國 立 交 通 大 學

電信工程研究所

碩 士 論 文

同步分級神經網路在 CUDA 架構上的平行化研究

A Study on the Parallelism of
Synchronous Ranked Neural Networks in CUDA System

研 究 生：陳星豪

指導教授：田伯隆　教授

中 華 民 國一百零一年 七 月

同步分級神經網路在 CUDA 架構上的平行化研究

A Study on the Parallelism of Synchronous Ranked Neural Networks in CUDA System

研 究 生：陳星豪　　Student：Hsing-Hao Chen

指導教授：田伯隆　　Advisor：Po-Lung Tien

國 立 交 通 大 學

電 信 工 程 學 系

碩 士 論 文

A Thesis

Submitted to Institute of Computer and Information Science

College of Electrical Engineering and Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Communication Engineering

July 2012

Hsinchu, Taiwan, Republic of China

中華民國一百零一年七月

同步分級神經網路在 CUDA 架構上的平行化研究

學生：陳星豪　　指導教授：田伯隆

國立交通大學電信工程研究所碩士班

摘要

在本篇論文中，我們將藉由 NVDIA 公司提出的一個利用圖形處理器(Graphic Processor Unit，GPU)的運算架構，名為計算統一設備架構(Compute Unified Device Architecture，CUDA)，模擬同步分級神經網路(Synchronous Ranked Neural-Network，SRNN)的平行分塊更新。另外，針對 SRNN 模型的運作特性，討論分塊更新及神經元等級分佈，對我們的 SRNN 模型最後達收斂態所需計算量造成的影響，希望最後能夠針對我們 SRNN 模型所要處理的問題，找出一些較佳的設定，使得我們能夠用較少的計算量，就讓我們的 SRNN 模型達收斂狀態。最後，我們利用分波多工光相位排列交換器互連系統(WDM OPAS-based Optical Interconnect System，WOPIS)內的封包排程問題，作為我們實作 SRNN 模型平行分塊更新的處理問題，並且從中驗證我們所提出的兩個影響因素，是否如我們所預料的方式，對收斂所需計算量造成影響。另外再針對 CUDA 在各個平行度下執行效率的觀察，找出一個更新區塊個數及平行度最理想的權衡比例。

A Study on the Parallelism of Synchronous Ranked Neural Networks in CUDA System

Student : Hsing-Hao Chen Advisor : Dr. Po-Lung Tien

Department of Communication Engineering

National Chiao Tung University

ABSTRACT

In this thesis, we using the compute unified device architecture (CUDA) which NVIDIA announced a graphic processor unit (GPU) computing architecture to simulate the feature that synchronous ranked neural network (SRNN) can be updated synchronously. And aim at the SRNN operating feature, we discuss the effect of block update manner and neurons' rank distribution on the amount of computation of SRNN convergence. We hope that we can aim at different SRNN handling problems to find some better setting for minimum computation. In the end, we use the packet scheduling of WDM OPAS-based optical interconnect system (WOPIS) problem as our SRNN model's handling problem in block update manner. And prove the two effect factors that is as our expectation. In addition, we also want to find the best trade-off between the number of update block and execution parallelism by observing the execution efficiency in different parallelism.

誌謝

本篇論文的完成，首要感謝指導教授田伯隆老師，在這兩年中，無論是修課或是研究都給我相當大的自由發揮空間，在遇到瓶頸的時候給予適當的指導，就算犯錯也是用鼓勵的方式，要我從中記取教訓再出發，同時對於研究所需的資源也不吝於支持與付出。再來，還要感謝實驗室的柯柏宇及徐子凱學長，平時不管在研究還是課業上都給我相當多的幫助。此外，實驗室的蕭佑霖、馬毓晴同學，以及戴光廷學弟，平時頻繁的研究及課業上的討論，還有日常生活交流，讓整個實驗室的研究氣氛非常融洽，使我能隨時保持愉快的心情進行研究。最後也是最重要的，感謝父母一路支持鼓勵我攻讀碩士學位，不論是生活的幫助或是心靈方面的鼓勵，讓我能夠無後顧之憂的專心完成研究。

目錄

圖目錄

表目錄

1 簡介

傳統的霍普菲耳神經網路(Hopfield Neural Network，HNN)[1][2][3]是一種單層回饋(single-layer feedback)的神經網路，由一組零壹態(binary-value)的神經元透過有權重的連結(weighted connection)完全相連。藉由一個能量函式(energy function)，神經元的激活狀態(0 或 1)被依非同步的(asynchronous)方式更新，直到整個模型達到收斂狀態。且由於 VLSI 及平行計算技術的進步，HNN 及它的變形已經被成功的用來解決許多最佳化問題[4][5][6][7]。但是為了確保 HNN 在離散時間下能夠達收斂狀態，我們必須遵循非同步的更新方式、自我連結權重為零及對稱的連結權重，三條模型的限制。因此，為了能夠提供離散時間的同步更新，我們藉由引進等級神經元的概念，從傳統的 HNN 模型出發，提出一個全新的離散時間同步分級神經網路(Synchronous Ranked Neural Network，SRNN)。而不像傳統的 HNN 模型，藉由讓每一個神經元都擁有不同的等級，我們可以將模型內的神經元分成數個更新區塊，並且讓區塊內的神經元做同步的更新，且區塊間的更新順序毫無限制。而 SRNN 本身的運作方式和 HNN 並無太大差異，透過讓模型內的每個神經元獨立運作遞迴式的更新函數，依其它神經元當前的狀態，來判斷自己的下一個狀態。另外，為了能夠降低模型達收斂狀態所需的計算量，我們將在論文中討論可能影響模型達收斂狀態所需更新週期的因素，希望藉由適當的設定，讓模型能夠花費比較少的時間就達到收斂的狀態。而為了能實做 SRNN 模型的分塊同步更新，我們將利用分波多工的(Wavelength-Division Multiplexing, WDM)光連結網路內的封包排程，來做為我們 SRNN 的處理問題。另外再利用由 NVDIA 公司提出的一個利用圖形處理器(Graphic Processing Unit，GPU)的運算架構，名為計算統一設備架構(Compute Unified Device Architecture，CUDA)[8][9][10]，來做為我們模擬的平台。

為了有效地連結大量的處理器和伺服器，在高性能計算（High Performance Computing, HPC）系統[11][12]和數據中心（Data Center, DC）[13][14]的網路中，我們需要提供高頻寬，大規模，和低延遲的連結網路。而分波多工的光連結網路[15]，剛好提供了高頻寬和低功耗的特性，因此最近一直被視為支援 HPC 和 DN 系統的最好人選。因此在本篇論文中，我們將利

用由可調光波長轉換器（Tunable Optical Wavelength Converter, TOWC）[16][17][18]、光纖延遲緩衝區(Fiber Delay Line Optical Buffer, FOB)[19]，及光相位排列交換器 Optical Phased-Array Switch, OPAS）所組成的，分波多工光相位排列交換器互連系統(WDM OPAS-based Optical Interconnect System，WOPIS) 內的封包排程問題，作為我們實作 SRNN 模型平行分塊更新的處理問題。並且從中驗證及觀察我們在 SRNN 中討論的各種影響收斂所需更新週期因素。

而最近幾年在個人運算架構(personal computer architecture)上，有很巨大的改變正在進行，隨著中央處理器(Central Processing Unit，CPU)遇到半導體製程的問題，如功耗問題(power consumption)、繞線延遲(wire delay)、記憶體速度提升瓶頸(processor-memory gap)...等等，而無法再提高時脈，目前 CPU 都希望就現有的時脈往多核心的方向發展。但為了提供廣泛的一般用途，CPU 必須在計算核心(computation cores)、快取(cache)、輸入/輸出(I/O)，等其它資源中取得一個最有效率的平衡，使 CPU 能達到單一指令執行延遲的最佳化。因此當我們要執行高平行度的程式時，CPU 就未必是我們的最佳選擇。然而，反觀 GPU 本身的設計目的，就是為了達到整體計算吞吐量(throughput)的最大化[20]。因此，GPU 利用非常少的快取及複雜流量控制器，取而代之的是含有多個平行執行的執行器(processor)，並且利用大量的運算元件，而非資料快取來隱藏記憶體的存取延遲。而 CUDA 就是一個在這樣背景下所發展出來的技術，這也是為什麼我們會選擇它做為我們平行化的工具。CUDA 是為 GPU 計算設計的硬體和軟件框架。透過提供各種常用的程式語言，如 C, C++, Fortran 等的衍伸語法，來讓程式設計者很快的學習，而不需使用繪圖專用的程式語言(如: OpenGL)進行開發。現在的 NVIDIA GPU 皆包含必要的硬體組件，靈活的控制和分配到數百個處理器內核的工作量。

