配到多於一個的群集中，可以更充分的利用到不同群集中的相關資訊，也可以減少初始分群對結果的影響。

K 分群張量近似的流程如演算法 3所示。如同分群張量近似，K 分群張量近似一樣擁有初始化、近似和分群三個階段，其中初始化階段與分群張量近似一樣必須先進行一次初始分群以及初始的近似以得到各個群集的核心張量和基底矩

doi:10.6342/NTU201800667

演算法 3 K 分群張量近似演算法流程 [7]

給定: $\mathcal{A}, \{R_n\}_{n=1}^{N}, C, K_m$

求: $\left\{\mathcal{Z}_c, \{\mathbf{U}_{n,c}\}_{n=1}^{N}\right\}_{c=1}^{C}$

初始化 $\left\{\mathcal{Z}_c, \{\mathbf{U}_{n,c}\}_{n=1}^{N}\right\}_{c=1}^{C}$

repeat

for $c \leftarrow 1$ to C **do**

計算 $\mathbf{V}_{m,c}$

令 $\mathbf{U}_{m,c}$ 的所有元素為零

end for

for $i \leftarrow 1$ to I_m **do**

利用式 2.13和式 2.14決定 c_{i_1}

利用式 2.15更新 $(\mathbf{U}_{m,c_{i_1}})_{i*}$

for $k \leftarrow 2$ to K_m **do**

利用式 2.16決定 c_{i_k}

利用式 2.20更新 $\left\{(\mathbf{U}_{m,c_{i_j}})_{i*}\right\}_{j=1}^{k}$

end for

end for

for $c \leftarrow 1$ to C **do**

$\mathbf{U}'_{m,c}, \mathbf{Y}_c \leftarrow SVD(\mathbf{U}_{m,c})$

$\mathbf{U}_{m,c} \leftarrow \mathbf{U}'_{m,c}$

$\mathcal{Z}_c \leftarrow \mathcal{Z}_c \times_m \mathbf{Y}_c$

end for

for $c \leftarrow 1$ to C **do**

以交替最小平方法解 2.23得 $\mathcal{Z}_c, \{\mathbf{U}_{n,c}\}_{n=1}^{N}$

$\mathbf{U}_{m,c} \leftarrow \mathbf{M}_c \mathbf{U}_{m,c}$

end for

until $\sum_{c=1}^{C} \|\mathcal{Z}_c\|_F$ 收斂

doi:10.6342/NTU201800667

陣。接著進入分群階段，在這個階段會將每個子張量 $\mathcal{A}_{m_i}$ 分配到 K_m 個不同的群集。第一個群集，記作 c_{i_1}，可以透過與分群張量近似一樣的式 2.13和式 2.14決定，然後可以計算出利用群集 c_{i_1} 重建出 $\mathcal{A}_{m_i}$ 所需的係數，也就是群集 c_{i_1} 於模 m 的基底矩陣中，對應到子張量 $\mathcal{A}_{m_i}$ 的列向量：

$$(\mathbf{U}_{m,c_{i_1}})_{i*} = uf_m(\mathcal{A}_{m_i,c_{i_1}})uf_m(\mathcal{Z}_{c_{i_1}})^+ \tag{2.15}$$

其中”+”代表的是 Moore-Penrose 偽逆矩陣 (Moore-Penrose pseudoinverse)，而 $\mathcal{A}_{m_i,c_{i_1}}$ 則與在式 2.13中有相同的定義。決定了分配到的第一個群集後，其他的每個群集則必須根據已經選擇的群集來決定。在為子群集 $\mathcal{A}_{m_i}$ 決定要分配的第 k 個群集時，我們要找的是可以最完整近似前面的群集無法近似的殘差 (residual) 的群集，而殘差最小化的問題則等同於 [7]：

$$\max_{c_{i_k}} \left\| uf_m(\mathcal{A}_{m_i,c_{i_k}})\mathbf{V}_{m,c_{i_k}}^T - \sum_{j=1}^{k-1} uf_m\left(\mathcal{Z}_{c_{i_j},c_{i_k}} \times_m (\mathbf{U}_{m,c_{i_j}})_{i*}\right)\mathbf{V}_{m,c_{i_k}}^T \right\|_F^2 \tag{2.16}$$

$$\text{subject to} c_{i_k} \in \{1,2,3,...,C\}, c_{i_k} \notin \{c_{i_1}, c_{i_2}, ..., c_{i_{k-1}}\}$$

其中，

$$\mathcal{Z}_{c_{i_j},c_{i_k}} = \mathcal{Z}_{c_{i_j}} \bigtimes_{\substack{n=1 \\ n\neq m}}^{N} \mathbf{U}_{c_{i_k}}^T \mathbf{U}_{c_{i_j}} \tag{2.17}$$

可以看到，式 2.16中的第一項與式 2.14是一樣的目標函式，用來選取出較能完整重建出子張量的群集。然而，若僅以此為選取群集的標準，選取到的群集可能會與前面已經被選取的群集較為類似，沒辦法提供額外資訊來改善重建的效果。也就是說，我們要找的是一方面能完整的近似子張量 $\mathcal{A}_{m_i}$，另一方面與其他被分配到的群集相關性較小的群集。式 2.16中後面一項便可以理解為一個懲罰函式 (penalty function)，用來排除掉與前面已選擇的群集擁有太大相關性的群集。

在每次選擇了新的群集後，前面計算的重建係數必須要進行更新。首先計算出下列矩陣：

$$\mathbf{Z}_{m_i}^{(k)} = \begin{bmatrix} uf_m\left(\mathcal{Z}_{c_{i_1},c_{i_1}}\right)uf_m(\mathcal{Z}_{c_{i_1}})^T & \cdots & uf_m\left(\mathcal{Z}_{c_{i_1},c_{i_k}}\right)uf_m(\mathcal{Z}_{c_{i_k}})^T \\ \vdots & \ddots & \vdots \\ uf_m\left(\mathcal{Z}_{c_{i_k},c_{i_1}}\right)uf_m(\mathcal{Z}_{c_{i_1}})^T & \cdots & uf_m\left(\mathcal{Z}_{c_{i_k},c_{i_k}}\right)uf_m(\mathcal{Z}_{c_{i_k}})^T \end{bmatrix} \tag{2.18}$$

$$\mathbf{a}_{m_i}^{(k)} = \left[uf_m\left(\mathcal{A}_{m_i,c_{i_1}}\right)uf_m(\mathcal{Z}_{c_{i_1}})^T \quad \cdots \quad uf_m\left(\mathcal{A}_{m_i,c_{i_k}}\right)uf_m(\mathcal{Z}_{c_{i_k}})^T \right]^T \tag{2.19}$$

doi:10.6342/NTU201800667

接著，更新後的重建係數便可以下列式子計算出來：

$$\mathbf{u}_{m_i}^{(k)} = \mathbf{Z}_{m_i}^{(k)^+}\mathbf{a}_{m_i}^{(k)} \tag{2.20}$$

其中，

$$\mathbf{u}_{m_i}^{(k)} = \left[\left(\mathbf{U}_{m,c_{i_1}}\right)_{i*} \quad \cdots \quad \left(\mathbf{U}_{m,c_{i_k}}\right)_{i*} \right]^T \tag{2.21}$$

最後，在所有的子張量都分配結束後，由於前面的方法分別為每個子張量更新了重建係數，每個群集中模 m 的基底矩陣行向量並不具有正交的特性。因此必須利用奇異值分解進行後續處理來讓基底矩陣保有正交的性質。對基底矩陣 $\mathbf{U}_{m,c}$ 進行奇異值分解，可以得到

$$\mathbf{U}_{m,c} = \mathbf{U}'_{m,c}\mathbf{Y}_c \tag{2.22}$$

其中 $\mathbf{U}'_{m,c} \in \mathbb{R}^{I_m \times R_m}$ 便是分解後得到的行向量基底矩陣中主要 R_n 個奇異向量，$\mathbf{Y}_c$ 則是奇異值對角矩陣和轉置後列向量基底矩陣的乘積。接著可以用 $\mathbf{U}'_{m,c}$ 取代原本的基底矩陣，且為了不影響前面得到的結果，利用 $\mathbf{Y}_c$ 將核心張量更新為 $\mathcal{Z}_c \times_m \mathbf{Y}_c$。

在分群階段完成所有子張量的分群後，在近似階段會根據分群的結果重新進行張量近似，以得到新的核心張量和基底矩陣。在 K 分群張量近似中，因為每個子張量都必須由多個群集進行重建，必須要透過迭代的方式一個一個的對群集進行近似。在第 c 個迭代中，只會對群集 c 的核心張量和基底矩陣進行更新，其他群集則保持不變，目標函式為：

$$\min_{\mathcal{Z}_c, \{\mathbf{U}_n\}_{n=1}^N} \left\| \mathcal{A} - \sum_{\substack{j=1 \\ j \neq c}}^{C} \left(\mathcal{Z}_j \bigtimes_{n=1}^{N} \mathbf{U}_{n,j} \right) - \mathcal{Z}_c \bigtimes_{n=1}^{N} \mathbf{U}_{n,c} \right\|_F^2 \tag{2.23}$$

式中的前兩項是目標張量扣除其餘群集的重建結果後所得到的殘差，而式 2.23便代表群集 c 將以此殘差為近似目標。要得到對殘差近似的解，可以直接使用交替最小平方法來近似殘差。要注意的是，在這次的迭代中，只有被分配到群集 c 的子張量要被近似，因此需要先從殘差中取出屬於群集 c 的子張量。我們可以定義一個群集 c 成員子張量的索引集合：

$$M_c = \{i \in \{1, 2, ..., I_m\} | \mathcal{A}_{m_i} \text{ is a member of cluster } c\} \tag{2.24}$$

doi:10.6342/NTU201800667

並定義群集 c 的成員矩陣 $\mathbf{M}_c \in \mathbb{R}^{I_m \times I_{m,c}}$ 其元素為：

$$(\mathbf{M}_c)_{i_1 i_2} = \begin{cases} 1 & \text{if } i_1 = (M_c)_{i_2} \\ 0 & \text{otherwise} \end{cases} \tag{2.25}$$

透過將殘差與 $\mathbf{M}_c^T$ 沿模 m 進行模 n 乘法的方式便可以將群集 c 的成員子張量的殘差取出，接著只要對此結果張量進行近似即可。此外，近似後得到的模 m 基底矩陣 $\mathbf{U}_{m,c}$ 也必須更新為 $\mathbf{M}_c\mathbf{U}_{m,c}$ 以供後續計算殘差時使用。分群階段與近似階段將重複直到各個群集的 Frobenius 範數 (Frobenius norm) 的加總值收斂為止。

2.5 實驗結果

在本章中，我們將利用雙向紋理函數的資料組來展示交替最小平方法、分群張量近似和 K 分群張量近似的近似效果並加以分析比較，也會探討各項參數對於近似結果的影響。我們利用光追蹤軟體 (ray tracer)POV-Ray 生成一組測試資料來進行測試，另外也測試了於 [8] 中提取出的雙向紋理函數資料組。

2.5.1 利用光線追蹤軟體生成測試資料

光線追蹤是電腦圖學 (computer graphics) 中一種用來生成三維場景圖像的演算法，利用逆向追蹤到達人眼的光線取代由光源開始追蹤，可以有效率的生成出高品質的圖像。而我們使用的 POV-Ray 便是一個使用光線追蹤演算法來生成圖像的開源軟體，透過場景描述語言 (scene description language) 來描寫場景中的物件、材質和光照環境，也可以調整虛擬相機的各項參數來達到不同的成像效果。

為了生成雙向紋理函數的資料組，我們在場景中設置一個具有特定紋理的三維物件，並將虛擬相機以及光源於一個半球面上移動，在不同光照角度下以不同視角對物件進行取樣。光源和相機在取樣半球面上的移動在球坐標系的 θ 和 ϕ 方向都是以 30 度為間距，總共會有 25 個不同的視角以及 25 個光照的角度。圖 2.5為生成的影像中 5 個不同光照角度與 5 個不同視角的影像。從圖中可以看到，置於傾斜視角的虛擬相機所觀測到的會是傾斜的平面影像，這樣的影像中影像像素並不會對應到特定的紋理位置，沒辦法直接作為雙向紋理函數的資料組。為了將像素位置對應到固定的紋理位置，我們必須將得到的影像進行校正 (rectification)。首先我們使用 POV-Ray 生成另外一組影像，將原本的三維物件替

doi:10.6342/NTU201800667

換為一個黑白相間的棋盤格並從與之前相同的視角觀測，生成的影像如圖 2.6所示。接著，可以用 Matlab 中內建的函數 *detectCheckerboardPoints* 來取得每一個影像中棋盤格格點的位置。已知一個傾斜視角影像中的棋盤格點位置以及正面視角影像中棋盤對應格點的位置，就可以再利用 Matlab 中的 *fitgeotrans* 函數來估算出兩個影像中棋盤格平面之間的單應性 (homography) 矩陣。得到兩個平面間的單應性矩陣後就可以將與該棋盤格於同一個傾斜視角擷取到的紋理影像轉換為正面視角，轉換後的影像如圖 2.7所示。轉換後的影像中同樣的像素位置將會對應到同樣的紋理位置，我們便可以用這些影像建構張量並進行近似。

2.5.2 合成資料組近似結果

在這一節中我們將對上一節中所生成的資料組 *Cobblestone* 進行張量近似。因為這組資料的形狀與結構較明顯，在視覺上較容易判斷差異，我們用這組資料來展示張量近似對不同模進行維度縮減的效果。*Cobblestone* 中每一張影像的解析度為 480×640，具有 25 個不同的視角及不同的光照角度，可以組成一個 $480 \times 640 \times 25 \times 25$ 的張量，我們使用交替最小平方法來進行近似，並對各個不同的模進行維度縮減，其中三原色 (RGB) 的資料將分開近似後再重新組合為影像。此外我們也用主成分分析對資料組進行壓縮量相同的維度縮減作為比較。近似後重建出的影像如圖 2.8所示。這裡的壓縮率以原始資料的資料量，與維度縮減後資料量的比值計算。以圖中的結果為例，原始資料量為 $480 \times 640 \times 25 \times 25$。主成分分析將資料排列成一個 $(480 \times 640) \times (25 \times 25)$ 的矩陣，只保留 25 個基底向量的情況下資料量為 $(480 \times 640) \times 25$，可得到 25 倍的壓縮率。而張量近似將資料組成一個 $480 \times 640 \times 25 \times 25$ 的張量，在光照角度模保留所有基底向量，視角模僅保留 1 個基底向量的情況下 (圖 2.8c)，壓縮後會得到一個 $480 \times 640 \times 25 \times 1$ 的核心張量，壓縮率也是 25 倍。從圖中可以看出，在相同壓縮率的情況下，張量近似可以透過對張量的不同模進行維度縮減來保留需要的資訊。在 2.8c中我們保留了所有光照角度模的基底向量，在視角模則只保留一個基底向量，因此重建出來的影像保留了光影的效果，石塊的輪廓則較為模糊。相反的，在 2.8b保留了所有的視角模基底向量和一個光照角度模的基底向量，因此石塊的輪廓便完整的被保留了下來，光影的效果則不明顯。儘管主成分分析在這個例子中擁有較小的均方根誤差 (root-mean-square error)，但因為沒有辦法控制欲保留的資訊，視覺上的效果並不一定比較好。此外，在二維矩陣的主成分分析中，只要截去對應到較小奇異值的基底向量即可以得到最佳化的解，相比於交替最小平方法並不能保證得到最