而在接下來的論文中，我們將先介紹我們的 SRNN 模型，包含它的架構及運作方式，並且透過範例討論各種影響收斂所需更新週期的因素。再來介紹一下我們 WOPIS 的系統架構，及內部各種路徑競爭和硬體要求限制，最後再說明我們是如何將 SRNN 模型套用到 WOPIS 的封包排程問題。而在進入模擬討論之前，還要先介紹一下我們平行化的工具 CUDA，包含它的運作模式及各種硬體軟體限制。最後，再就我們的模擬結果，討論及驗證之前所推測的理論，並且做出個結論。

2 離散時間同步分級神經網路(SRNN)

離散時間同步分級神經網路(Synchronous Ranked Neural Network, SRNN)，藉由引入分級神經元(ranked neuron)的概念，讓 SRNN 不再像傳統的霍普菲爾神經網路(Hopfield Neural Network , HNN)模型，它可以在離散時間上做到分塊更新(block-update)，也就是說，所有被分到同一個區塊內的神經元會同步更新它們的狀態，且不同區塊間的更新順序沒有任何限制。當我們將所有的神經元分在同一區塊內時，SRNN 的更新方式就變成了完全同步/平行(full parallel)，相反的，假如我們把每一個神經元都視為一個區塊，那 SRNN 的更新方式就變成了序列式的(sequential)。接下來我們會先簡介一下 SRNN 的架構及運作方式，並且說明如何將它用在我們的 WOPIS 的路徑排程上，最後再討論影響達收斂所需的計算量因素。

2.1 模型架構

SRNN 是一種重複回饋的神經網路，每一個神經元 x 依更新規則:$V_x^{(k+1)} = u(net_x^{(k)})$ 更新，其中 $V_x^{(k+1)}$ 代表神經元 x 在第 k+1 個重複(iteration)的輸出；$u(z)$ 代表單位階函數(unit step function)，當它等於 1 時，代表 z 大於等於 0，相反的當它等於 0 時，代表 z 小於 0；$net_x^{(k)}$ 代表神經元 x 在第 k 個重複的網路權重輸入(net weighted input)，也就是說 $net_x^{(k)} = \sum_{y \neq x} W_{xy} V_y^{(k)} - \theta_x$，其中 W_{xy} 代表神經元 y 對神經元 x 的連接權重(connection weight)，而 θ_x 則代表神經元 x 的門檻(threshold)，換句話說，每一個神經元在第 k+1 個重複的輸出，只和預先指定的(pre-assigned) 連接權重，還有其它神經元在第 k 個重複的輸出有關。且一個 SRNN 模型內的所有神經元更新一次，整體計算量會和模型內的神經元個數成平方關係，因為每一個神經元，都要去計算模型內的其它 $n-1$ 個神經元對自己的影響，因此整個模型的計算量就變成了 $n \times (n-1) = n^2 - n : O(n^2)$。且對任意神經元，一個來至其它神經元的正權重(positive weight)，可被視為一種補償(expiatory)，相反的，一個來至其它神經元的負權重(negative weight) ，則可被視為一種約束(inhibitor)。

SRNN 背後主要的設計原則是，將我們的最佳化問題拆解成兩個子問題：規則相關(constraint-related)問題和分級相關(ranked- related) 問題。如同上一個章節的定義，我們對所有和神經元 x 因違反規則 g 而產生競爭的神經元，定義一個衝突集合(conflicting group) G_x^g，再利用函數 $\eta(G_x^g, y) = 1$ 來表示神經元 $y \in G_x^g$，因此為了能夠符合規則，神經元 x 和神經元 y 之間的權重應該要為一種約束關係，另外，為了不失一般性，我們可以將受限於規則 g 的神經元表示為:

$$C: \sum_x \sum_{y \neq x} \sum_g \rho_g \eta(G_x^g, y) V_x^{(k)} V_y^{(k)} + \sum_x c_x V_x^{(k)} + \sigma = 0 \tag{3}$$

其中 ρ_g, c_x 及 σ 為常數。

為了建立符合規則 C 的 SRNN 模型，我們必須先決定神經元間的連接權重及神經元本身的門檻，一開始我們依非同步(asynchronous)的方式(區塊大小為一個神經元)更新推導，且每個神經元在一個遞減的規則相關(constraint-related) 函數內更新，函數定義如下：

$$f^{(k)} \equiv \sum_x \sum_{y \neq x} \sum_g \rho_g \eta(G_x^g, y) V_x^{(k)} V_y^{(k)} + \sum_x c_x V_x^{(k)} + \sigma$$

因為一次只會有一個神經元 x 更新，$V_y^{(k+1)} = V_y^{(k)} \quad \forall y \neq x$ 且 $\Delta f^{(k)} = f^{(k+1)} - f^{(k)} = \frac{\partial f^{(k)}}{\partial V_x^{(k)}} \times [V_x^{(k+1)} - V_x^{(k)}]$，藉由設定 $V_x^{(k+1)} = u\left(-\frac{\partial f^{(k)}}{\partial V_x^{(k)}}\right)$，我們可以保證 $\Delta f^{(k)} \leq 0$，也就是說，如果 $f^{(k)}$ 也是一個非負函數，且有下界 0，最後都會收斂到 0(符合規則 C)。因此，從式子 $net_x^{(k)} = \sum_{y \neq x} W_{xy} V_y^{(k)} - \theta_x = -\frac{\partial f^{(k)}}{\partial V_x^{(k)}}$，再經過一些額外的運算後，我們可以得到 $\frac{\partial f^{(k)}}{\partial V_x^{(k)}} = \sum_{y \neq x} \sum_g 2\rho_g \eta(G_x^g, y) V_y^{(k)} + c_x$，而相對應的權重和門檻也可表示為:

$$\begin{cases} W_{xy} = \sum_g -2\rho_g \eta(G_x^g, y) \\ \theta_x = c_x \end{cases} \tag{4}$$

接下來為了處理分級相關問題，我們將引入分級函數(rank function)，神經元 x 的等級(rank)我們就以 $r(x)$ 表示，其中函數 r 為一實數函數，且為了保有 SRNN 模型可同步分塊更新的特性，我們會讓每個神經元的等級皆不相同。除此之外，兩個神經元互相的權重關係也是非對稱的，高等級的神經元會從低等級的神經元那獲得額外的正激發(positive stimulation)，而這種額外的等級激發(rank stimulation)我們就將它表示成 $A\times\{1-u[r(y)-r(x)]\}$，舉例來說，從圖 2-1 可知神經元 x 的等級比神經元 y_1 來的高，因此神經元 x 對神經元 y_1 的權重不會有任何改變 $W'_{y_1x}=W_{y_1x}$(沒有額外的等級激發)，但神經元 y_1 對神經原 x 的權重就會變成 $W'_{xy_1}=W_{xy_1}+A$(有額外的等級激發)，因此彼此間的權重關係也變成了不對稱的。如果我們讓 $A=-W_{xy_1}$，那因為我們定的規則而產生的約束權重 W_{xy_1}，就會因此而失去效用，因為等級激發的設計，就是讓我們能夠無視規則 C，在一個衝突集合中激活(enable)一個等級較高的神經元，且不管在同一個衝突集合內，是否已經有等級較低的神經元被激活了，因為到最後這些等級較低神經元，還是會因為等級較高神經元激活而被壓抑(disable)掉。

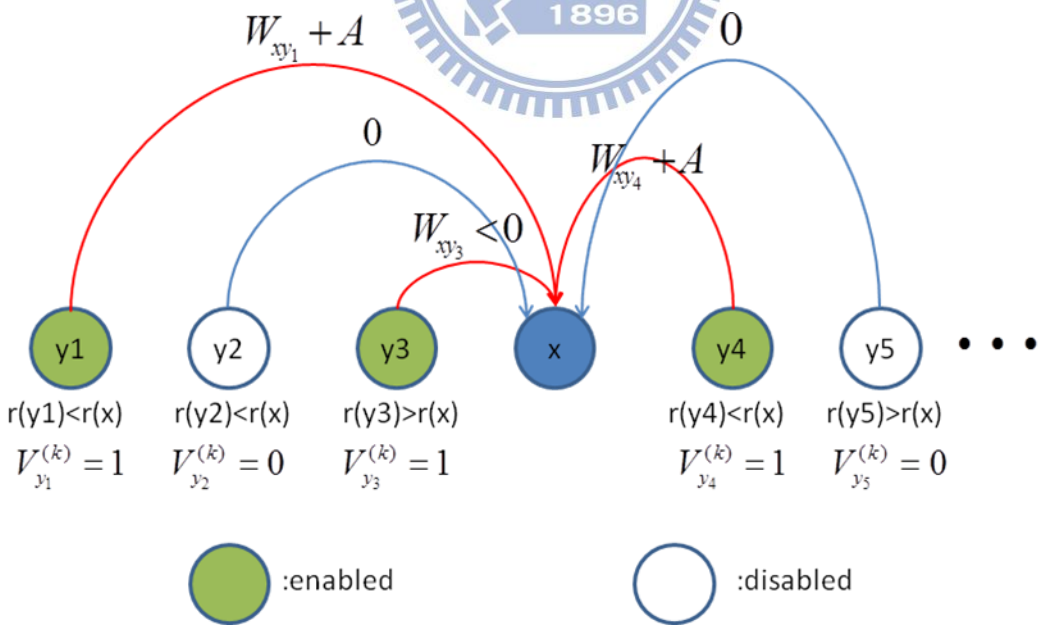


圖 2-1 SRNN 等級激發

同步(分塊)更新和神經元間的非對稱權重，或多或少都對 SRNN 的收斂帶來了挑戰，所以接下來我將參考圖 2-2 提供一個定理，確保 SRNN 的最後結果都會收斂到最佳解，且定理中還包含神經元間的權重和神經元本身門檻的制定，及最後收斂需要的重複次數上限。

定理:給予神經元集合U，且將神經元切為K個更新區塊，神經元候選人集合$T=\{x \mid \theta_x \le 0, x \in U\}$，一些衝突集合$G_x^g$完全包含於集合$U$內，且$E=\{x_1^1, x_1^2 \cdots, x_1^H \mid x_1^i$是集合$T-\bigcup_{j=1}^{i-1}\left(\{x_1^j\}\cup\bigcup_g G_{x_1^j}^g\right)$內最高等級的神經元，且集合滿足$T-\bigcup_{j=1}^{H}\left(\{x_1^j\}\cup\bigcup_g G_{x_1^j}^g\right)=\varnothing\}$，且我們稱 W=$\{x_1^1, x_1^2 \cdots, x_1^H\}$為當選人集合。若我們將神經元間的權重定為$W_{xy}^{'}=\sum_g -2\rho_g \eta(G_x^g, y)u[r(y)-r(x)]$，神經元本身門檻$\theta_x > -2\rho_g \ \forall \rho_g > 0$，那麼在(1+2$H$)$K$個重複內，SRNN 就會收斂到只有在集合$S$內的神經元才會被激發，且其它集合$U$內的神經元都會被壓抑掉。

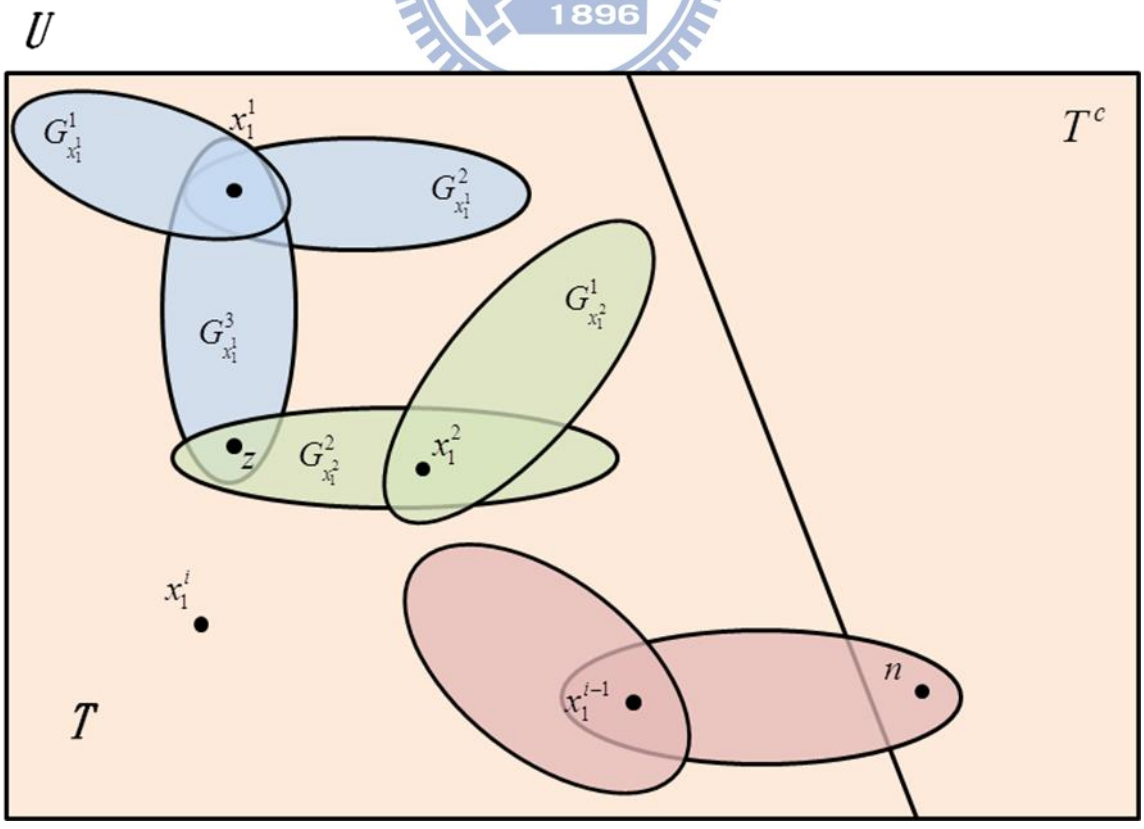


圖 2-2 多重碰撞集合說明圖

值得注意的是，定理提供了我們一個處理 SRNN 有優先權路徑排程問題的架構，我們可以簡單的把每一條路徑對應到一個神經元，並且把路徑的衝突集合也對應到神經元的衝突集合，且依據定理，SRNN 路徑排程最後會選出最大符合規則且等級最高的路徑集合。在最後一節我們會說明所有路徑排程對應到 SRNN 上的細節。

而從上述的介紹過程中我們可以得知，我們的 SRNN 模型是藉由模型內的神經元，各別獨立運算遞迴式的更新函數，使得最後整個模型能夠到達穩定狀態，且因為所有神經元更新一次的計算量為 $O(n^2)$，而每個神經元達收斂所需的更新次數又和最後會為激發態的神經元個數成正相關$((1+2H)K)$。因此為了降低我們 SRNN 模型達收斂所需的時間，我們可以從兩個地方下手：

1. 善用每個神經元可獨立平行運算的特性，將整個模型平行化。但必須解決因遞迴式的更新函數，所造成的高度資料相依性，竟而導致模型在平行化時，需要付出大量的同步負擔。
2. 藉由降低達收斂每個神經元所需的更新次數，來降低模型達收斂所需的計算量。

2.2 SRNN 運作方式

為了讓大家對上述定理有更深度的瞭解，在這節中我們將針對這個定理，提出一個範例用來說明及驗證。但在開始說明範例之前，我們要先說明兩個接下來會一直被我們提到的名詞：

- 更新週期(update cycle)：在我們的 SRNN 模型內，無論我們將模型切成幾個更新區塊，當我們將所有的神經元都更新一次，就稱作一個更新週期(update cycle)。
- 生還者集合(survivor set):在我們 SRNN 模型的運作過程中，假如我們已經激發了 $x_1^1, x_1^2 \cdots, x_1^i, i \le H$ ，並且也壓抑了，和這些神經元在相同碰撞集合內的神經元，那麼我們稱集合 $S = T - \bigcup_{j=1}^{i-1} \left(\{x_1^j\} \cup \bigcup_g G_{x_1^j}^g \right)$ 為現在這個狀態的生還者集合。

如圖 2-3，這裡我們提供十六個神經元，且隨機給予不同的等級及初始狀態(激發態或壓抑態)，另外我們要求在每一行及每一列只能有一個最高等級的神經元被激發，因此在同一行或同一列的神經元就形成了一個碰撞集合，換句話說，每一個神經元都同時被包含在兩個碰撞集合內(行跟列)。另外，我又假設第二列的神經元為非候選人集合(non-candidate set)，而更新時，我們將神經元分割成四個更新區塊(圖中的虛線部分)，並且依左上右上左下右下的順序，依序同步更新區塊內的神經元(順序是可以任意改變，只是為了簡化題目)。