doi:10.6342/NTU201800667

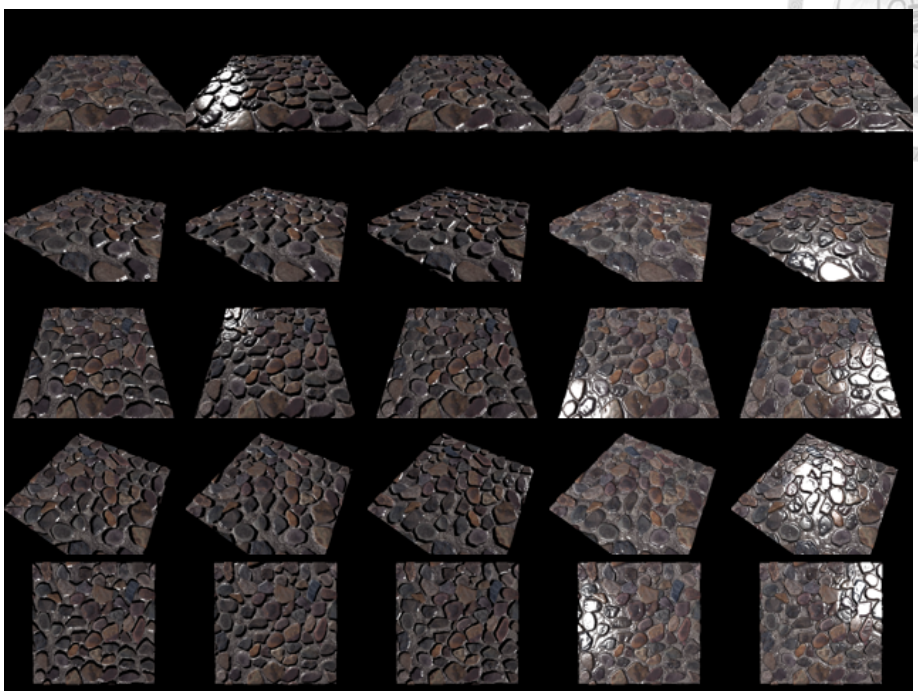


圖 2.5: 利用 POV-Ray 生成三維物件 [9] 於不同光照角度從不同視角觀測之影像

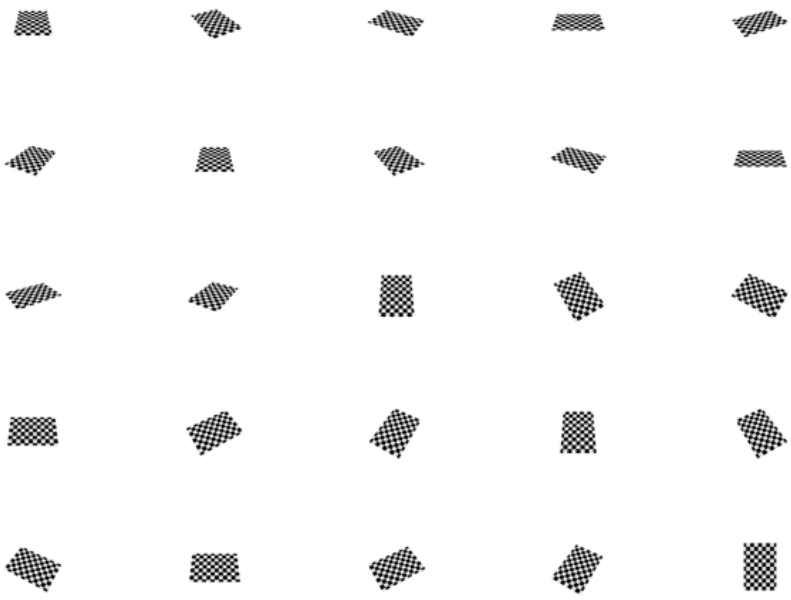


圖 2.6: 利用 POV-Ray 生成棋盤格於不同視角觀測之影像

doi:10.6342/NTU201800667

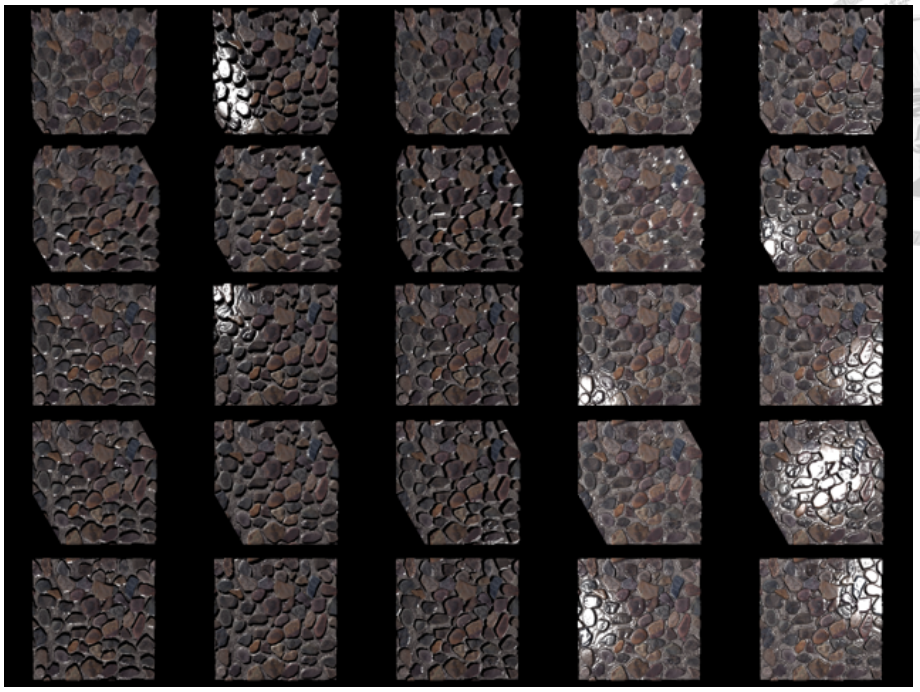


圖 2.7: 利用棋盤格估計單應性矩陣後轉換為正面之紋理影像

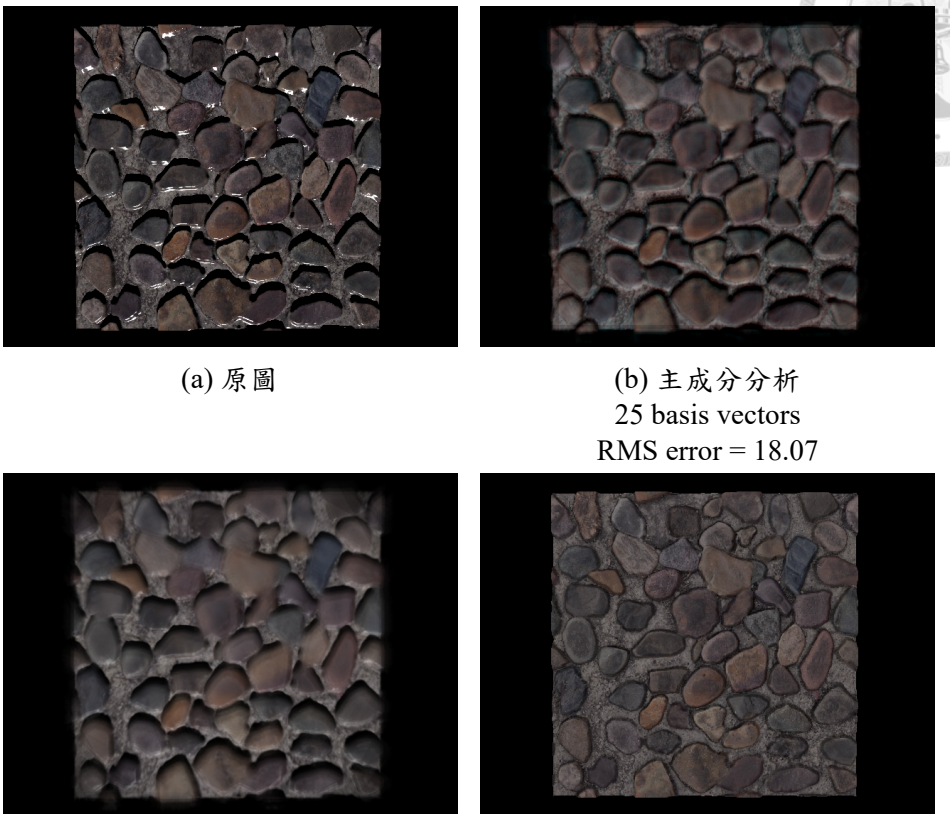
佳解，在未對影像的行向量與列向量兩個模進行縮減的情況下，主成分分析擁有較小的誤差是可以預期的。

主成分分析中將影像展開為向量使資料組得以排列為矩陣的方法雖然讓其可以用簡單的截切達到最佳解，卻也使得影像中相鄰像素的相關性資訊無法被利用。若是將影像維持其二維的型態組成張量進行近似，便可以對影像的行向量與列向量模也進行維度縮減，保留更多的視角模與光照角度模基底向量，如此一來便可以在相同壓縮率的情況下達到比主成分分析更小的誤差。圖 2.9是交替最小平方法於行向量和列向量模也進行維度縮減後與主成分分析在相同壓縮率的重建影像比較，2.9a是主成分分析保留了 100 個基底向量進行重建的結果。2.9b則是交替最小平方法的重建結果，在行向量模保留 300 個基底向量，列向量保留 400 個基底向量，視角模和光照角度模則各自保留 16 個基底向量。由前述的方法計算，兩者的壓縮率皆為 6.25。從圖中可以看得出來在相同壓縮率下，交替最小平方法有較小的均方根誤差，視覺上也有較好的重建效果。

2.5.3 公開資料組近似結果

除了上述自行生成的資料組，我們也使用了 [8] 所提供的公開資料組來做測試。這些資料組有較多不同的視角和不同的光照角度，在這一節中我們會使用這些資料組來比較交替最小平方法、分群張量近似和 K 分群張量近似的效果差異。

doi:10.6342/NTU201800667



(a) 原圖

(b) 主成分分析
25 basis vectors
RMS error = 18.07

(c) 交替最小平方法
1 view basis vector, 25 illumination basis vectors
RMS error = 20.33

(d) 交替最小平方法
25 view basis vectors, 1 illumination basis vector
RMS error = 23.35

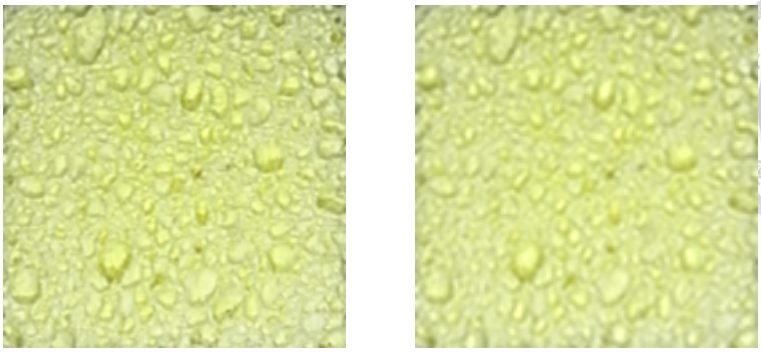
圖 2.8: 張量近似與主成分分析用於 *Cobblestone* 後之重建影像

(a) 主成分分析
RMS error = 12.89

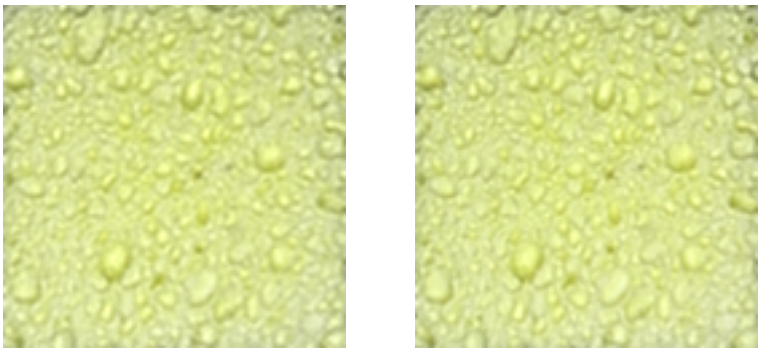
(b) 交替最小平方法
RMS error = 9.89

圖 2.9: 張量近似於行向量與列向量模維度縮減後與主成分分析之比較

doi:10.6342/NTU201800667



(a) 原圖　　(b) 交替最小平方法



(c) 分群張量近似　　(d) K 分群張量近似

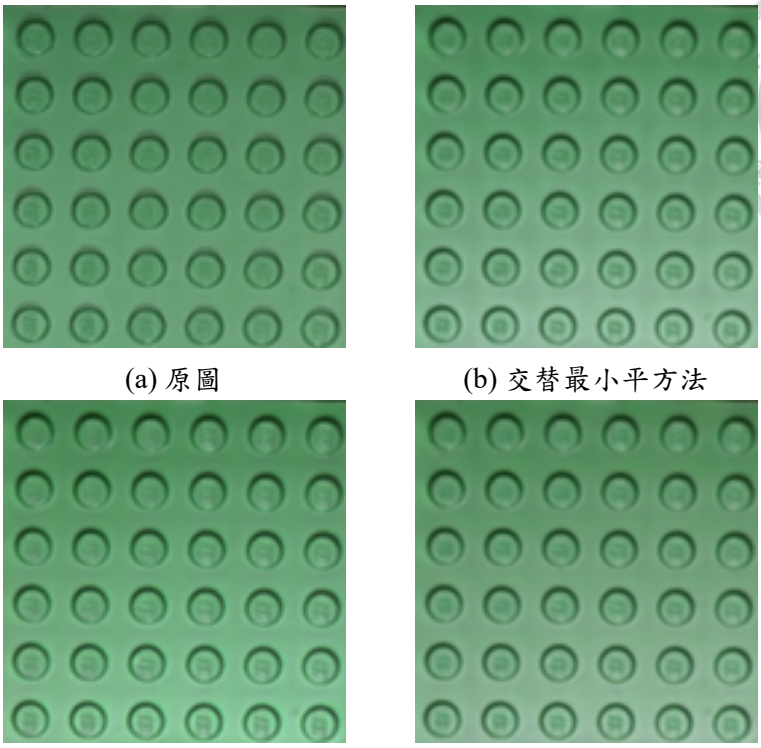
圖 2.10: *Sponge* 資料組重建結果

我們使用了 *Sponge* 和 *Lego* 兩組資料，它們之中每張影像的解析度為 128×128，由 90 個不同的視角於 120 個不同的光照角度進行取樣。近似後重建的影像分別如圖 2.10和 2.11所示，而其所使用的參數則列於表 2.1中。

在此次實驗中分群張量近似與 K 分群張量近似都是將視角模分為 5 個群集，因此當每個群集於視角模縮減後的秩為 4 時，會與交替最小平方法在視角模縮減後的秩為 20 時有相當的壓縮率。在重建品質上，分群張量近似中因為每個子張量只能使用 4 個基底進行重建，有較大的重建誤差，而交替最小平方法與 K 分群張量近似則有著差不多的表現。值得注意的是，K 分群張量近似甚至有誤差較低的情況。這部分可能的解釋是因為 K 分群張量近似演算法在近似階段中，每個群集是以其他群集重建後與目標張量的殘差為近似目標，儘管只能使用較少的基底來重建子張量，但近似得到的基底卻能夠更有效的逼近目標張量，也因此能達到更好的重建效果。至於在運算時間上，K 分群張量近似因為其運算較為複雜且收斂也較緩慢，花了比分群張量近似和交替最小平方法多相當多的時間，其運算時間甚至達到交替最小平方法的一百倍以上，但也以此換取了重建的品質以及應用時的影像生成速度。