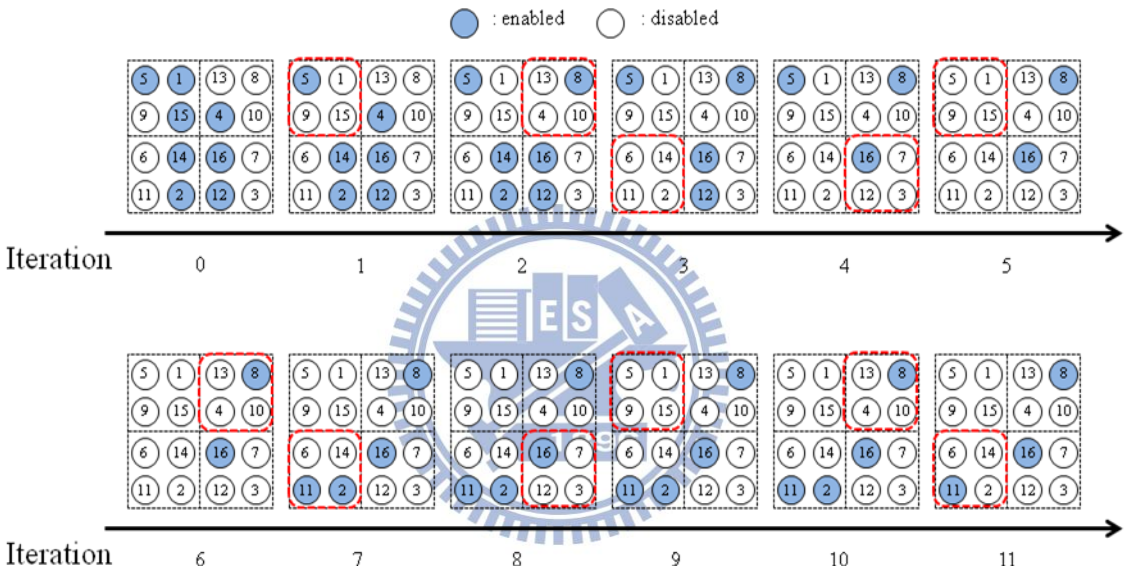


圖 2-3 定理範例分塊更新

如此依照定理，我們可以令 $\rho_g = 1$ 對所有的 g 皆成立，且令第二列中的神經元 $\theta_x = c_x = 1$，剩下的神經元則為 $\theta_x = c_x = -1$。因此對於在第二列中的任一神經元，因為 $net_x^{(k+1)} = \sum_{y \neq x} W'_{xy} V_y^{(k)} - \theta_x \leq -\theta_x < 0$，所以只要它們在更新過一次後，無論是接下來的任一重複內，這些神經元都會一直維持著壓抑態。

至於其它在候選人集合內的神經元，在第一個重複內(第一個更新區塊)，因為 $r(x(1)) < r(x(5))$ 且 $x(5)$ 在第零個重複時(初始狀態)也已經被激發，$net_{x(1)}^{(1)} = \sum_{y \neq x(1)} W'_{x(1)y} V_y^{(0)} - \theta_{x(1)} = -2 \sum_{y \neq x, r(y) > r(x(1))} V_y^{(0)} - \theta_{x(1)} \leq -2V_{x(5)}^{(0)} - \theta_{x(1)} = -2 + 1 < 0$，因此很明顯的 $x(1)$ 會被壓抑。接下來到了第二個重複，$x(13)$ 會因為同一行中的 $x(16)$ 為

激發狀態而被壓抑，而 $x(8)$ 則是因為在同一行或同一列中，沒有等級比它高的神經元是激發狀態，而會被激發。到了第三個重複，因為更新區塊內的神經元，在同一行或同一列中都有等級比較高的神經元為激發狀態，因此所有的神經元在更新後都會被壓抑。依照這個規則，持續的執行我們的更新演算法，直到第七個重複，因為在 $x(11)$ 和 $x(2)$ 的碰撞集合內都沒有神經元是被激發的，因此 $x(11)$ 和 $x(2)$ 在這一個重複會同時被激發，且無視於他們兩個在同一個碰撞集合內。然而，等到下一次更新到同一個更新區塊時(第十一個重複)，$x(2)$ 還是會因為 $x(11)$ 為激發狀態而被壓抑，而我們整個模型也就達到收斂狀態了。

在定理的最後，我們有提出一個模型達收斂狀態所需的重複次數上限 $((1+2H)K)$，接下來我們將藉由上述的範例，來說明這個上限的由來。首先，是式子最後面的那個 K，我們知道 K 的意義是代表我們更新區塊的個數，而在式子的最後會乘上 K，則是因為，當我們將模型切成越多個更新區塊時，所有模型內的神經元，都更新一次所需的重複數量也越多(也就是一個更新週期所需的重複次數)，且和區塊個數就恰好為正比關係，也就是說，有幾個更新區塊，就需要花多少個重複去將所有的神經元更新一次。在上述範例中，我們將模型切成四個更新區塊，因此所有神經元更新一次剛好也需要四次重複。而式子前面的 $1+2H$，我們又可以將它拆解為 1 跟 $2H$ 分別解釋。首先是 1，從範例裡的第一個更新週期中(重複一到四)，我們可以發現，所有更新到的非候選人神經元，都會在他第一次被更新後就一直維持著壓抑狀態，而為了確保應該被激發的神經元($x_1^1, x_1^2 ..., x_1^H$)，不會因為非候選人的激發狀態，而被壓抑掉，我們必須先花費一個更新週期，以確保在非候選人集合內的神經元都已經被壓抑。換句話說，如果問題內所有神經元皆在候選人集合內，那麼就不需要這額外的一個更新週期。至於接在後面的 $2H$，則是因為我們模型更新的最壞情況，就是當我們在每個更新週期內，激發了生還者集合中(如圖 2-3 中的重複四，除了第三行和的第三列外的神經元)，等級最高的，然後再花費一個更新週期，壓抑掉所有在同一個碰撞集合內的神經元後，所剩下的生還者集合中，等級最高的神經元為壓抑狀態(如圖 2-3 重複四的 $x(11)$)。這代表我們要激發一個最後應該要為激發狀態的神經元，都要花費兩個更新週期，而 H 剛好就是最後模型為收斂狀時，會是激發狀態的神經元數量。

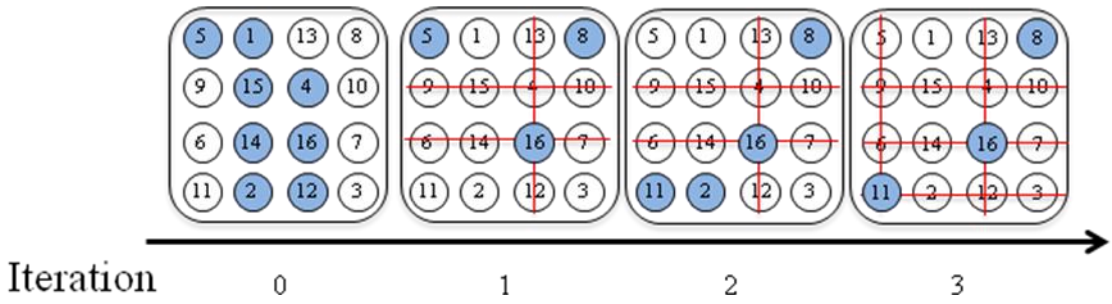


圖 2-4 定理範例全平行更新

接下來，我們使用相同的範例，只是把 SRNN 的更新區塊變成一個(K=1)，換句話說，也就是以全平行的方式更新，如圖 2-4 所示。在第一個重複時，第二列的神經元(非候選人集合)一樣會直接被壓抑掉。此外，在十六個神經元中等級最高的 $x(16)$ 一樣維持激發狀態。事實上，因為 $x(16)$ 在初始狀態時就為激發態，因此可以導致和它在相同碰撞集合(第三行和第三列)內的神經元被壓抑，所以我們可以發現，在第一個重複內，所有第三行和第三列的神經元都被壓抑掉了。依照這個道理，$x(11)$ 為扣掉 $x(16)$ 所在的碰撞集合後，生還者集合內等級最高的神經元，因此它會在第二個重複時被激發，而和它在相同碰撞集合的神經元，則會在下一個重複(重複三)中被壓抑掉。最後，在剩下的神經元中($x(1)$ 和 $x(8)$)，因為它們兩個屬於同一個碰撞集合，且最高的那一個也已經被激發，而比較低的那個則一樣維持壓抑，SRNN 則在這一個重複後收斂。

值得注意的是，我們可以發現，全平行更新和分塊更新，最後收斂的結果是一模一樣的，但很明顯的，全平行更新所花的重複次數卻是比分塊更新來的少的。另外，我們還可以發現，藉由激發高等級的神經元，可能導致在同一個碰撞集合內，同時有兩個以上的神經元為激發狀態，也就是暫時的違反了我們的規則相關函數(規則相關函數變成不是嚴格遞減)，然而，在一定的更新次數後，低等級的神經元還是會因為我們的規則而被壓抑掉，並在最後使我們的規則相關函數收斂到穩態。

2.3 影響收斂所需更新週期的因素

在上一節中我們提到，我們將模型內的神經元都更新一次，稱作一個更新週期，而很明顯的，我們達收斂所需的更新週期，其實也可以代表模型達收斂所需的總計算量。因此，如果我們能夠找到一些影響收斂所需更新週期的因素，是有助於降低我們 SRNN 模型達收斂所需的計算量的。而在這一章節中，我將嘗試提出一些因素，並且討論它們是如何影響我們收斂所需更新週期。但可以事先得知的是，無論是哪一種因素，我們都無法保證怎樣做一定可以降低我們的更新週期，或是可以降低多少更新週期，因為不管是哪一個因素，都還會受到其它因素的影響。因此，我們只能針對單一因素，提出一個方向，是有助於提高用比較少的更新週期就達收斂的機率。

而在進入因素的討論前，我們要先提出我們 SRNN 模型的，最少更新週期更新方式：

[最少更新週期更新方式]如果我們可以依神經元的等級高低，由大到小，依序一個一個神經元更新，那麼無論是怎樣的問題模型，我們都可以保證在一個更新週期內，SRNN 模型就達到收斂狀態。

藉由神經元序列式的更新方式(一次只會有一個更新它的狀態)，我們可以確保彼此有影響關係的(在同一個碰撞集合內)神經元不會同時更新。另外，由大到小的更新方式，則是要確保，所有可能對自己造成影響的神經元，都以達收斂該有的狀態，藉此避免神經元會受還沒達收斂狀態的神經元影響。因此，無論碰撞集合或是神經原等級如何分佈，我們都可以確保每一個神經元在更新時，可以對它造成影響的神經元都已經達到收斂時該有的狀態。所以每一個神經元都只要更新一次，就可以達到收斂該有的狀態。而這樣的更新方式，則可以作為我們後面討論影響收斂所需更新週期因素的大方向，更新方式越像，越有可能用較少的更新週期達到收斂狀態。

2.3.1 全平行更新和分塊更新

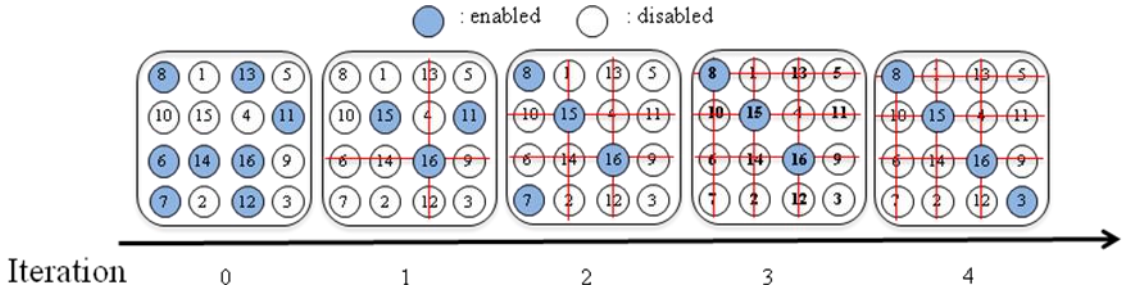


圖 2-5 隨機等級安排全平行更新

在這個因素的探討中，我們沿用上一節的範例，只更改一開始各個神經元的狀態及等級隨機分佈的情形，並假設所有的神經元皆在候選人集合內。從圖 2-5 我們可以發現，因為在初始狀態(重複零)時，最高等級的神經元($x(16)$)就已經為激發狀態了，因此重複一時，所有和它在相同碰撞集合內的神經元就會全部被壓抑掉。且因為等級第二高的神經元($x(15)$)和它分屬不同一個碰撞集合內，所以也會順便在這一個重複內被激發。到了重複二，所有和 $x(15)$ 在同一碰撞集合的神經元都會被壓抑掉，順便激發生還者集合中的兩個神經元(因為 $x(5)$和 $x(3)$ 在重複一時有 $x(11)$ 在壓抑，因此只激發 $x(8)$ 和 $x(7)$)。到了重複三，因為在生還者集合中，等級最高的($x(8)$)已經在上一個重複中被激發，因此在這個重複就可以壓抑生還者集合中和它在同一個碰撞集合的神經元。等到了最後一個重複，只需要激發最後一個神經元，整個模型就達到收斂狀態了。

接下來，我們依照上一節範例的分割方式，將我們的模型分割成四個更新區塊(如圖 2-6)，且一樣依左上、右上、左下、右下的順序，依序同步更新區塊內的神經元。

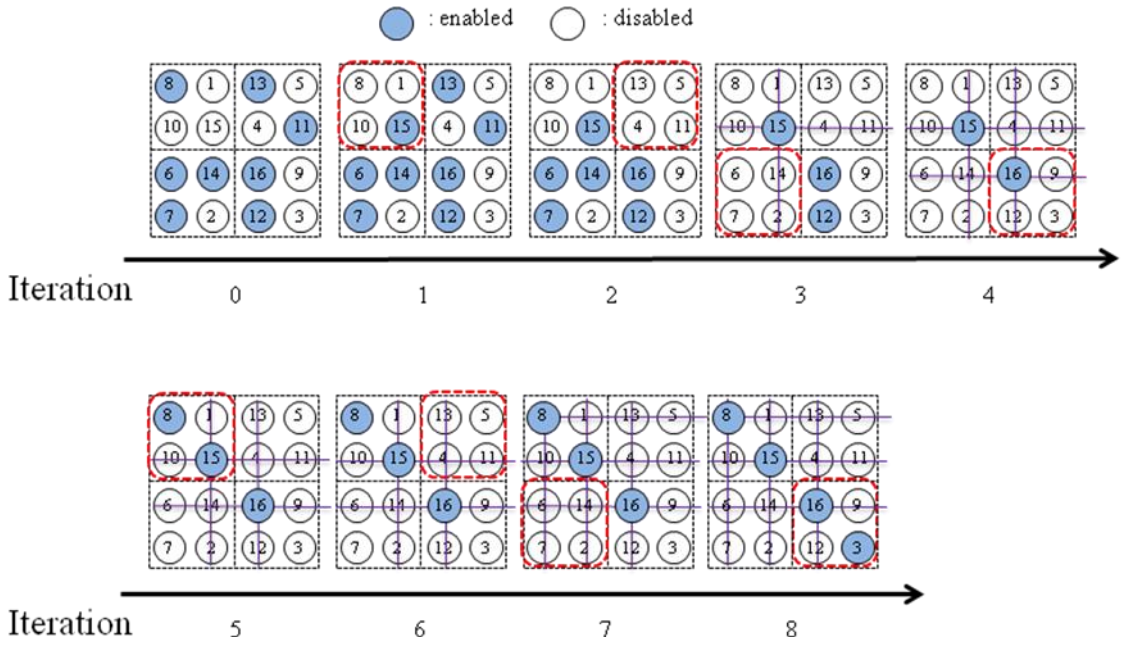


圖 2-6 隨機等級安排分塊更新

依照我們的更新演算法，在第一個重複時，$x(8)$因為$x(13)$為激發狀態而被壓抑，但$x(15)$因為是他所在的兩個碰撞集合中，等級最高的神經元，因此可以直接被激發，且就這樣一直維持激發狀態到最後。到了重複二，$x(13)$及$x(11)$分別因為$x(16)$及$x(15)$為激發狀態而被壓抑。到了重複三，$x(14)$，$x(6)$及$x(7)$也都因為所在的碰撞集合內有其它等級較高的神經元為激發狀態而被壓抑。只不過有一點值得住意的是，因為$x(15)$在第一個重複時就已經被激發，而和它在同一個更新區塊也同時屬於相同碰撞集合的神經元($x(1)$及$x(10)$)，在經過重複一後，也沒有被激發，因此在重複三時，我們就已經將和$x(15)$屬於同一碰撞集合的神經元都壓抑下來了。而到了重複四，因為最高等級神經元$x(16)$在初始狀態就為激發，因此到了這個重複時，它就已經將和同屬相同碰撞集合的神經元都給壓抑下來了。另外，到了重複四，其實我們才經過一個更新週期，但和全平行相比一下，我們已經做到了它花費兩個更新週期才做的到的事。到了重複五，因為總共也只剩下了四個神經元，且剛好等級最高的$x(8)$又在第一個更新區塊內，因此直接激發(因為剩下的四個神經元都不是激發態，因此就算在第一更新區塊的神經元等級不是最高也會被激發)。然後我們直接跳到重複七，這時$x(8)$已經把剩下且和他在同一碰撞集合的神經元都壓抑下來了，比照全平行更新，這是經過三個更新週期後才能得到的狀態，但在分塊更新內，我們只花費不到兩個更新週期就達到了這個狀態。最後重