最後，我們比較了 K 分群張量近似中子張量可以分到的群集個數 K 以及群集數 C 對結果的影響，其結果如表 2.2所示。從表中可以看出，在群集數 C 固定的

doi:10.6342/NTU201800667



(a) 原圖　(b) 交替最小平方法

(c) 分群張量近似　(d) K 分群張量近似

圖 2.11: *Lego* 資料組重建結果

情況下，子張量分配到的群集數 K 越大重建後的誤差越小，這是因為未能完整重建的部分可以由較多的群集來補足。然而當 K 值越來越高，其增加所能減少的誤差也越來越少，因為較後面所選到的群集也是較無法重建出子張量的群集，因此即使被用來補足前面群集無法重建的部分，效果也比較有限。另外在群集個數方面，首先注意到當群集個數改變時，視角模縮減後的秩也做了相應的改變，因此實驗的各組數據是在壓縮率差不多的情況下得到的。而也因為縮減後的秩減少了，每個群集所保留的基底數也會比較少，使得群集完整重建子張量的能力減弱，誤差也就因此跟著提高，這點在 K 值為 1，也就等同於分群張量近似，的時候特別的明顯。在 K 值大於 1 的情況中則因為有了其他群集幫助重建，群集個數對誤差的影響也就較為不顯著。總結來說，在壓縮率保持固定的情況下，群集數越少，重建出來的效果也會越好，然而也會使得群集進行維度縮減後的秩較大，重建時的運算量也較多，會拖慢應用時的影像生成速度。對分 K 分群張量近似來說群集數增加所導致的重建誤差增加較不顯著，且可以利用增加 K 值來增進重建的效果，然而 K 值所提供的重建品質也會逐漸減少，且 K 值的增加也同樣會增加重建時的運算量，進而影響影像生成的速度。

doi:10.6342/NTU201800667

	Sponge			*Lego*		
Parameter & Result	ALS	CTA	KCTA	ALS	CTA	KCTA
$I_x \times I_y \times I_v \times I_l$	$128 \times 128 \times 90 \times 120$					
$R_x \times R_y \times R_l$	$80 \times 80 \times 16$					
R_v(clustered mode)	20	4	4	20	4	4
C	1	5	5	1	5	5
K	1	1	3	1	1	3
RMS error	8.88	10.16	8.53	16.39	19.19	16
Compression time(min.)	0.71	13.93	73.61	5.53	15.04	81.93

表 2.1: 用各個演算法近似公開資料組的參數與結果

	RMS error		
K	$C = 6(R_v = 3)$	$C = 5(R_v = 4)$	$C = 4(R_v = 5)$
1	19.13	18.26	17.6
2	17.19	16.47	16.17
3	16.72	15.97	15.83
4	16.46	15.74	-
5	16.35	-	-

表 2.2: K 分群張量近似不同參數設定對結果之影響

doi:10.6342/NTU201800667

Chapter 3　分群張量近似之硬體架構設計

在前面的章節中我們介紹了利用交替最小平方演算法來達成張量近似的方法，以及適合應用於快速影像生成的分群張量近似和 K 分群張量近似。然而這些近似方式主要應用於資料量相當龐大的多維視覺資料，當需要處理的資料越來越龐大，近似的過程也會變得相當冗長。尤其在分群張量近似與 K 分群張量近似的演算法中，需要重複進行多次的近似，運算時間又會變得更久。考量到張量的運算方式類似於矩陣運算，我們嘗試設計一個硬體架構，利用硬體平行運算的能力達到加速的效果。其中，我們選擇可以平行運算各個群集近似過程，因此可擴充性 (scalability) 更好的分群張量近似演算法為目標來進行設計。

3.1　奇異值分解之硬體實作演算法

在進行張量近似的過程中，奇異值分解是用來分析資料的主要工具，在我們的硬體架構中是相當重要的一塊。在開始說明硬體的架構及各個模組前，我們將在這一節先對我們所使用的奇異值分解演算法做介紹。因為其廣泛的應用領域以及複雜的運算過程，用硬體實作奇異值分解的研究並不少見。其中最典型的方法之一便是雙邊 Jacobi 旋轉 (two-sided Jacobi rotation)，透過對矩陣中對角線上任意兩個元素所對應到的 2×2 矩陣進行旋轉，將整個矩陣作對角化 (diagonalize)，配合上脈動陣列 (systolic array) 的架構可以達到高度的平行化，大幅提升運算的的效率。然而這樣的架構也需要使用大量的運算單元來達到平行化的運算，通常用於較小的矩陣，可擴充性較差，且只能用於方陣，並不適合張量近似中，對大尺寸長方形矩陣做奇異值分解的需求。因此，我們採用了較有彈性的 Hestenes-Jacobi 演算法來實作硬體中的奇異值分解運算。

在 Hestenes-Jacobi 演算法中，不同於雙邊 Jacobi 旋轉演算法直接利用消除非對角線元素來達到對角化的方法，是以迭代的方式將矩陣中任意兩個行向量正交化來達到奇異值分解。當目標矩陣透過乘上一個正交矩陣來達到行向量的正交化，表示為:

$$\mathbf{AV} = \mathbf{B} \tag{3.1}$$

doi:10.6342/NTU201800667

其中 **B** 的行向量具有正交的性質。接著將 **B** 正規化:

$$\mathbf{B} = \mathbf{B}\mathbf{\Sigma}^{-1}\mathbf{\Sigma} \tag{3.2}$$

$\mathbf{\Sigma}$ 是一個對角矩陣，其對角線上的元素是 **B** 中行向量的 Frobenius 範數。如果令 $\mathbf{U} = \mathbf{B}\mathbf{\Sigma}^{-1}$，式 3.1就可以改寫為:

$$\mathbf{AV} = \mathbf{U}\mathbf{\Sigma} \longleftrightarrow \mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^{T} \tag{3.3}$$

也就是奇異值分解的形式。在實作上，我們使用了 [10] 中所提出的方法，完整流程如演算法 4所示。在一開始，需要將目標矩陣轉置後再與原本的目標矩陣相乘，計算出一個共變異數 (covariance) 矩陣。共變異數矩陣中的非對角線元素包含了原本目標矩陣中任意兩個行向量的共變異數，也就是內積值，對角線上則包含了各個行向量的 Frobenius 範數平方。在迭代的階段，會重複的對每一個由任意兩行向量組成的行向量對進行 Jacobi 旋轉將其正交化。對於一對行向量，首先會從共變異數矩陣中取出他們的共變異數以及各自的範數平方值，利用這些值算出需要旋轉角度的正弦 (sine) 和餘弦 (cosine) 值。有了正弦與餘弦值，便可以組成旋轉矩陣對行向量進行旋轉。因為旋轉的運算也會影響到之前的旋轉結果，使之前已達到正交化的兩個行向量又失去正交的性質，因此透過多次的迭代來逐漸的讓每個行向量對都具有正交的性質。將目標矩陣的行向量正交化後所得到的矩陣，如同上式中的 B，其行向量的 Frobenius 範數便是目標矩陣的奇異值。然而在這裡並不會直接對目標矩陣的行向量進行旋轉，因為在迭代的過程中並不需要知道行向量的完整內容，只需要知道旋轉後行向量之間的共變異數和範數平方便可以繼續後續的迭代。而替代的方法是利用計算出來的旋轉參數，直接對共變異數矩陣中的共變異數和範數平方進行更新，如此一來便可節省掉對行向量進行旋轉並重新計算共變異數與範數平方的運算時間。另外，除了在每次迭代中所選取的兩個行向量之間的共變異數需要做更新，這兩個行向量與其他行向量之間的共變異數也會因旋轉而改變。演算法 4中於每次迭代的最後的三個 for 迴圈便是在對與其他行向量的共變異數值做更新。整個迭代會持續進行直到任意兩個行向量之間的共變異數都足夠小，也就是兩行向量足夠接近正交關係，或是達到設定的迭代次數為止。使用這個方法的另一個優點是，由於共變異數矩陣會是一個對稱矩陣 (symmetric matrix)，在儲存上可以用上三角矩陣或下三角矩陣的型式做儲存，僅需要儲存其

doi:10.6342/NTU201800667

演算法 4 Hestenes-Jacobi 演算法

給定: A

求: 由 **A** 的奇異值組成之向量 **Sig**

D $\leftarrow \mathbf{A}^T\mathbf{A}$

repeat

for $i \leftarrow 1$ to $NumberofColumn - 1$ **do**

for $j \leftarrow i$ to $NumberofColumn$ **do**

$norm_1 \leftarrow \mathbf{D}_{i,i}$, $norm_2 \leftarrow \mathbf{D}_{j,j}$

$cov \leftarrow \mathbf{D}_{i,j}$

$\rho \leftarrow (norm_2 - norm_1)/(2cov)$

$t \leftarrow sign(\rho)/(|\rho| + \sqrt{1+\rho^2})$

$cos \leftarrow 1/\sqrt{1+t^2}$

$sin \leftarrow cos * t$

$\mathbf{D}_{i,i} \leftarrow \mathbf{D}_{i,i} - t * cov$

$\mathbf{D}_{j,j} \leftarrow \mathbf{D}_{j,j} + t * cov$

$\mathbf{D}_{i,j} \leftarrow 0$

for $k \leftarrow 1$ to $i - 1$ **do**

$\mathbf{D}_{k,i} \leftarrow \mathbf{D}_{k,i} * cos - \mathbf{D}_{k,j} * sin$

$\mathbf{D}_{k,j} \leftarrow \mathbf{D}_{k,i} * sin + \mathbf{D}_{k,j} * cos$

end for

for $k \leftarrow i + 1$ to $j - 1$ **do**

$\mathbf{D}_{i,k} \leftarrow \mathbf{D}_{i,k} * cos - \mathbf{D}_{k,j} * sin$

$\mathbf{D}_{k,j} \leftarrow \mathbf{D}_{i,k} * sin + \mathbf{D}_{k,j} * cos$

end for

for $k \leftarrow j + 1$ to $NumberofRow$ **do**

$\mathbf{D}_{i,k} \leftarrow \mathbf{D}_{i,k} * cos - \mathbf{D}_{j,k} * sin$

$\mathbf{D}_{j,k} \leftarrow \mathbf{D}_{i,k} * sin + \mathbf{D}_{j,k} * cos$

end for

end for

end for

until 收斂

Sig $\leftarrow \sqrt{diag(\mathbf{D})}$

doi:10.6342/NTU201800667

一半的資料量即可，很適合用於我們硬體內部記憶體大小較有限的情況。

為了將上述演算法應用於張量近似中，必須根據我們的需要將演算法做一些更動。首先，在張量近似演算法中要進行奇異值分解的是一個張量展開後所得到的矩陣，其行向量數目遠大於列向量的數目。若依照上述演算法，會得到一個相當大的共變異數矩陣，造成儲存上的困難，因此我們先將展開後得到的矩陣進行轉置，如此一來共變異數矩陣便可以縮小到能夠存入內部記憶體的大小。另外，在張量近似中我們所需要的並不是奇異值，而是行向量基底矩陣，這部分必須另行計算。在前面的說明中有提到，行向量基底矩陣 **U** 是由經過正交化的矩陣 **B** 再經過正規化得到的，而從式子中則可以看出列向量基底矩陣 **V** 就是將 **A** 進行正交化的正交矩陣。雖然我們需要的是張量展開矩陣的行向量矩陣，但因為如前面所說，我們會將展開的張量先行轉置再進行 Hestenes-Jacobi 演算法，轉置後的矩陣之奇異值分解可以表示為:

$$\mathbf{A}^T = \left(\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T\right)^T = \mathbf{V}\mathbf{\Sigma}^T\mathbf{U}^T \tag{3.4}$$

因此，我們需要的張量展開矩陣的行向量基底矩陣，實際上便是用於將轉置後的張量展開矩陣的行向量進行正交化的正交矩陣，也就是在上述演算法中所有旋轉矩陣的乘積。所以在實際運算時，將會需要將每個迭代中算出的旋轉矩陣進行相乘，在迭代結束時便可得到需要的行向量基底矩陣。最後，透過這個演算法所得到的奇異值並非由大到小排列，所對應到的奇異向量也是一樣，因此在最後也必須另外做排序，如此才可以在截切時選取到主要的奇異向量。

3.2 整體架構

我們提出的硬體整體架構如圖 3.1所示。為了提高運算過程的精準度，我們使用了 IEEE 754 的單精度浮點數資料格式來儲存資料，每一筆資料將佔據 32 個位元，所有資料的運算也將以浮點數的格式進行。浮點數的使用再加上原本已經相當龐大的資料量，要將這樣大小的資料存於晶片中並不實際，因此在這個架構中，只有在運算中需要頻繁取用或是改寫的資料會被存在晶片內部的記憶體中。整個分群張量近似的演算法中包含三種主要的運算，張量對矩陣的模 n 乘法、奇異值分解，以及子張量投射到各群集核心張量的比重計算。其中，我們會將與張量進行模 n 乘法的基底矩陣、奇異值分解中的共變異數矩陣，以及進行分群時子

doi:10.6342/NTU201800667

張量投射到個群集核心張量的比重存於晶片中的靜態隨機存取記憶體 (SRAM) 中。為了能夠平行化的使用存於記憶體中的資料，在晶片中我們使用了十塊靜態隨機存取記憶體來儲存這些資料，以得到較高的存取頻寬。其他資料如輸入張量，各個群集的核心張量及基底矩陣等，則將存放在外部記憶體中，待需要時再做取用。

在與外部系統的溝通方面，我們使用 64 位元的輸入及輸出傳輸寬度，每次可以傳輸兩筆資料。在對大量的資料作運算時，系統的頻寬往往是影響系統速度的重要因素，然而在我們的架構中，輸入資料往往需要與記憶體中的資料一齊使用，因此雖然可以使用更大的傳輸寬度，但若靜態隨機存取記憶體的頻寬無法配合，也無法提升系統的效能。另外，因為在沿不同模的模 n 乘法中，會需要沿著不同的模提取模 n 向量來與矩陣進行運算，所以張量資料的存取並不在連續的位址上，為此必須要透過另外的訊號來告訴外部系統存取的位址，以我們預計處理的資料量，四階張量且每一維度可達到 128 筆資料的大小，需要用 28 個位元來表示欲存取的位址。最後，由於多種不同用途資料都被存放外部記憶體中，也需要有另外的訊號來告知外部系統輸入或輸出的資料類型，目前正在處理的群集和沿著哪一個模進行存取也都有個別的訊號來表示。加上時鐘 (clock) 和重置 (reset) 等基本控制訊號，我們共使用 67 個輸入腳位以及 135 個輸出腳位。各個輸入及輸出訊號的寬度及用途如表 3.1所示。

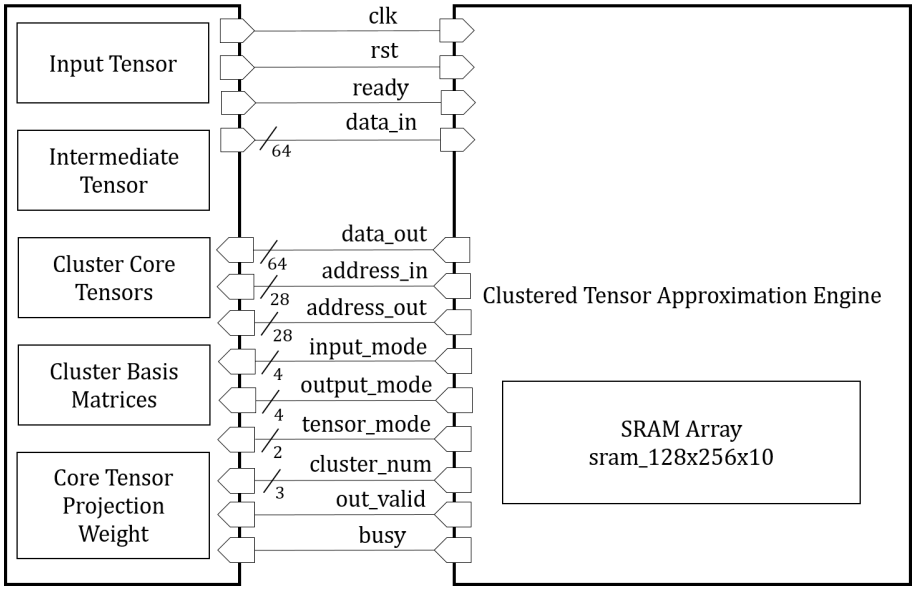


圖 3.1: 系統整體架構圖

doi:10.6342/NTU201800667

Signal	I/O	Bit Width	Description
clk	input	1	Clock signal
rst	input	1	Active low asynchronous reset signal
ready	input	1	Rises when system is ready to start
data_in	input	64	Input data
data_out	output	64	Output data
address_in	output	28	Indicate the address to read input data
address_out	output	28	Indicate the address to write output data
input_mode	output	4	Indicate the type of input data
output_mode	output	4	Indicate the type of output data
tensor_mode	output	2	Indicate the mode of tensor operation
cluster_num	output	3	Indicate the cluster in process
out_valid	output	1	Rises when the output data is valid
busy	output	1	Rises when system is busy