複八，因為剩下的最後一個神經元剛好也在最後一個更新區塊，因此直接激發，而整個模型也達到了收斂狀態。

由上述兩種不同的更新方式我們可以發現，全平行更新因為一次都是更新所有的神經元，導致較容易激發多餘(後面還是會被壓抑掉)的神經元(參考圖 2-4 重複二的 $x(7)$)，且當它多激發後，後面還要浪費一個更新週期將它壓抑下來 (參考圖 2-4 重複二到重複三的情形)。反觀分塊更新，因為一次更新較少的神經元，本身就比較不會發生激發多餘神經元的情形，且也比全平行更新還有機會，在壓抑和 x_1^i 同一個碰撞集合的神經元的過程中，就先激發了 x_1^{i+1}。同樣道理，又可以提早開始壓抑 和 x_1^{i+1} 同一個碰撞集合的神經元(參考圖 2-6 重複一到三的 $x(15)$)，且一樣有機會提早激發 x_1^{i+2}，以此類推。換句話說，分塊更新是有助於在一次的更新週期內，同時做到壓抑和現有最高等級神經元(x_1^i)，在同一碰撞集合內的神經元，並且同時激發下一個要被激發的神經元(x_1^{i+1})。另外可以想像的，當我們將模型分割成越多個更新區塊時，上面所提到的現象發生機率也會跟著提高，且也越接近我們最少更新週期的更新方式(越接近序列更新)。因此，有比較高的機率，用較少的更新週期模型就達收斂。

2.3.2 神經元等級安排

在利用 SRNN 處理某些問題時(如我們的 WOPIS 交換器排程)，我們是有機會對問題的神經元等級做有限程度的安排。因此，在看完更新方式對更新週期的影響，接下來換看看神經元等級安排的影響，對整體模型達收斂狀態，所需更新週期的影響。我們將把上一小節的範例，當成等級隨機安排的代表，並在下面提出另外兩種極端的案例，且同樣使用分塊更新和全平行更新，神經元初始狀態的相對位置也是一樣的。

在第一個規律中，我們安排的規律性是，每個神經元右邊及下面的神經元等級，一定都比自己來的高，因此當每一個當選人集合內的神經元在運作時，都要等到自己右邊及下面的神經元都已達收斂狀態時，自己才有機會達收斂狀態。而在第二個規律中，因為我們知道在我們的範例中，最後當選人集合內的神經元位置的相對關係(四個神經元的行跟列都不可以相等)，因此我們就直接將四個等級最高的神經元，放在對角線上。並且在相同的更新方

式下，比較所需的更新週期。

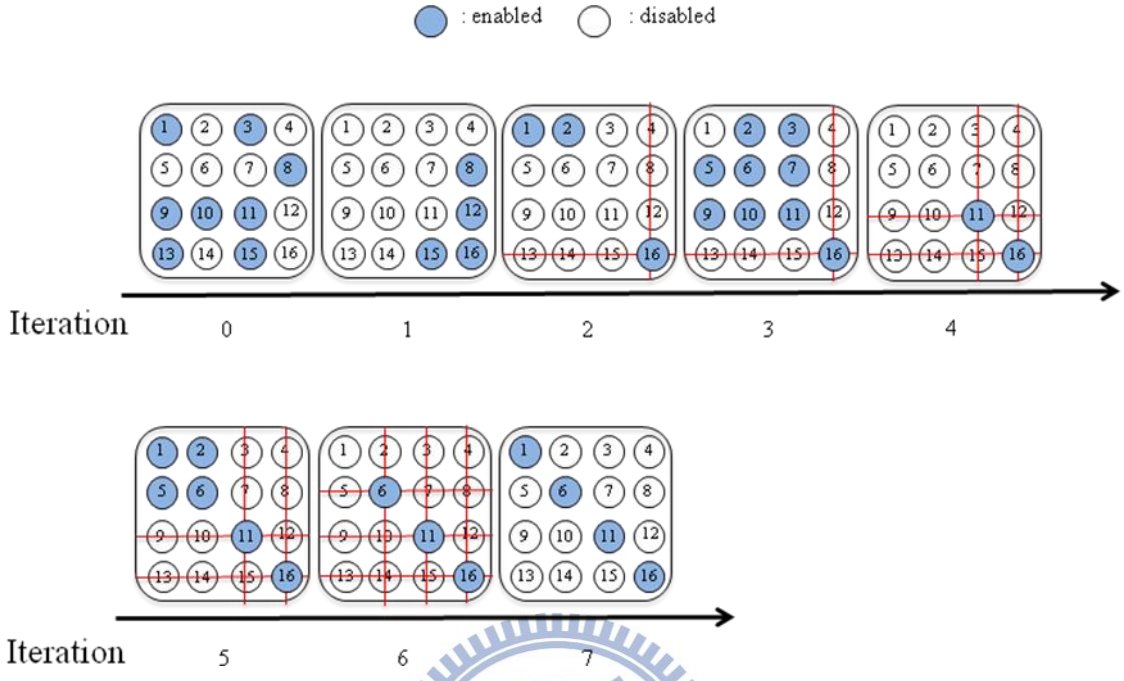


圖 2-7 規律(一)等級安排全平行更新

首先，我們先看圖 2-7 規則(一)等級安排全平行更新，從重複次數上我們可以直接發現，在這樣的等級安排和更新方式下，收斂所需的重複次數(七次)，只差一就達到了我們在定理中所提出的收斂重複次數上限(全平行 K=1；最後激發四個神經元 H=4；沒有非候選人集合，所以不用額外加 1。上限：$2H$=8)，而差一的原因，是因為激發了最後一個神經元後，就沒有神經元需要壓抑了。而從更新的過程裡我們也可以發現，因為等級安排的規則，導致 x_1^i 所在的碰撞集合內，永遠存在比 x_1^j (對所有的 $j|i<j\le H$)等級高，且和 x_1^j 也在相同的碰撞集合內。如圖 2-7 中，$x(16)$ 是整個模型的 x_1^1，而 $x(11)$ 是 x_1^2，但因為和 x_1^1 在相同碰撞區間的 $x(15)$ 及 $x(12)$，剛好也和 x_1^2 在同一個碰撞集合，因此我們一定要等到 $x(15)$ 及 $x(12)$ 被 x_1^1 壓抑掉後，才能激發 x_1^2，且相同的情形一樣會發生在 x_1^3 和 x_1^4 上。

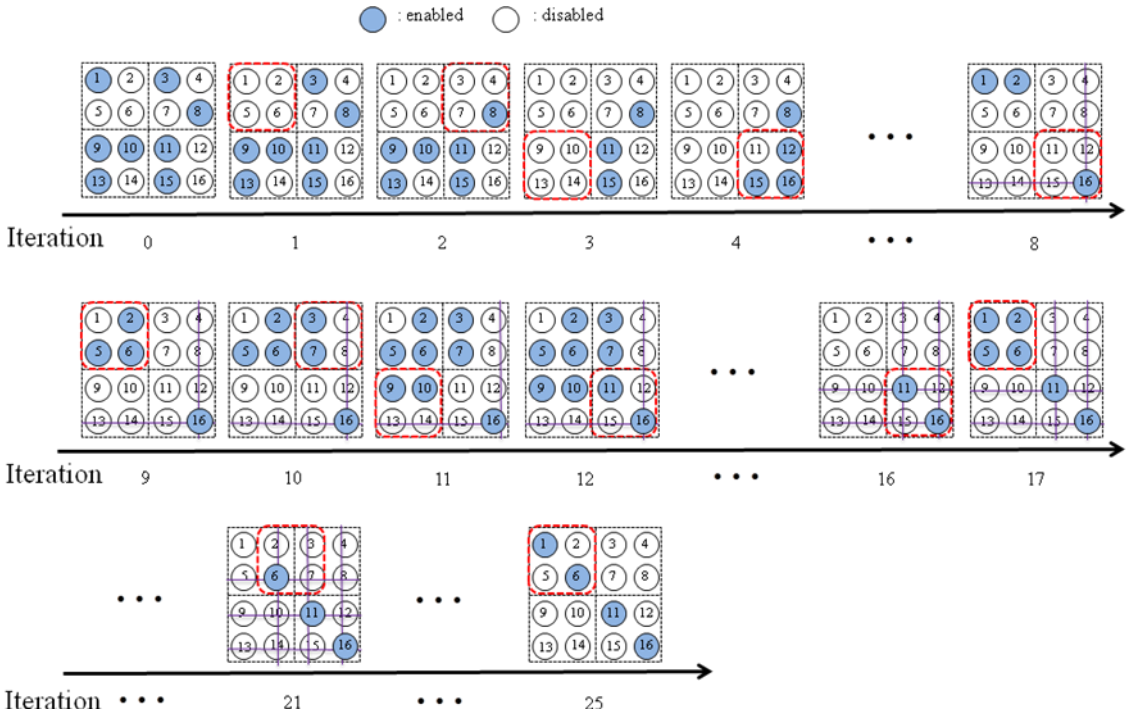


圖 2-8 規律(一)等級安排分塊更新

而在圖 2-8 的分塊更新中，我們也可以發現，其達收斂所需的重複次數，一樣和定理中的上限差不多，且從更新的過程裡，假如我們都先看每個更新週期後的結果(也就是先只看重複四，八…等，四的倍數的重複)，我們可以發現，在每一個週期的結果都和全平行的一模一樣，也就是說，分塊更新在這樣等級安排下，完全無法比全平行更新，用較少的更新週期達到收斂。假如我們在細看更新區塊，和最後為激發態神經元間的關係，會發現 x_1^1 及 x_1^2 都在我們的第四個更新區塊，也就是說，如果要激發這兩個神經元，其實就和全平新更新一樣，需要花完整的一個更新週期，才能將這兩個神經元激發。因此，假如我們將更新區塊的更新順序反過來，其實可以用較少的更新週期(五個)，就達到收斂狀態。或是可以依我們模型碰撞集合的特性，我們最後模型收斂的結果，一定會是四個分別在不同行跟列的神經元，因此，如果我們依這個特性去分割我們的更新區塊，這樣不管我們的區塊更新順序為何，都比我們上述的分割方式，用較少的更新週期就達可達收斂狀態。而這樣的結果反應出來的是，分塊的方式及區塊間的更新順序，對所需的更新週期都是會有影響的，且還會互相影響。而這也直接驗證了我們在這一章節的最一開始，就提出影響收斂所需更新週期的因素，不只一個且彼此還會互相影響。

而針對分塊的方式及更新區塊間的更新順序，因為這必須是在，先知道神經元等級及碰撞集合的分布的前提下，我們才有機會針對這兩因素，去做分塊更新的最佳化。但在一般的案例中，我們很難在事先就同時掌握這兩個因素，甚至在有些案例中，這兩個因素是會一直變化的，如我們在這篇論文需要解決的 WOPIS 有優先權路徑排程問題，它每條路徑的等級會因為要通過的封包優先權不同而改變，而內部碰撞集合的分佈，也會因為確定激發了某些路徑後而有了改變。因此，在本篇論文中，我們不深入探討這兩個因素，對收斂所需要的更新週期的影響，只依最少更新週期更新方式，提供兩個大方向：(1)讓同一個更新區塊內的神經元，所屬的碰撞集合不同。因為只有同一個碰撞集合內的神經元，才會對彼此的收斂狀態造成影響。因此，如果更新區塊內的神經元皆屬不同的碰撞集合，我們就不會發生，同一碰撞集合內的神經元，要因彼此的上一個狀態，同時更新。(2)讓等級高的神經元早點更新。因為我們都知道，在我們的 SRNN 模型內，只有等級高神經元能夠影響等級低神經元最後的結果，換句話說，等級越高的神經元，能夠影響它最後結果的神經元個數也越少。因此，如果我們能讓等級高的神經原先更新，我們也能比較快讓這些等級高的神經元達到最後的穩態，而藉此讓整個模型都能夠較快達到穩態。

而在規律二中，我們提出了一個相對應的反例，我們直接將我們模型中等級最高的四個神經元，分別放在彼此都不會互相碰撞的位置(也就是四個神經元，彼此都不在彼此的碰撞集合)，這樣的好處是，$x_1^1, x_1^2, x_1^3, x_1^4$ 都不會有其它神經元可以壓抑它們。而從圖 2-9 我們也可以觀察出，不管是全平行更新還是分塊更新，都只需要花一個更新週期，就可以同時激發 $x_1^1, x_1^2, x_1^3, x_1^4$，只是分塊更新在激發的同時也順便把其它神經元跟壓抑下來，而全平行更新，則還需要再花一個更新週期將 $x(8)$ 壓抑下來。

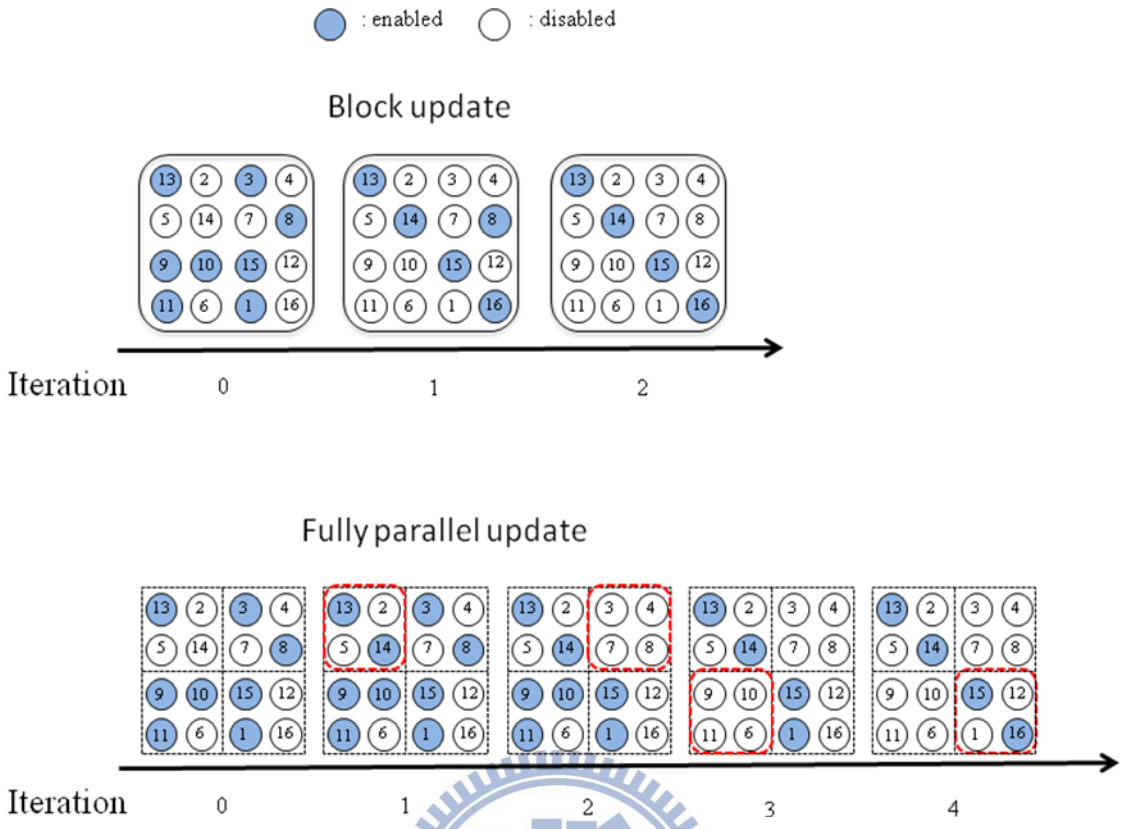
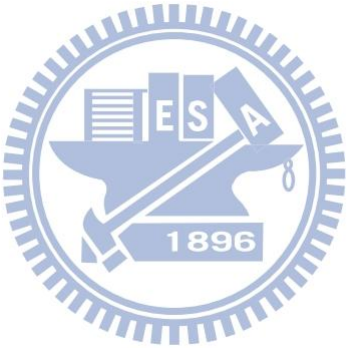