表 3.1: 輸出輸入信號一覽表

3.3 各電路模組

在這一節中，我們將會介紹內部各個模組的功能以及運作方式。圖 3.2展示了此硬體架構的內部模組以及它們之間的溝通訊號。主要的模組包括掌管全域變數並負責控制整個演算法流程的控制器模組 (controller)、計算對外讀取資料及寫出資料位址的位址計算器模組 (address calculator)、計算模 n 乘法的張量與矩陣乘法器模組 (Tensor-matrix multiplier)、負責奇異值分解中共變異數矩陣的計算，以及分群階段中各個子張量對群集核心張量投射比重的共變異數矩陣計算模組 (covariance matrix calculator)，與在奇異值分解演算法中計算 Jacobi 旋轉並更新共變異數矩陣的 Jacobi 旋轉處理器模組 (Jacobi rotation processor)。此外，張量與矩陣乘法器模組、共變異數矩陣計算模組和 Jacobi 旋轉處理器模組都將在執行其運算時使用由十塊靜態隨機存取記憶體所組成的靜態隨機存取記憶體陣列 (SRAM array) 來儲存需要使用到的資料並使用浮點數運算單元陣列 (floating point operator array) 中的浮點數運算單元來進行相關的運算。最後我們也設置了一個由 32×48 個暫存器所組成的緩衝區 (buffer) 作為運算時資料的暫存空間。

doi:10.6342/NTU201800667

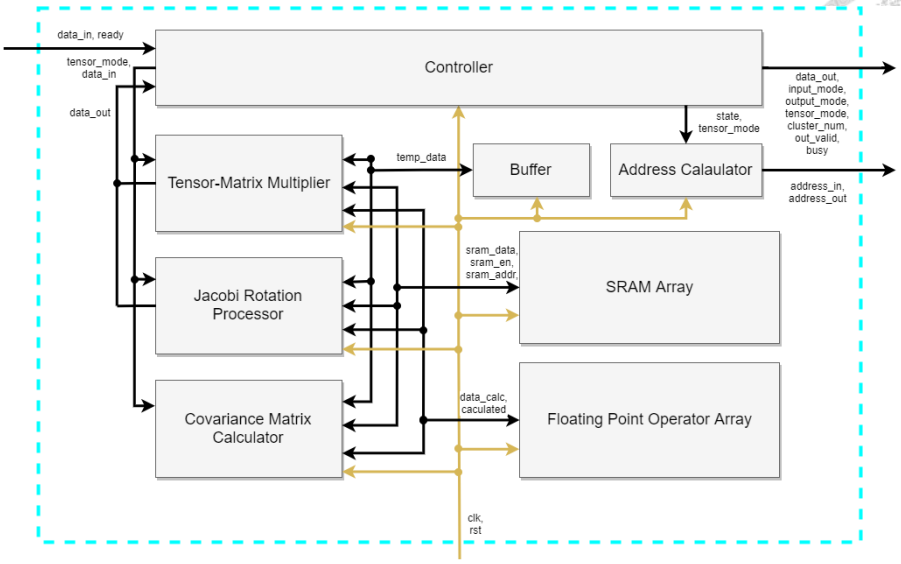


圖 3.2: 內部模組架構圖

3.3.1 控制器模組

控制器模組主要是由一個有限狀態機 (finite state machine) 構成，有限狀態機可以利用狀態來紀錄系統目前處於整個流程的哪一個部分，根據不同的狀態來做出相對應的動作，而狀態也會在特定條件達成時進行切換。此外，控制器也負責控管全域變數，在此系統中，全域變數包括如迭代的次數，張量運算的模，和正在處理中的群集等。

在我們的有限狀態機中共定義了 14 個狀態，各個狀態之間的順序及切換條件如圖 3.3所示，透過這樣流程完成分群張量近似的演算法，演算法的詳細流程可參照第 2.3節與第 2.4.1節。一開始，在外部系統尚未以 ready 訊號告知開始前，系統會處於閒置狀態 (IDLE)。在外部系統告知開始後，會進入群集資訊狀態 (CLUSTER_INFO) 讀取將要處理的群集所包含的子張量索引及群集的大小。接著，透過矩陣讀取狀態 (MATRIX_IN)、張量讀取狀態 (TENSOR_IN) 和張量輸出狀態 (TENSOR_OUT) 三個狀態可以完成一次張量對矩陣的模 n 乘法，三個狀態會重複使模 n 乘法運算可以依序沿著不同的模進行。完成各個模的乘法後，便會在共變異數矩陣計算狀態 (COV_CAL)、Jacobi 參數計算狀態 (JACOBI_PARA)、奇異向量矩陣更新狀態 (JACOBI_UPD_U) 以及三個負責共變異數矩陣更新的狀態 (JACOBI_UPD_COV_1、JACOBI_UPD_COV_2、JACOBI_UPD_COV_3) 中完成奇異值分解，其中，共變異數矩陣計算狀態負責計算初始的共變異數矩陣，而其他的五個狀態將執行迭代直到奇異值分解完成。

對一個模的最佳化流程由沿本身之外其他模的模 n 乘法，以及奇異值分解所

doi:10.6342/NTU201800667

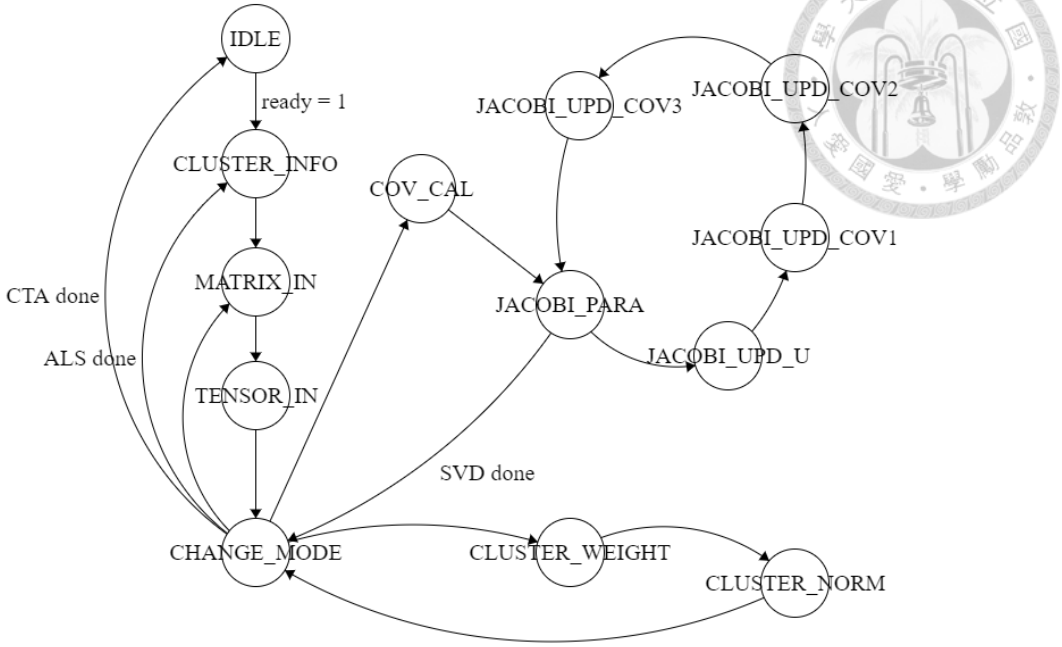


圖 3.3: 控制器有限狀態機

組成。此流程依序對各個模執行一遍後便完成了交替最小平方法中的一次迭代，根據使用軟體實驗的經驗，我們設定將執行五次迭代，如此一來便算是對一個群集完成了近似。上述交替最小平方法流程將依序用於各個群集的近似，對每個群集都完成近似後就會進入到分群階段。在分群階段中，會先回矩陣讀取狀態開始模 n 乘法，將所有子張量乘上群集中除了分群的模以外的所有基底矩陣，接著群集投射比重計算狀態 (CLUSTER_WEIGHT) 會計算出子張量投射至當前群集的核心張量得到的比重，群集範數計算狀態 (CLUSTER_NORM) 則會接著算出集比重的 Frobenius 範數，並根據當前的 Frobenius 範數最大值，將子張量重新分配至新的群集。在對每一個群集進行過投射比重的 Frobenius 範數計算與比較後，就可以得到這次迭代的分群結果。近似階段和分群階段會重複直到每個子張量都不再被分配到新的群集中為止，並回到閒置狀態等待下一次的使用。

最後，由於整個演算法是由一層層的迴圈組成，一些狀態根據全域變數所記錄的值可能會有多種切換的選擇，為了簡化各個狀態之間的關聯，我們另外設定了一個切換狀態 (CHANGE_MODE) 在一個完整的運算結束後統一判斷接下來該切換的狀態並更改全域變數。

doi:10.6342/NTU201800667

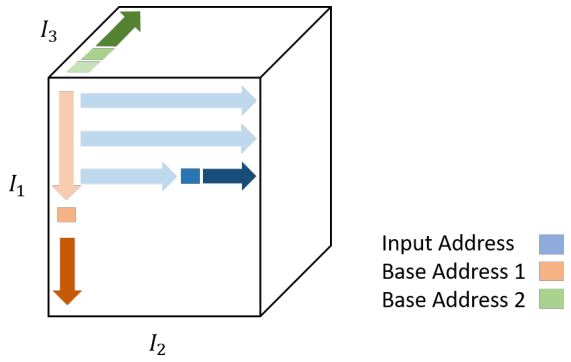


圖 3.4: 基準位址用於三階張量之模 2 乘法運算示意圖

3.3.2 位址計算器模組

位址計算器模組的功能是計算欲讀取或寫入的外部記憶體位址。進行張量對矩陣的模 n 乘法時，我們在做法上會將張量沿模 n 將張量展開為矩陣再進行矩陣乘法。展開後的矩陣是以模 n 向量作為該矩陣的行向量，也就是說，讀入矩陣行向量時，向量的元素不一定會在記憶體中連續的位址上。因此，我們必須在每次對外部記憶體讀寫資料時提供欲讀寫的位址。一般來說，若要計算出多維陣列中特定位址的線性排列位址，需要將各個維度的索引值減 1 再乘上較小的維度的維數 (dimensionality) 乘積，再將其加總。舉例來說，一個 $I_1 \times I_2 \times I_3$ 的三階張量中的 $(3, 4, 5)$ 位址轉換成以 0 為起始的線性位址時需要計算 $(3-1)+(4-1)\times I_1+(5-1)\times I_1 \times I_2$。這樣的做法會使得計算的時間路徑過長，沒辦法在高速的時脈下運行。再加上，因為使用了 28 個位元來表示位址，將需要使用 28 位元的乘法器來完成上述運算，佔據較多的硬體資源。為盡量避免乘法器的使用，我們使用基準位址來記錄各個模 n 向量接下來的起始位址。為了計算四階張量的線性位址，我們記錄了三個基準位址，第一個基準位址代表的是下一個模 n 向量的起始位址，第二個基準位址代表的是第一個基準位址達到該模的維數上限時的下一個起始位址，第三個基準位址則代表第二個基準位址達到該模的維數上限時的下一個起始位址。圖 3.4展示了基準位址在一個三階張量進行模 2 乘法運算時的運作情況。我們以此為例來做說明，其中藍色線段代表的是目前正在讀取的外部記憶體位址，橘色線段為基準位址一，而綠色線段則是基準位址二。而線段中顏色較深的部分代表各個變數接下來將代表的位址，淺的部分代表之前所代表的位址，而介於兩者中間的則是目前所代表的位址。因為進行的是模 2 乘法運算，所以讀取位址會沿著藍色線段讀入資料，以模 2 向量作為乘法時使用的矩陣行向量。當讀取位址到達模 2 的維數上限 I_2，便會以基準位址一作為下個模 2

doi:10.6342/NTU201800667

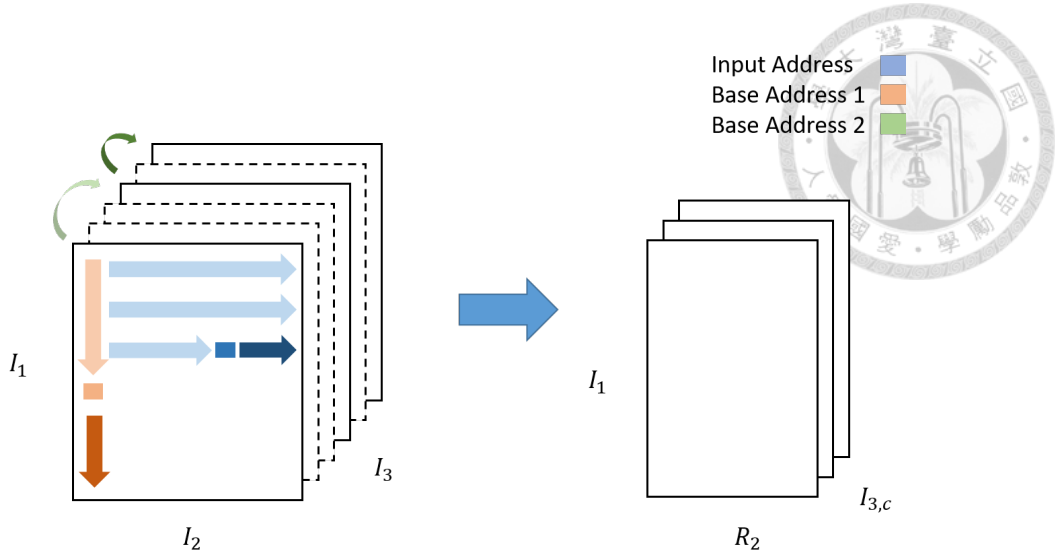


圖 3.5: 讀取群集子張量並於模 n 乘法組合之示意圖

向量的起始位址，而基準位址一也會移動到下一次要使用的起始位址。在這個例子中，由於基準位址一是沿著模 1 向量移動，所以下一個基準位址便是目前的位址加 1。同樣的，當基準位址一到達其維數上限 I_1，便會以基準位址二做為下一次的起始位址，基準位址二也將移動到下一個起始位址，也就是目前的位址加上 $I_1 \times I_2$。只要先定義好基準位址移動時的間距，並根據不同的模選取相應的間距，就可以在避免使用乘法器的情況下完成張量的展開與讀取。而因為計算後的結果在短暫的等候後便會輸出到外部記憶體，會有同時輸入與輸出資料的情況，所以輸出位址將有另一組基準位址來進行計算。

另外，在讀取分群後的群集張量時，因為分配到群集的並不一定是連續的子張量，所以在讀取位址移動時並沒有辦法單純透過加上定義好的間距得到下一個讀取位址。因此在實作上，我們利用張量的模 n 乘法不同模之間可交換的特性 (式 2.2)，首先對非分群模進行模 n 乘法，如此一來便可以在讀取位址沿著其他的模移動時，先行將沿著分群模的基準位址以該模的間距累加至下一個需要的起始位址。而在輸出結果時分群模的基準位址將直接以連續的方式做移動，也就是在進行模 n 乘法的同時，將原本張量中分散的子張量組合起來，以便後續的使用。如圖 3.5所示。

最後，位址計算器模組也會在讀取基底矩陣以及更新奇異向量時負責計算讀寫的位址，基底矩陣的讀取因為是以連續的位址將整個矩陣讀取進來，做法上相當直觀。至於更新奇異向量的讀寫則必須將奇異向量矩陣的任意兩個行向量讀入進行更新。一般來說需要利用乘法器算出特定行向量的起始位址，但我們無論正在進行哪一個模的高階奇異值分解，都固定使用同一個大小為 128×128 的矩陣來

doi:10.6342/NTU201800667

儲存奇異向量矩陣，因此只須將欲存取的行向量索引向左移位 7 個位元便可得到行向量的起始位址，簡化了這部分位址的運算。

3.3.3 靜態隨機記憶體陣列與緩衝區

在前面已提到，因為資料的數量太過龐大，我們必須將大部分的資料儲存在外部記憶體，而在運算過程會頻繁取用或更新的資料則最好存放在晶片內部使其可以快速的存取。為了可以有效率的運用每一次從外部讀入的資訊，我們希望可以一次將每個要與目前讀入的張量資料作運算的矩陣資料從記憶體中讀取出來並做平行化的運算，因此內部記憶體的頻寬將是影響加速相當重要的因素。一般單埠靜態隨機存取記憶體一次可以讀寫記憶體中的一個字元，若使用較多的位元數來儲存一個字元，並用希望一起存取的資料組合成一個字元，便可以一次的讀寫中存取多筆資料，提高記憶體的頻寬。我們使用的製程供應商提供的靜態隨機存取記憶體規格最高可以達到 144 位元一個字元，由於一筆資料是 32 位元，我們選用其倍數 128 位元作為一個字元的寬度，即一個字元中存放 4 筆資料。此外，在一些使用情況中我們也需要快速的存取記憶體中不同字元的資料，因此使用了雙埠靜態隨機存取記憶體，可以一次對記憶體中的兩個字元進行讀寫。然而，這樣的頻寬仍無法達到我們加速的需求，所以我們將資料分開儲存於多個字元量較少的記憶體中，使用上便可以同時對多個記憶體進行讀寫，更進一步提升整體的頻寬。最後，考量要存入內部記憶體的資料量且要將晶片保持在一個合理的面積，我們使用十個字元量為 256，每個字元由 128 位元組成的雙埠靜態隨機存取記憶體來組成一個靜態隨機存取記憶體陣列，作為晶片內部的主要資料存放區域。

除了用靜態隨機記憶體陣列來儲存運算所需的資料，一些運算時的中間產物或是正在等待處理的資料也需要額外的儲存空間，因為這些資料通常較少，可以使用暫存器來儲存，存取上較為快速。我們使用了一個 32×48 的暫存器陣列來組成一個緩衝區，供上述的資料作為暫存用途。靜態隨機記憶體陣列和緩衝區都是由各個運算模組所共用，每個模組使用的方式皆不太相同，在接下來各個模組的介紹中會個別做說明。

3.3.4 浮點數運算單元陣列

在此架構中，我們使用 DesignWare 中所提供的浮點數運算模組來完成需要的浮點數運算。在張量與矩陣的模 n 乘法中，我們會同時對張量展開矩陣中，20 個列的 4 筆資料進行內積運算，每個四點內積運算需要 4 個乘法器與 3 個加法器，

doi:10.6342/NTU201800667

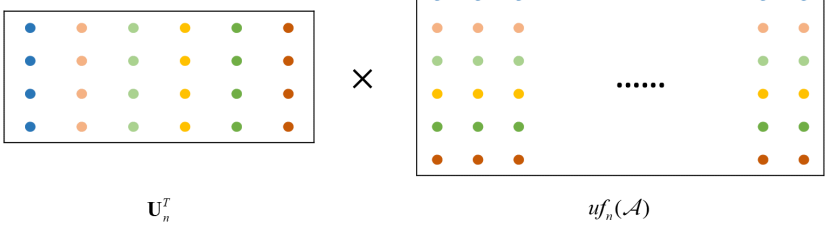


圖 3.6: 張量與矩陣乘法之資料對應示意圖

總共需要 80 個乘法器與 60 個加法器。另外，還需要 20 個加法器來將 20 個列的內積結果進行累加。因此，我們分別使用了 80 個浮點數乘法器和 80 個浮點數加法器來達到張量模 n 乘法運算的平行化。此外也使用了 2 個浮點數開根號運算模組及 2 個浮點數除法器，用於計算奇異值分解中 Jacobi 旋轉所需的參數和分群階段對核心張量投射比重的 Frobenius 範數。上述所提到的浮點數運算單元都具有較長的運算時間，為了能在高速的時脈中使用，我們於其中安插了暫存器並進行時序重置 (retiming) 以達到管線化 (pipeline) 的效果，利用多個時鐘週期來完成所需的運算。

3.3.5 張量與矩陣乘法器模組

張量與矩陣的模 n 乘法在分群張量近似的軟體實作中，是最耗時且使用次數也最多的一種運算，因此也是在此架構中要達到加速的重點。我們所使用的張量資料量相當的龐大，光是將整個張量的資料傳進晶片中，在一個時鐘週期傳入兩筆資料並運作於 500 MHz 時鐘頻率的情況下，便會花費掉軟體運算時間的三分之一左右。也就是說，要達到加速的目的，我們必須將每筆資料的傳入次數控制在三次以內，並在這幾次的傳入就將與其相關的運算完成。我們利用靜態隨機存取記憶體陣列的高頻寬以及浮點數運算單元陣列的平行運算來達到此要求。

為了有效率的利用資料的每一次傳入，我們先檢視每一筆輸入的張量資料所對應到的運算及與之做運算的矩陣資料，圖 3.6中展示了張量沿著模 n 展開後與轉置後的模 n 基底矩陣相乘，也就是模 n 乘法，的資料對應關係。其中，右邊的矩陣為張量沿著模 n 展開所得到的矩陣 $uf_n(\mathcal{A})$，而其中的每一筆資料在整個運算中需要和左邊的轉置基底矩陣 $\mathbf{U}_n^T$ 中所有相同顏色的資料相乘。當 $uf_n(\mathcal{A})$ 中的資料沿著行向量方向被讀入時，我們希望能將讀入的資料與對應到的矩陣行向量中每一筆資料相乘。因此我們在讀入基底矩陣資料時，會將矩陣行向量中的每一筆資

doi:10.6342/NTU201800667

SRAM[0]

Addr	0	1	2	...	127	128	129	...	255
[31:0]	$\mathbf{U}_{1,1}^T$	$\mathbf{U}_{1,2}^T$	$\mathbf{U}_{1,3}^T$	...	$\mathbf{U}_{1,128}^T$	$\mathbf{U}_{41,1}^T$	$\mathbf{U}_{41,2}^T$	...	$\mathbf{U}_{41,128}^T$
[63:32]	$\mathbf{U}_{2,1}^T$	$\mathbf{U}_{2,2}^T$	$\mathbf{U}_{2,3}^T$	...	$\mathbf{U}_{2,128}^T$	$\mathbf{U}_{42,1}^T$	$\mathbf{U}_{42,2}^T$	...	$\mathbf{U}_{42,128}^T$
[95:64]	$\mathbf{U}_{3,1}^T$	$\mathbf{U}_{3,2}^T$	$\mathbf{U}_{3,3}^T$	...	$\mathbf{U}_{3,128}^T$	$\mathbf{U}_{43,1}^T$	$\mathbf{U}_{43,2}^T$	...	$\mathbf{U}_{43,128}^T$
[127:96]	$\mathbf{U}_{4,1}^T$	$\mathbf{U}_{4,2}^T$	$\mathbf{U}_{4,3}^T$	...	$\mathbf{U}_{4,128}^T$	$\mathbf{U}_{44,1}^T$	$\mathbf{U}_{44,2}^T$	...	$\mathbf{U}_{44,128}^T$

SRAM[1]

Addr	0	1	2	...	127	128	129	...	255
[31:0]	$\mathbf{U}_{5,1}^T$	$\mathbf{U}_{5,2}^T$	$\mathbf{U}_{5,3}^T$	...	$\mathbf{U}_{5,128}^T$	$\mathbf{U}_{45,1}^T$	$\mathbf{U}_{45,2}^T$	...	$\mathbf{U}_{45,128}^T$
[63:32]	$\mathbf{U}_{6,1}^T$	$\mathbf{U}_{6,2}^T$	$\mathbf{U}_{6,3}^T$	...	$\mathbf{U}_{6,128}^T$	$\mathbf{U}_{46,1}^T$	$\mathbf{U}_{46,2}^T$	...	$\mathbf{U}_{46,128}^T$
[95:64]	$\mathbf{U}_{7,1}^T$	$\mathbf{U}_{7,2}^T$	$\mathbf{U}_{7,3}^T$	...	$\mathbf{U}_{7,128}^T$	$\mathbf{U}_{47,1}^T$	$\mathbf{U}_{47,2}^T$	...	$\mathbf{U}_{47,128}^T$
[127:96]	$\mathbf{U}_{8,1}^T$	$\mathbf{U}_{8,2}^T$	$\mathbf{U}_{8,3}^T$	...	$\mathbf{U}_{8,128}^T$	$\mathbf{U}_{48,1}^T$	$\mathbf{U}_{48,2}^T$	...	$\mathbf{U}_{48,128}^T$

⋮

SRAM[9]

Addr	0	1	2	...	127	128	129	...	255
[31:0]	$\mathbf{U}_{37,1}^T$	$\mathbf{U}_{37,2}^T$	$\mathbf{U}_{37,3}^T$	...	$\mathbf{U}_{37,128}^T$	$\mathbf{U}_{77,1}^T$	$\mathbf{U}_{77,2}^T$	...	$\mathbf{U}_{77,128}^T$
[63:32]	$\mathbf{U}_{38,1}^T$	$\mathbf{U}_{38,2}^T$	$\mathbf{U}_{38,3}^T$	...	$\mathbf{U}_{38,128}^T$	$\mathbf{U}_{78,1}^T$	$\mathbf{U}_{78,2}^T$	...	$\mathbf{U}_{78,128}^T$
[95:64]	$\mathbf{U}_{39,1}^T$	$\mathbf{U}_{39,2}^T$	$\mathbf{U}_{39,3}^T$	...	$\mathbf{U}_{39,128}^T$	$\mathbf{U}_{79,1}^T$	$\mathbf{U}_{79,2}^T$	...	$\mathbf{U}_{79,128}^T$
[127:96]	$\mathbf{U}_{40,1}^T$	$\mathbf{U}_{40,2}^T$	$\mathbf{U}_{40,3}^T$	...	$\mathbf{U}_{40,128}^T$	$\mathbf{U}_{80,1}^T$	$\mathbf{U}_{80,2}^T$	...	$\mathbf{U}_{80,128}^T$

圖 3.7: 轉置基底矩陣於靜態隨機存取記憶體陣列之資料配置

料存於靜態隨機存取記憶體陣列中同樣的位址中，之後便可以一次將行向量中的資料讀出做運算。圖 3.7是轉置後的基底矩陣於靜態隨機存取記憶體陣列中的資料配置圖，靜態隨機存取記憶體的字元由 128 個位元組 成，所以可以將行向量中連續的四筆資料儲存於一個字元中，十塊靜態隨機存取記憶體便可在一次的讀取中同時讀取行向量中的 40 筆資料。此外，因為我們設定的張量各個模的最大維數是 128 且維度縮減後的秩都將小於 80，轉置基底矩陣的列向量最多只會有 128 筆資料且列向量的數量會小於 80 列，因此位址 0 到位址 127 之間的空間可以用來儲存第 1 到第 40 列的資料，位址 128 以後的空間則用來儲存第 40 到第 80 列的資料。

完成了轉置基底矩陣的讀取和儲存後，控制器模組便會進入讀取張量的狀態，開始將張量資料讀入。前面的段落我們都以讀入一筆資料的情況來做說明，實際上因為輸入資料的腳位有 64 個位元，我們在一個時鐘週期中可以讀進兩筆資料。為了將第二筆資料也充分利用，可以利用雙埠靜態隨機存取記憶體的第二個讀寫埠將下一個行向量也同時讀取出來。從外部讀進來的兩筆張量資料會和從記憶體中讀出的每一列的兩筆資料做內積運算，每一列內積所得的結果會再加上下個週期中讀入的資料與記憶體資料的內積值。如此一直累加下去，到整個模 n 向量被讀入後，就能得到該模 n 向量與轉置基底矩陣的 40 個列向量的內積值，也就是模 n 乘法的結果張量中，對應到的模 n 向量裡面的 40 筆資料。因為靜態隨機存取記憶體陣列一次只能輸出 40 列的資料，若轉置基底矩陣擁有超過 40 個列向量就必須重新讀入一次模 n 向量來做運算，在此情況下每筆張量資料在整個運算中

doi:10.6342/NTU201800667

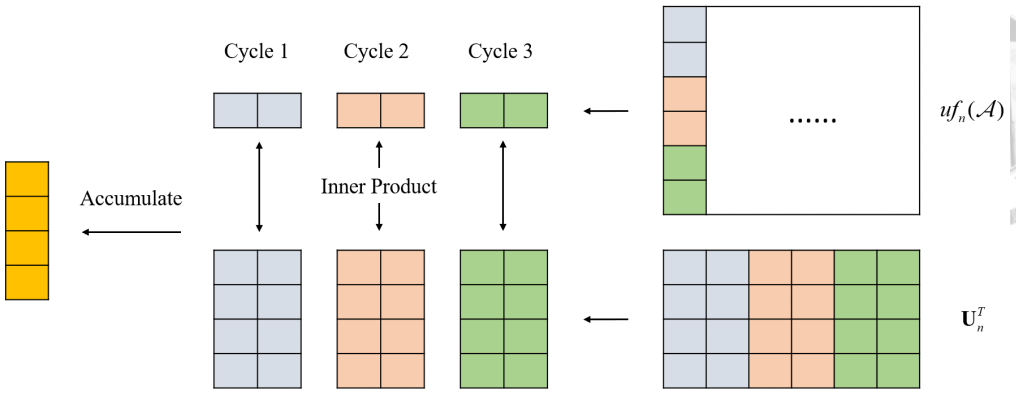
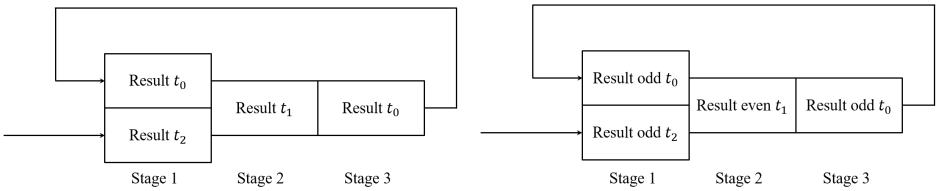


圖 3.8: 模 n 乘法運算示意圖



(a) 奇數與偶數列同時計算　　(b) 奇數與偶數列交錯計算

圖 3.9: 浮點數加法器管線化後用於內積值累加示意圖

會被讀入兩次。圖 3.8是上述運算過程示意圖。

在上述的運算過程中，每個時鐘週期都會從外部記憶體及內部記憶體中讀取需要的資料進行運算，而運算的結果要與之前運算的結果累加起來。然而，在前一小節中曾提到，為了讓浮點數運算單元運作於高速的時鐘頻率，我們對運算單元進行了時序重置與管線化，其中乘法器與加法器都被切為三級，輸入的資料會在兩個時鐘週期後才能得到結果。如此一來在累加各個時鐘週期的內積結果時，便會出現累加的結果來不及加入下一個週期內積結果的情況。如圖 3.9a所示，時鐘週期 t_0 的內積結果輸入到加法器後，要到 t_2 才能得到累加的結果，來不及與 t_1 的內積結果相加，需要用額外的時鐘週期來等待之前的累加結果，也會因此拉長運算時間。為了解決這個問題，我們採用了交錯計算奇數列與偶數列內積值的方式。在每一個時鐘週期中，只計算奇數列或是偶數列的內積值，讓需要累加的資料之間多間隔一個時鐘週期，如此一來前面累加的結果與新的內積結果會在同一個時鐘週期計算完成並進行下一次的累加。圖 3.9b中，時鐘週期 t_0 中僅計算奇數列的內積結果，t_1 中則僅計算偶數列的內積結果，而下一次奇數列的內積結果會在 t_2 時計算出來，剛好可以與前面奇數列的累加結果進行相加。此外，為了不讓運算的速度因為每個週期只計算一半的列向量個數而變慢，在一個週期中會用張量資料中四筆資料和轉置基底矩陣列向量中的四筆資料作內積運算。四筆資料需

doi:10.6342/NTU201800667

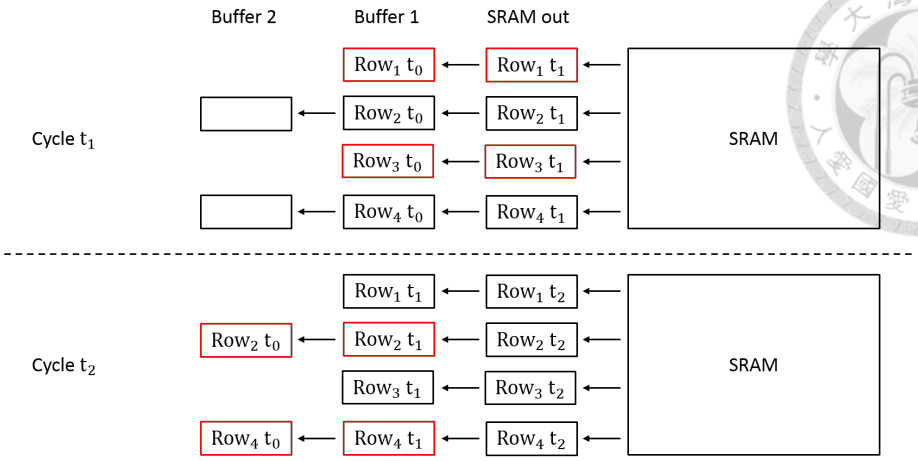


圖 3.10: 交錯計算內積值示意圖

要兩個時鐘週期來讀取，在第三和第四筆資料讀取進來的週期可以開始計算與奇數列的內積值，而讀取下一組資料的前兩筆資料的時鐘週期則可以用來計算與偶數列的內積值。也因此我們需要在資料輸入與靜態隨機存取記憶體的資料輸出腳位使用額外暫存器來將後續時鐘週期會使用到資料做暫存，以配合資料的連續讀取。在資料讀取腳位和靜態隨機存取記憶體的奇數列會在第三與第四筆資料進入的時候連同前一個時鐘週期的兩筆資料做取用，只需要額外使用一組暫存器，靜態隨機存取記憶體的偶數列則因為會再晚一個週期才取用前兩個時鐘週期的四筆資料，需要使用兩組暫存器。圖 3.10是交錯計算奇數與偶數列內積值的時序及暫存器資料示意圖，其中紅色方框內的資料在當個時鐘週期將與時鐘週期 t_0 及 t_1 所讀入的張量資料做內積運算。

最後，當整個模 n 向量都被傳入，會直接再讀取下一個模 n 向量，而當前的模 n 向量將在一段潛時 (latency) 後完成所有內積值的加總並存放於緩衝區中。在進行下一個模 n 向量的運算的同時，緩衝區內的運算結果也會依序輸出到外部記憶體。當所有模 n 向量都完成運算且輸出後，張量與矩陣的模 n 乘法也就完成了。

3.3.6 共變異數矩陣計算模組

共變異數矩陣計算模組在此架構中負責奇異值分解時所需的共變異數矩陣的計算，以及在分群階段將子張量投射到核心張量基底的運算。如在 3.1節所提到的，共變異數矩陣所包含的是張量展開所得的矩陣中任意兩個列向量之間的共變異數值，也就是內積值，而共變異數矩陣也就等同於張量展開所得的矩陣與其轉置矩陣相乘後的結果。另一方面，子張量對核心張量的投射運算則如式 2.13中所示，等同於在與其他模的轉置基底矩陣相乘後的子張量沿分群模展開後，與同樣

doi:10.6342/NTU201800667

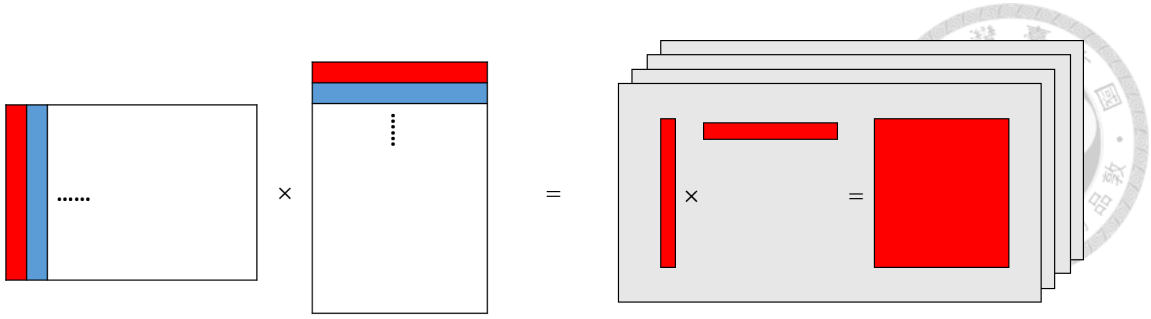


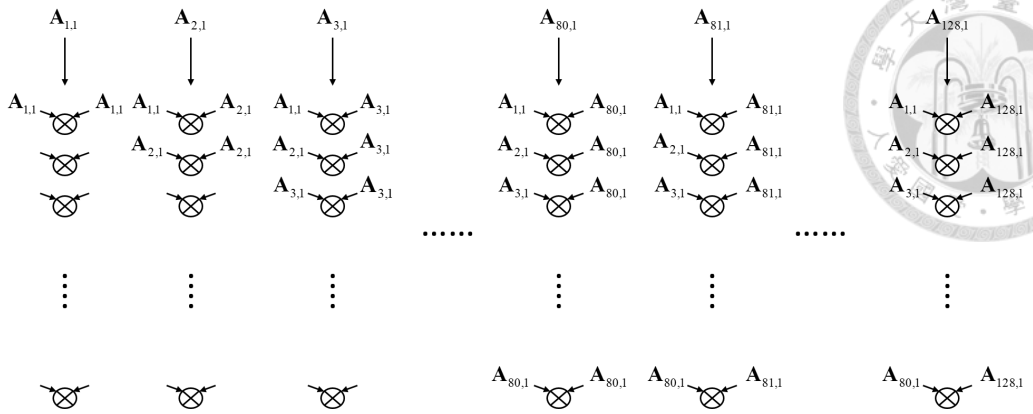
圖 3.11: 矩陣乘積轉換為向量乘積加總示意圖

沿分群模展開並做過正規化的核心張量進行矩陣乘法。以上兩個運算都可以等同於是矩陣的乘法，然而不同於張量對矩陣的模 n 乘法，這裡需要做相乘的兩個矩陣資料都相當龐大，沒辦法沿用模 n 乘法中先將基底矩陣讀入內部記憶體的方式。相對的，這兩個運算所得的結果都是較小的矩陣，且需要於後續的運算中使用，可以存於內部記憶體中。因為這兩個運算的性質相似，都會利用共變異數矩陣計算模組來完成，然而它們之間仍有相異之處，因此模組中也以兩個不同的模式來分別計算。

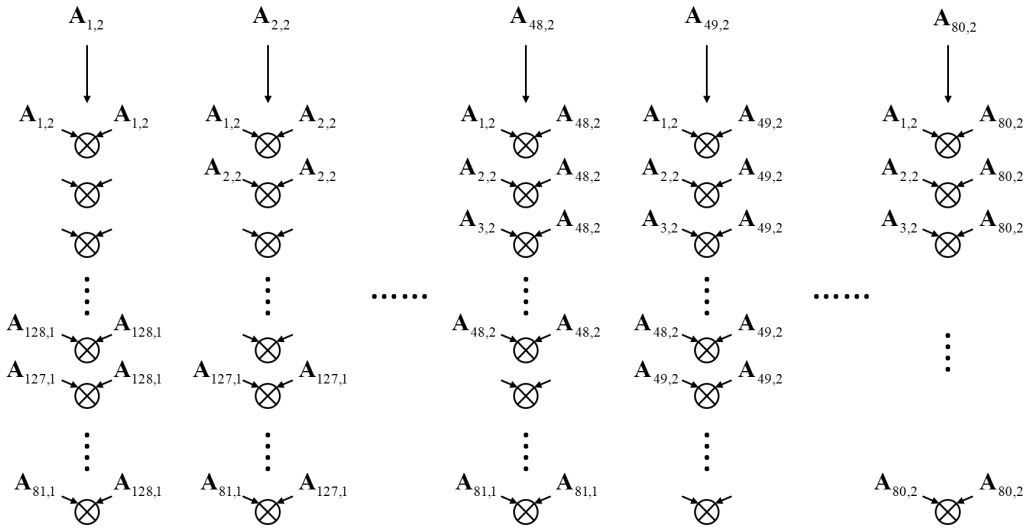
共變異數矩陣之計算

在進行共變異數矩陣的運算時，與模 n 乘法一樣，因為資料量相當大，有效率的運用每一筆傳入的資料是相當重要的。共變異數矩陣計算上有兩個特性。首先，共變異數矩陣是由張量展開所得的矩陣與其轉置矩陣相乘所計算出來的，用於相乘的兩個矩陣具有一樣內容，因此實際上只需傳入一個矩陣的資料。此外，兩個矩陣相乘所得的結果是一個對稱矩陣，除了在儲存上可以上三角矩陣的方式儲存以節省空間，在計算的過程中也只有一半的值需要計算。我們可以利用兩矩陣相乘時相乘結果會等於前者的行向量與後者對應的列向量相乘並累加起來結果的性質，如圖 3.11所示，因為前者的行向量與後者的列向量是相同的，只需要將向量讀入一次便可以計算向量乘積。垂直向量與水平向量相乘所得的矩陣內容是前者向量中元素與後者中任一元素的乘積。在讀入向量時，我們會依序將讀入的資料排列於浮點數乘法器運算單元陣列所有乘法器的一個輸入端，而另一個輸入端則會是該時鐘週期所讀入的資料，等同於將每一個時鐘週期所讀入的資料與所有在前面時鐘週期讀入的資料相乘。然而，在浮點數運算單元陣列中只有 80 個乘法器，張量展開矩陣的行向量大小最多卻會達到 128，因此第 80 個以後的元素在讀入時並沒有辦法馬上與所有已讀入的元素相乘。而在這邊可以注意到，在第 80 個元素以前讀入的元素，在進行運算時並不會使用到所有的乘法器，於是我們將

doi:10.6342/NTU201800667



(a) 向量資料讀入後於乘法器陣列的分配



(b) 利用空閒乘法器完成前個行向量未完成之乘法運算

圖 3.12: 共變異數矩陣運算中乘法器陣列輸入資料示意圖

第 80 個以後的元素，都暫時只與第 80 個以前的元素進行相乘，並將它們存入緩衝區，等到第二個行向量開始讀入，且使用到的乘法器數量還不多的時候，利用空閒的乘法器來完成剩下的乘法運算。圖 3.12為讀入資料後利用乘法器陣列進行乘法運算的示意圖。

行向量元素利用上述方式讀入並且與其他元素相乘後，必須立刻寫入靜態隨機存取記憶體陣列中，讓整資料的讀入與運算可以連續進行。每個時鐘週期中，浮點數乘法器陣列會產生 80 個結果，與靜態隨機存取記憶體陣列一個週期最多可以寫入的資料數相同，可以達成將每一個週期的結果一次寫入記憶體的需求。值得注意的是，雖然靜態隨機存取記憶體陣列的儲存空間足夠存入結果的上三角矩陣，但一個靜態隨機存取記憶體是由 256 個 128 位元的字元組成，以 32 位元的資料來看相當於一個 4×256 的矩陣，10 塊記憶體便是一個 40×256 的矩陣，沒辦法將一個維數為 128 的上三角矩陣以完整的形狀存入，勢必要將其進行切割。因

doi:10.6342/NTU201800667

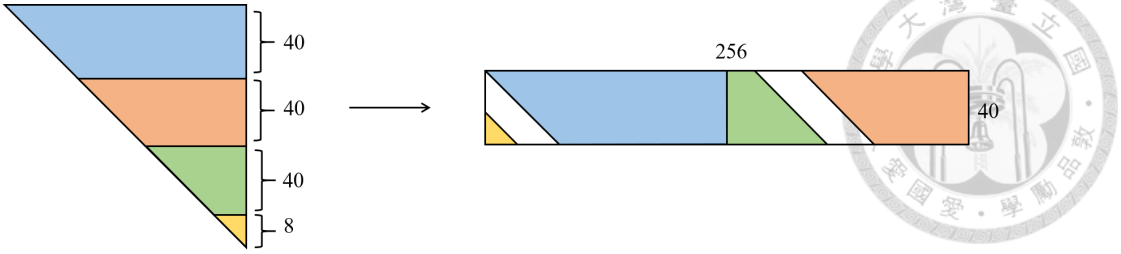


圖 3.13: 共變異數上三角矩陣存於靜態隨機存取記憶體陣列之資料分布

此在寫入資料時，我們利用雙埠隨機存取記憶體的兩個讀寫埠將輸出的 80 筆資料存入記憶體的不同位址中，等於將結果的矩陣的行向量拆開並放置於不同的位址。為了盡量將相近的資料放在鄰近的位置方便後續的存取，行向量中第一筆資料的運算結果，包含前一個行向量未完成的乘法運算結果，將存在位址 0 與位址 128，行向量中第二筆資料的運算結果存在位址 1 與位址 129，以此類推。所有元素的運算結果存入後於靜態隨機存取記憶體陣列中的分布如圖 3.13所示。

在一個行向量全部被讀入並做乘法運算後，下一個行向量便會接著開始傳入，在所有資料都被傳入並計算過後便完成了整個共變異數上三角矩陣的計算。而除了第一個行向量以外，其餘的行向量資料在進行過乘法運算後必須與前面行向量所計算出對應的結果進行加總。然而在寫入時已經佔用了所有的讀寫埠，必須另外穿插一個時鐘週期來讀取之前的運算結果。此外，上述的資料運算方式並沒有辦法在一個時間週期中將讀入的兩筆資料都做運算，因此沒辦法利用輸入腳位一次傳入兩筆資料的優勢。整個運算將花費約整個張量展開矩陣資料量兩倍的時鐘週期來完成。

子張量投射比重之計算

根據第 2.4.1節所介紹的演算法，計算子張量投射比重需要經過三個步驟，首先將要計算比重的群集 c 的核心張量沿分群模展開，並將列向量進行正規化得到 $\mathbf{V}_c$，接著將要分群的子張量與群集中除了分群模外各個模的基底矩陣的轉置矩陣進行模 n 乘法，最後再將子張量沿分群模展開為一個向量並與 $\mathbf{V}_c^T$ 進行乘法運算。雖然在前面的討論中我們以一個子張量來做說明，但因為子張量在進行分群模以外的模 n 乘法時是互相獨立的，實作上可以直接將整個目標張量做模 n 乘法形成一個除分群模外各個模都已經得到縮減的張量，再將其沿分群模展開並與 $\mathbf{V}_c^T$ 行矩陣乘法，

doi:10.6342/NTU201800667

$$\mathbf{B}_c = uf_m(\mathcal{A} \underset{\substack{n=1 \\ n\neq m}}{\overset{N}{\times}} \mathbf{U}_{n,c}^T)$$
$$\mathbf{W}_c = \mathbf{B}_c \mathbf{V}_c^T \tag{3.5}$$

所得到的結果 $\mathbf{W}_c \in \mathbb{R}^{I_m \times R_m}$ 中的第 i 個列向量便是子張量 $\mathcal{A}_{m_i}$ 投射到群集 c 核心張量基底的比重。

與計算共變異數矩陣一樣是矩陣乘法運算，且同樣是無法先將其中一個矩陣存入內部記憶體的情況，我們可以再利用圖 3.11的性質來完成投射比重的計算。但是不同於共變異數矩陣運算時，輸入矩陣的行向量具有較大的維數，$\mathbf{V}^T$ 的列向量維數 R_m 通常小很多，以我們測試時的設定為例，維數僅有 4。因此當乘法中 $\mathbf{B}_c$ 的行向量元素被讀進來時，並不需要與很多的 $\mathbf{V}^T$ 列向量元素做乘法，不會佔用很多的乘法器，資源的分配上可以更彈性，不必像前面的運算中使用兩個時鐘週期來分別進行讀寫，也可以充分的運用一個週期中讀入的兩筆資料。

在這個模式中，會先讀入 $\mathbf{V}^T$ 的一個列向量再接著讀入 $\mathbf{B}_c$ 中對應的行向量，每一筆資料讀入時會依序排列在浮點數運算單元陣列中乘法器的一個輸出端，但並不馬上進行乘法運算。等輸入的資料個數達到 40 個，也就是乘法器個數的一半，接下來的資料會繼續填入後 40 個乘法器的輸入端，而前 40 個乘法器則會開始依序進行與 $\mathbf{V}^T$ 列向量中各個元素的乘法，並將結果寫入靜態隨機存取記憶體陣列中。相反的，接下來當後 40 個乘法器完成資料讀取後，便會開始進行乘法運算，而前 40 個乘法器則開始進行資料讀取，如此交替進行。這樣一來，儘管實際上對靜態隨機存取記憶體的讀寫仍分別使用了一個時鐘週期，但因為資料仍可以持續的讀入到另一個區間的乘法器，並不會拖慢整個運算的進行。此外，資料不用在一讀進來時便做運算，只需要先排放在乘法器的輸入端，一次讀入兩筆資料的設計便可以確實的提升運算的整體速度。最後，儘管用 $\mathbf{V}$ 為符號來做說明，但我們至此尚未對核心張量展開後的矩陣列向量做過正規化。所以我們在讀入其列向量元素時，也會與 $\mathbf{B}_c$ 的行向量一起排列在乘法器陣列的同一個輸入端，如此便能同時計算出核心張量展開後的矩陣中任意兩個列向量之間的內積值，其中也會包含各個列向量與自身的內積值，也就是其 Frobenius 範數的平方，可以直接用於之後比重 Frobenius 範數的計算。圖 3.14是此運算中乘法器與資料分布的示意圖，圖中以 $R_m = 2$ 且前 40 乘法器剛完成資料讀取的兩個時鐘週期為例，可以看到後 40 個乘法器讀入資料時前 40 個乘法器正依序乘上 $\mathbf{V}^T$ 的列項量元素，一次可以讀入兩筆資料，以及將 $\mathbf{V}^T$ 列向量的範數平方一併計算的部分。

doi:10.6342/NTU201800667

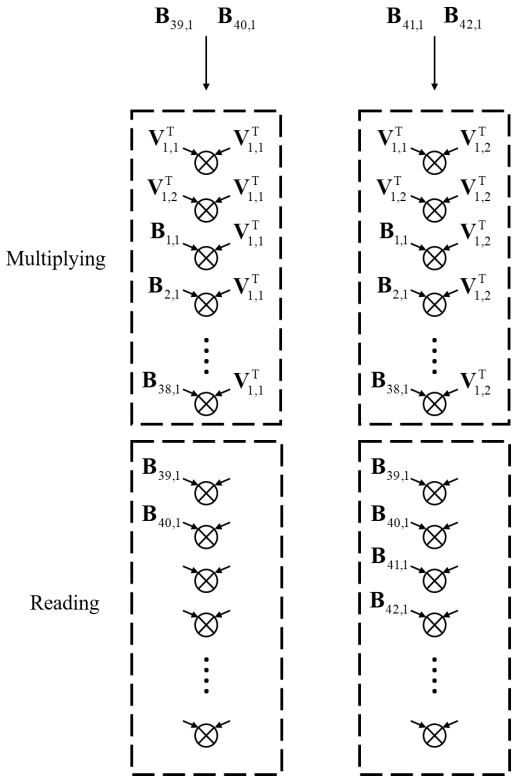


圖 3.14: 子張量投射比重計算之乘法器與資料分布示意圖

3.3.7 Jacobi 旋轉處理器模組

Jacobi 旋轉處理器模組負責奇異值分解中計算出共變異數矩陣後將目標矩陣行向量正交化的部分。如 3.1節所提到的，在這個演算法中我們不直接對行向量進行正交化，而是對共變異數矩陣中的共變異數和範數平方值進行更新。在每次迭代中，Jacobi 旋轉處理器模組會選取一對行向量並進行三個主要功能：利用兩個行向量的共變異數與範數平方值計算出旋轉所需參數、對奇異向量矩陣進行旋轉，以及更新共變異數矩陣。當所有的行向量對都經過旋轉後，即完成一次拂掠 (sweep)，會再重新開始下一次拂掠，直到所有行向量對的共變異數都小於特定閾值 (threshold) 為止。

Jacobi 旋轉參數之計算

在選定一對行向量後，Jacobi 旋轉處理器模組會從靜態隨機存取記憶體陣列中讀取出它們之間的共變異數以及各別的範數平方值，接著以演算法 4中所列的方式計算出用於旋轉及更新共變異數矩陣的參數。其算式可以整理為：

doi:10.6342/NTU201800667

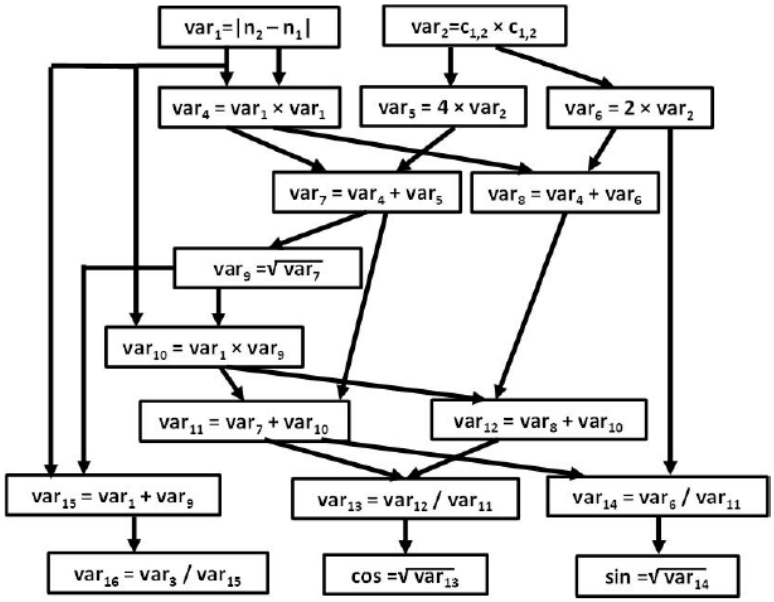


圖 3.15: Jacobi 旋轉參數運算流程 [10]

$$cos = \sqrt{\frac{(n_2-n_1)^2+2c_{1,2}^2+|n_2-n_1|\sqrt{(n_2-n_1)^2+4c_{1,2}^2}}{(n_2-n_1)^2+4c_{1,2}^2+|n_2-n_1|\sqrt{(n_2-n_1)^2+4c_{1,2}^2}}}$$

$$sin = (sign)\sqrt{\frac{2c_{1,2}^2}{(n_2-n_1)^2+4c_{1,2}^2+|n_2-n_1|\sqrt{(n_2-n_1)^2+4c_{1,2}^2}}} \tag{3.6}$$

$$t = (sign)\frac{|2c_{1,2}|}{|n_2-n_1|+\sqrt{(n_2-n_1)^2+4c_{1,2}^2}}$$

其中 $c_{1,2}$ 代表的是兩個行向量間的共變異數，n_1 和 n_2 則分別代表兩個行向量的範數平方值。此外，sin 和 t 的正負號將由 $n_2 - n_1$ 和 $c_{1,2}$ 的正負號決定，當兩者同號時 sin 和 t 為正，反之則為負。整理成上列的形式後，可以將運算過程用浮點數運算單元以如圖 3.15的方式來實現。儘管已經將沒有相依性的變數平行地計算，因為浮點數計算單元耗時較長，這樣串接的方式仍會有 30 個時鐘週期的潛時。為了不浪費等待的時間，在計算參數的同時也會開始讀入接下來要進行更新的奇異向量。

奇異向量矩陣之計算

在張量近似所需要的奇異向量矩陣，如前面所提到的，就是一個可以將目標矩陣的行向量正交化的正交矩陣，所以必須計算每個迭代中將行向量正交化的旋轉矩陣的乘積。在開始進行迭代前，會在外部記憶體初始化一個單位矩陣，在每

doi:10.6342/NTU201800667

$$\begin{bmatrix} 1 & 0 & & & \cdots & & 0 & 0 \\ 0 & 1 & & & \cdots & & 0 & 0 \\ & & \ddots & & & & & \\ & & & cos & & sin & & \\ \vdots & \vdots & & & \ddots & & & \vdots \\ & & & -sin & & cos & & \\ & & & & & & \ddots & \\ 0 & 0 & & & & & & 0 \\ 0 & 0 & & & \cdots & & 0 & 1 \end{bmatrix}$$

圖 3.16: 旋轉矩陣

一次迭代計算出旋轉矩陣後再將其乘上旋轉矩陣。旋轉矩陣的形式如圖 3.16所示，可以看出正如其功能是將兩個行向量正交化，乘上旋轉矩陣後只有其對應到的兩個行向量會受到影響。

因此，在更新奇異向量矩陣時，我們只會將對應到的兩個行向量讀入晶片內並進行旋轉運算：

$$\begin{aligned} \mathbf{v}'_i &= cos \times \mathbf{v}_i - sin \times \mathbf{v}_j \\ \mathbf{v}'_j &= sin \times \mathbf{v}_i + cos \times \mathbf{v}_j \end{aligned} \tag{3.7}$$

迭代中的兩個行向量索引決定了之後，會馬上開始選定的行向量中元素的讀取。因為乘法器數量的限制，我們一次只會讀取兩個行向量中各 40 筆資料進行運算。兩個行向量的 40 筆資料都讀入後會馬上進行更新後行向量資料的計算，並將計算的結果存入緩衝區，利用暫存的方式讓資料的讀取不會中斷，可以直接再讀取接下來的資料。計算完成的結果會在資料讀取持續進行的同時，從緩衝區中依序輸出到外部記憶體。這樣的流程會重複直到將兩個行向量都全部更新完為止。

共變異數矩陣之更新

兩個被選定的行向量之間的共變異數與個別的範數平方的更新相當直觀，只要利用前面計算出的參數 t 及兩向量之間的共變異數就可以根據演算法 4中的方式計算出更新後的範數平方值，共變異數值則直接更新為正交化後應有的內積值 0。因為只要更新三個值，耗時相當少。而另一方面，兩個行向量與其他行向量之間的共變異數值的更新所要更新的值更多，加上共變異數於靜態隨機存取記憶體陣列中的擺放方式影響，更新的範圍較為複雜也更加耗時。若在一次迭代中，被選定的兩行向量索引分別為 i 和 j，共變異數矩陣中需要更新的資料分布如圖 3.17所

doi:10.6342/NTU201800667

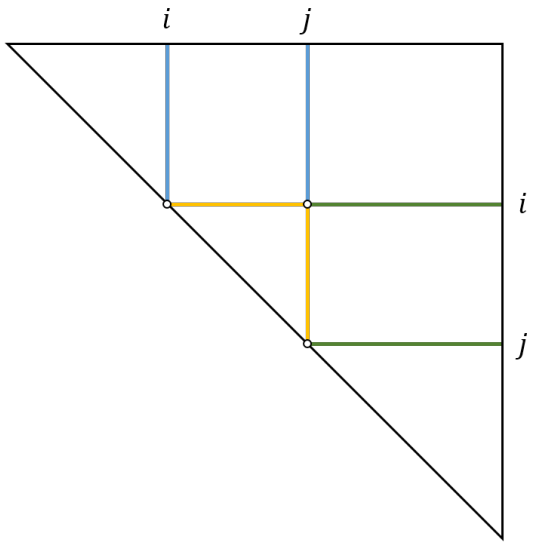


圖 3.17: 共變異數矩陣中需要更新之資料

示，其中以相同顏色標記的向量對需要利用彼此的值來進行更新。可以看到圖中三種不同顏色的向量對以不同的方式排列於共變異數矩陣中，因此更新時的流程也會有所不同，在控制器模組中使用了三個不同的狀態來分別進行三個向量對的更新。

上述的三種向量對之中，以藍色標記的向量對的更新是耗時最少的。因為垂直排列的行向量元素可以利用靜態隨機存取記憶體陣列中每塊記憶體可以平行做讀取及寫入的特性，加上我們使用了雙埠隨機存取記憶體，在一個時鐘週期中最多可以讀寫兩個行向量中的各 40 筆資料，利用浮點數運算單元陣列進行平行運算，可以快速地將兩個向量中的共變異數值更新完畢。因為共變異數矩陣是以切割後地上三角矩陣儲存於靜態隨機存取記憶體陣列中，行向量最多會被分割為四個區段，每個區段會儲存於靜態隨機存取記憶體的不同位址。在更新的過程中會依序將各個區段從記憶體陣列中讀取出來進行更新的運算，運算方式則與式 3.7相同。兩個向量中各個區段於靜態隨機存取記憶體陣列中的位址如圖 3.18所示，可以看到在向量的最後一個區段並不是所有位於該位址的值都需要更新，因此必須根據 i 值將不需更新的靜態隨機存取記憶體停用，並用靜態隨機存取記憶體的寫入遮罩 (write mask) 防止覆蓋掉不該被更新的資料。

接下來，以橘色標示的向量對的更新需要對垂直和水平的兩個向量進行更新。儘管垂直向量跟前面一樣可以快速的從記憶體陣列中讀取出來，但水平向量中的資料是儲存於同一塊靜態隨機存取記憶體中的不同位址，沒有辦法做平行的輸出，必須一個一個的讀取出來，因此整個更新的流程將耗費相當於向量中資料量的時鐘週期。為了簡化讀寫時的記憶體位址計算，我們利用記憶體的兩個讀寫

doi:10.6342/NTU201800667

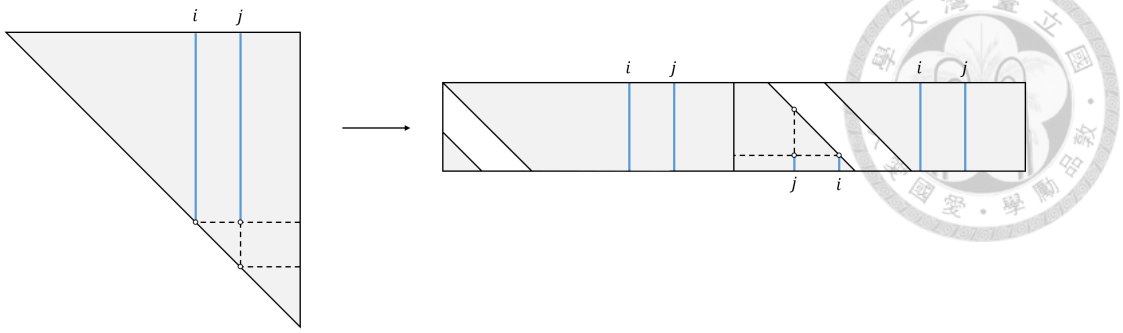


圖 3.18: 垂直向量於靜態隨機存取記憶體陣列中的分段與位址

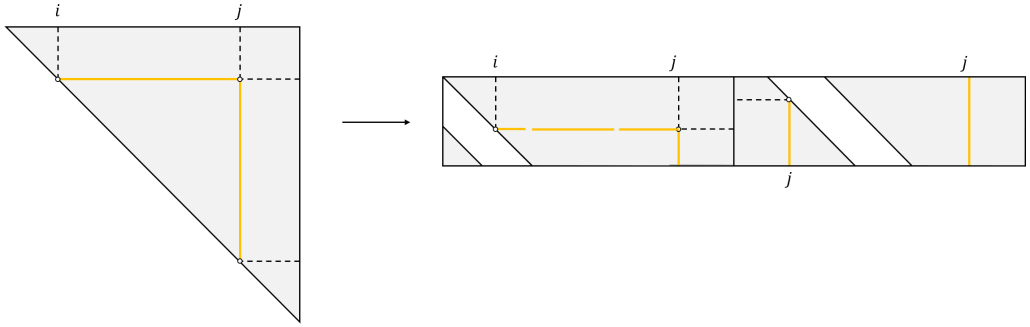
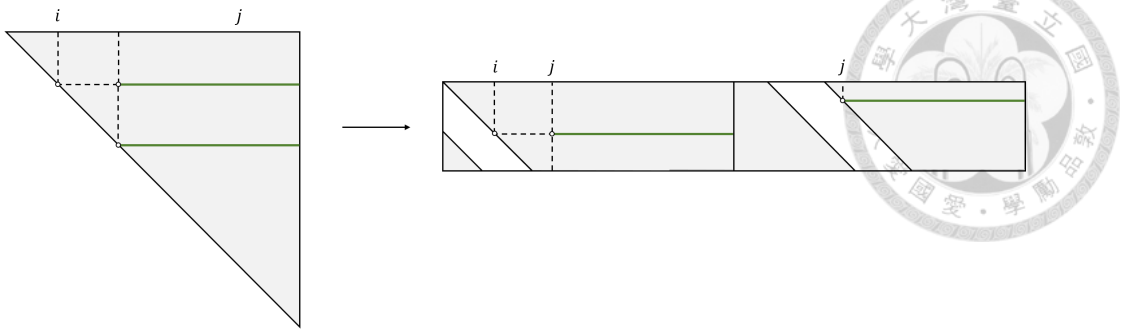


圖 3.19: 垂直向量中的分段與對應的水平向量於靜態隨機存取記憶體陣列中的位址

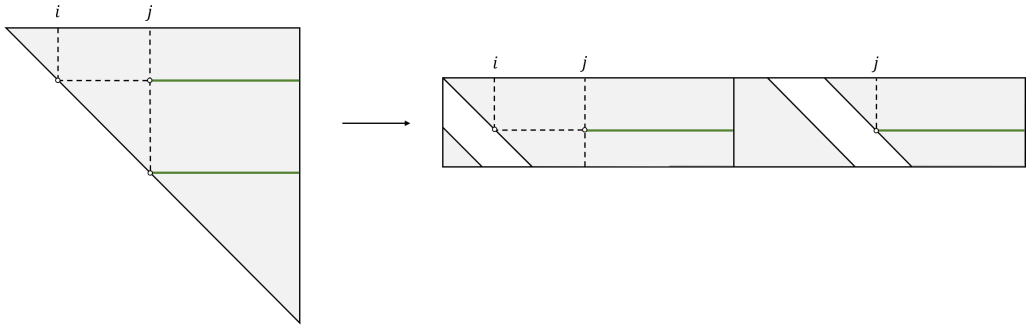
埠分別負責資料的讀取與寫入。跟前面處理垂直向量時一樣，垂直的向量會被切割成幾個區段，水平的向量則因為存於同一塊記憶體中而不會被切割。在更新的過程中，會先將要做更新的垂直向量區段讀取出來。接著，水平向量中的資料會被依序讀出並計算出水平向量更新後的結果，結果一被計算出來負責寫入資料的讀寫埠便會馬上將結果寫入要更新的位址。當目前的垂直向量區段對應到的水平向量資料被全部讀入後，負責讀取資料的讀寫埠會再將下一個區段的垂直向量資料讀出來並接著繼續水平方向的讀取。同時當前垂直區段的更新運算也會在水平資料齊備後開始，當計算完成，負責寫入的讀寫埠便可將整個垂直區段的更新結果平行寫入記憶體中。兩個向量在靜態隨機存取記憶體陣列中的位址分布如圖3.19所示。

綠色的向量在更新時一樣會因為水平方向的讀取而速度較慢。做法上則相當直觀，原則上一樣用一個讀寫埠讀取資料一個讀寫埠寫入資料。一般情況下兩個水平向量中對應的兩筆資料會存放在不同的記憶體中，因此可以同時被讀取出來，負責讀取的讀寫埠依序將兩個向量中對應的兩筆資料讀取出來後可以馬上用於計算更新後的結果，計算出的結果再由負責寫入的讀寫埠寫入應放置的位址即可。有時候需要更新的兩個水平向量會是存放於同一塊記憶體中的兩個向量，儘管如此，如果兩個向量中對應的資料是存放在同一個位址中的不同位元，兩個資

doi:10.6342/NTU201800667



(a) 兩水平向量存放於不同塊記憶體



(b) 兩水平向量存放於同記憶體且不同位址

圖 3.20: 兩水平向量於靜態隨機存取記憶體陣列中的位址分布

料仍然可以同時被讀取與寫入，不會影響到整體的更新速度。然而，因為上三角矩陣被切割存放於靜態隨機存取記憶體陣列的關係，兩個對應的資料在一些情況中會存放在同一塊記憶體中的不同位址，如此一來便必須動用兩個讀寫埠才能夠將兩筆資料同時讀出或寫入。因此在這個情況中兩個讀寫埠將各自負責一個水平向量的所有讀寫，也使得讀與寫的操作不能同時進行，更新的時間會增加為兩倍。圖 3.20展示了兩水平向量於靜態隨機存取記憶體陣列中的位址分布，其中3.20a和 3.20b分別為兩向量存放於不同塊記憶體與存放於同塊記憶體的情形。

最後，完成共變異數矩陣的更新後，會繼續開始下一次的迭代，每一次選取到的兩個行向量間的共變異數若小於設定的閾值，便會被跳過不做旋轉與更新。當一次拂掠中所有的行向量對的共變異數都小於閾值，Hestenes-Jacobi 的演算法便算完成了。共變異數矩陣中對角線上的元素即是目標矩陣的奇異值的平方。然而，共變異數矩陣中的奇異值並沒有按照大小排列，因此當迭代結束後，會將對角線上的元素全部讀取出來，並進行平行泡式排序 (parallel bubble sort) 來取得由大到小的奇異值索引。排序過後的奇異值索引會輸出到外部系統，由外部系統根據此索引將奇異向量矩陣中的奇異向量依序組成我們需要的基底矩陣，完成奇異值分解的計算。

doi:10.6342/NTU201800667

3.4 硬體實驗結果

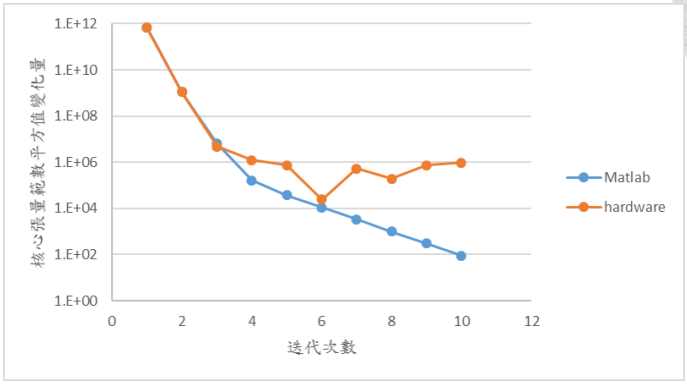
在這一節中，我們將硬體加速架構的近似結果以及運算速度與軟體實作的版本進行比較，以此來檢視設計的成果。此硬體架構可以處理的最大張量尺寸為 $128 \times 128 \times 128 \times 128$ 的四維張量，但由於硬體模擬時間過於冗長，我們以較小的張量作為近似目標完成整個演算法流程的模擬，完整尺寸的張量資料則僅分別就各個運算模組測試其運作的正確性。因此，以下所用於比較的近似結果將是小尺寸張量的近似結果，張量的尺寸為 $64 \times 64 \times 60 \times 60$，近似後的核心張量尺寸為 $40 \times 40 \times 4 \times 8$ 並沿視角模分為 3 個群集進行近似。此外，晶片佈局 (layout) 和合成 (synthesis) 後的模擬時間更是高達暫存器轉移層次 (register transfer level, RTL) 模擬的數十倍以上，因此在現階段，布局後的模擬我們僅做到各個運算正確性的確認，合成後的模擬則做到演算法中一次迭代的正確性確認。

在近似結果方面，因為硬體使用單精度浮點數進行運算，相較於軟體使用雙精度浮點數運算，硬體會有較大的誤差，而這個誤差會影響到交替最小平方法的收斂。在這裡我們使用兩個連續的迭代中核心張量的範數平方值的差做為衡量收斂的依據，軟體與硬體的收斂過程如圖 3.21a所示，硬體實作的結果從第 5 次迭代開始便停止收斂，在一定的範圍內起伏。然而，從圖 3.21b中可以看出，儘管無法收斂到如軟體實作般精準，對於近似結果的影響並不大，軟硬體於每次迭代後的近似誤差都幾乎相同，且從第 4 次迭代後便幾乎不再變動。由此可以看到硬體的近似結果並不比軟體的結果差。

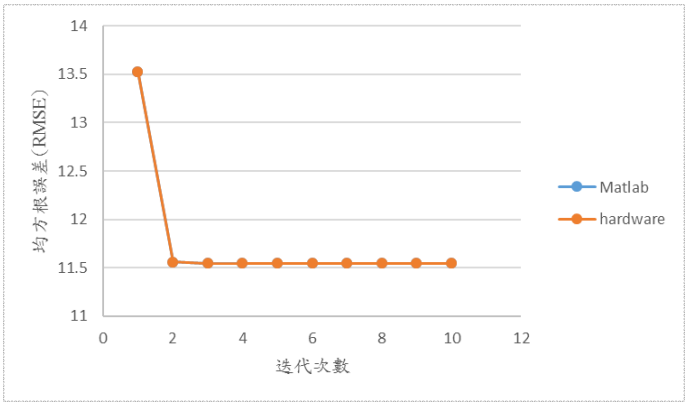
在運算時間方面，在以如前面所說的張量尺寸與參數設定進行近似時，我們的硬體架構可以達到 9.41 倍的加速。軟體的實作環境為 *Intel Core™ i7@4.00GHz* 的中央處理器與 32 GB 的記憶體，使用的程式語言則為 Matlab。硬體與軟體的運算時間列於表 3.2

最後，本硬體架構的規格如表 3.3所示，使用了 TSMC 40nm 製程進行合成與佈局，運作於 476 MHz 頻率，面積與功率分別為 3.152 mm^2 和 744.8 mW。圖 3.22為晶片佈局圖與時脈相位延遲圖。

doi:10.6342/NTU201800667



(a) 軟硬體收斂過程比較圖



(b) 軟硬體近似誤差比較圖

圖 3.21: 軟硬體近似結果之比較

	Matlab	Hardware
Tensor size	$64 \times 64 \times 60 \times 60$	
Reduced rank	$40 \times 40 \times 4 \times 8$	
C	3	
Computation time	30.39s	3.23s
Speed up	-	9.41×

表 3.2: 軟硬體運算時間比較表

doi:10.6342/NTU201800667

Cell library	TSMC 40nm
Operating corner	Slow: 0.81 V, 125°C Fast: 0.99 V, -40°C
Clock frequency	476 MHz
Core area	2.609 mm²
Chip area	3.152 mm²
Gate count	2684679
SRAM usage	40 KB
Power consumption	744.8 mW

表 3.3: 晶片規格 (Pre-layout Simulation)

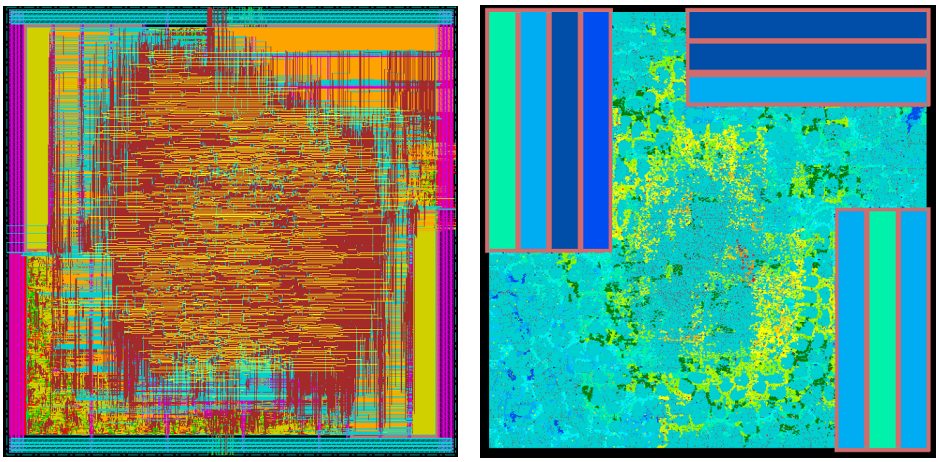


圖 3.22: 晶片佈局圖與時脈相位延遲圖

doi:10.6342/NTU201800667

Chapter 4　結論與展望

4.1　結論

隨著電腦視覺與電腦圖學等領域的發展，需要處理的資訊與資料量都大幅增加，多維度的視覺資料也被廣泛的應用，為了處理如此大量的資料，如何以簡潔的方式表示與儲存資料是一個相當重要的研究議題。不同於傳統的維度縮減演算法，張量近似在維度縮減的同時，保留了資料的多維度結構，充分利用結構中的冗餘資訊以達到更好的壓縮效果。其分別針對各個維度進行維度縮減的特性也讓使用者能更有彈性地控制需要保留的資訊。除此之外，為了將張量近似應用於需要快速生成影像的應用中，可以將張量先行分群後再做近似，減少重建時的運算量，達到快速重建的需求。在本篇論文中我們提出了一個硬體架構來加速分群張量近似演算法，利用硬體平行運算的優點將因為龐大資料量及運算量而相當冗長的近似過程進行加速。我們運用了十塊靜態隨機存取記憶體組成的高頻寬內部記憶體陣列來達到高度的平行運算，並實作了適用於大尺寸長方形矩陣的 Hestenes-Jacobi 奇異值分解演算法來對張量資料進行分析，最終達到了 9.4 倍的加速效果。

4.2　展望

在我們的硬體設計過程中，考量到靜態隨機存取記憶體陣列與浮點數運算單元陣列已經占據相當大的面積，我們盡量將已使用的硬體資源做重複利用，以減少額外的面積。如此一來雖然能減少硬體資源的使用，但也使得設計上較沒有彈性，若在資源使用上適度的取捨，也許能夠在不增加太多資源使用的情況下達到更大幅度的加速。以奇異值分解為例，為了不使用額外的記憶體來儲存共變異數矩陣，將共變異數矩陣進行切割，卻也增加了存取的複雜度，讓平行化進行 Jacobi 旋轉更加困難，也因此減少了再加速的可能性。在往後的研究中，可以此做為主要的改良方向。

doi:10.6342/NTU201800667

doi:10.6342/NTU201800667

參考文獻

[1] Marc Levoy and Pat Hanrahan. Light field rendering. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 31–42. ACM, 1996.

[2] Kristin J Dana, Bram Van Ginneken, Shree K Nayar, and Jan J Koenderink. Reflectance and texture of real-world surfaces. *ACM Transactions on Graphics (TOG)*, 18(1):1–34, 1999.

[3] Hongcheng Wang, Qing Wu, Lin Shi, Yizhou Yu, and Narendra Ahuja. Out-of-core tensor approximation of multi-dimensional matrices of visual data. *ACM Transactions on Graphics (TOG)*, 24(3):527–535, 2005.

[4] M Alex O Vasilescu and Demetri Terzopoulos. Tensortextures: Multilinear image-based rendering. In *ACM Transactions on Graphics (TOG)*, volume 23, pages 336–342. ACM, 2004.

[5] Lieven De Lathauwer, Bart De Moor, and Joos Vandewalle. On the best rank-1 and rank-(r 1, r 2,..., rn) approximation of higher-order tensors. *SIAM journal on Matrix Analysis and Applications*, 21(4):1324–1342, 2000.

[6] Yu-Ting Tsai and Zen-Chung Shih. All-frequency precomputed radiance transfer using spherical radial basis functions and clustered tensor approximation. *ACM Transactions on Graphics (TOG)*, 25:967–976, 2006.

[7] Yu-Ting Tsai and Zen-Chung Shih. K-clustered tensor approximation: A sparse multilinear model for real-time rendering. *ACM Transactions on Graphics (TOG)*, 31(3):19, 2012.

[8] Melissa L Koudelka, Sebastian Magda, Peter N Belhumeur, and David J Kriegman. Acquisition, compression, and synthesis of bidirectional texture functions. In *3rd International Workshop on Texture Analysis and Synthesis (Texture 2003)*, pages 59–64, 2003.

doi:10.6342/NTU201800667

[9] Cobblestones3. https://free3d.com/3d-model/cobblestones-3-86328.html.

[10] Xinying Wang and Joseph Zambreno. An fpga implementation of the hestenes-jacobi algorithm for singular value decomposition. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 220–227. IEEE, 2014.

doi:10.6342/NTU201800667