圖 2-9 規律(二)等級安排全平行及分塊更新

在規律一中，因為除了 x_1^1 之外的每一個 x_1^i，在 x_1^j（對所有的 $j<i$）中都有等級比它高，且和它又在同一個碰撞集合內的神經元，因此我們永遠需要等 x_1^{i-1} 壓抑掉其它同碰撞集合的神經元後，才能激發 x_1^i。相反的，如果我們的 $x_1^1, x_1^2, x_1^3, x_1^4$ 剛好也是整個模型中等級最高的四個神經元，那麼就可以省去先壓抑其它同碰撞集合神經元的動作，直接花一個更新週期，就可以同時激發 $x_1^1, x_1^2, x_1^3, x_1^4$。而在一般問題中，如果我們可以對神經元等級分佈的安排，做一些適當的調整，讓它比較接近規律二的等級分佈方式，那麼這樣就可以大大的降低我們達收斂狀態所需的更新週期。

經由上述討論的幾個，可以降低收斂所需更新週期的因素，我們可以經由適當的設定，大大的降低 SRNN 模型達收斂所需的計算量。因為隨著我們模型內的神經元個數的增加，整個模型的計算量其實是以平方在增長的 $(O(n^2))$，因為每當我們增加了一些神經元，不只模型本身的神經元增加了，連每個神經元要計算的對象也增加了。因此如果我們可以透過一些調整，讓我們節省幾個更新週期，那麼對整個模型收斂所需的時間是可以降低不少的。另外，雖然我們在前面的探討中，都有提出真的因為這些因素，而降低

收斂所需更新週期的例子，但還是要在最後重申一次，因為這些因素之間事會彼此影響的，因此當我們只是針對某些特定的因素做出適當的設定時，只是提高用較低的更新週期，就達到收斂的機率。



3 分波多工光相位排列交換器互連系統(WOPIS)

為了有效地連結大量的處理器和伺服器，在高性能計算（High Performance Computing, HPC）系統和數據中心（Data Center, DN）的網路中，我們需要提供高頻寬，大規模，和低延遲的連結網路。而分波多工(Wavelength-Division Multiplexing, WDM)的光連結網路，剛好提供了高頻寬和低功耗的特性，因此最近一直被視為支援 HPC 和 DN 系統的最好人選。而在接下來的內容裡，我們會把問題專注在一個 WDM 的光相位排列交換器(Optical Phased-Array Switch, OPAS)互連系統(Interconnect System, IS)(WOPIS)的封包排程上。在這個章節的開始，我們會先簡介整個系統的架構，及各個硬體的特性及限制，最後再將這些特性及限制數學化，定義出我們的排程問題。

3.1 硬體架構

WOPIS 主要由兩個子系統組成(參考圖 1-1)：光交換器(optical switch)，和 SRNN 排程器(scheduler)。光交換器子系統包括四個部分：輸入光纖(input fibers)，三階光相位排列交換器（Three Stage Optical Phased-Array Switch, TSOPS），光纖延遲緩衝區(Fiber Delay Line Optical Buffer, FOB)，及輸出光纖(output fiber)。

在輸入光纖的部分，有 N 條輸入光纖每條有 W 個輸入波長(圖 1-1 的 N 和 W 分別為四和六)，在經過解多工器(demultiplexer)後，可調光波長轉換器（Tunable Optical Wavelength Converter, TOWC）會將每個封包的外部輸入波長轉換為在輸出光緩衝區沒被佔用的內部波長(internal wavelength)。

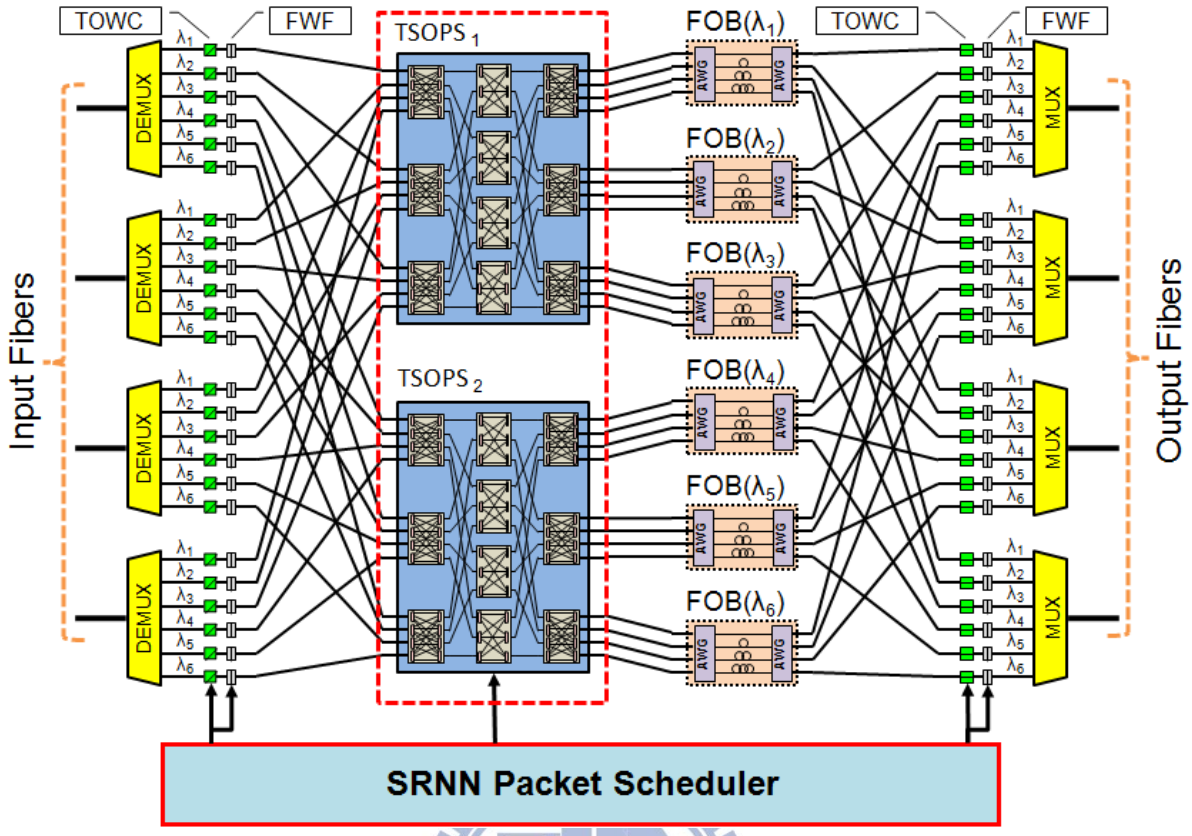


圖 3-1 WOPIS 系統架構圖

在 TSOPS 部分，每個 TSOPS 是負責交換由一個特定內部波長集群所附載的封包 。為了方便說明，該系統如圖 3-2 只有畫出單一的集群。圖 3-2 可以用來說明一個 12×12 的 TSOPS 是如何組成的。總共有 10 個 OPAS 交換器，在第一和最後一個階段(S_1 及 S_3，且 $S_1 = S_3$)分別有三個 4×4 的 OPAS 交換器，而中間階段則有四個 3×3 的 OPAS 交換器。值得注意的是，在 S_1 及 S_3 的 OPAS 交換器數量必須要和輸入(輸出)的波長數目相同，再來每個 $n \times n$ 的 OPAS 交換器(圖 3-2 右上角為一個 4×4 的 OPAS 交換器)是由 n 個 $1 \times n$ 的 OPAS 交換器元件，完全連接到其它 $n \times 1$ 的 OPAS 交換器元件所組成，且由於 OPAS 擁有波長選擇性(wavelength selective)，所以由不同波長所附載的封包抵達 $1 \times n$ 的 OPAS 交換器元件後，將會從不同的輸出端輸出。反過來說，當多個封包同時要經過一個 $n \times 1$ 的 OPAS 時，必須每個封包的載波波長都不同，否則就會有碰撞發生。

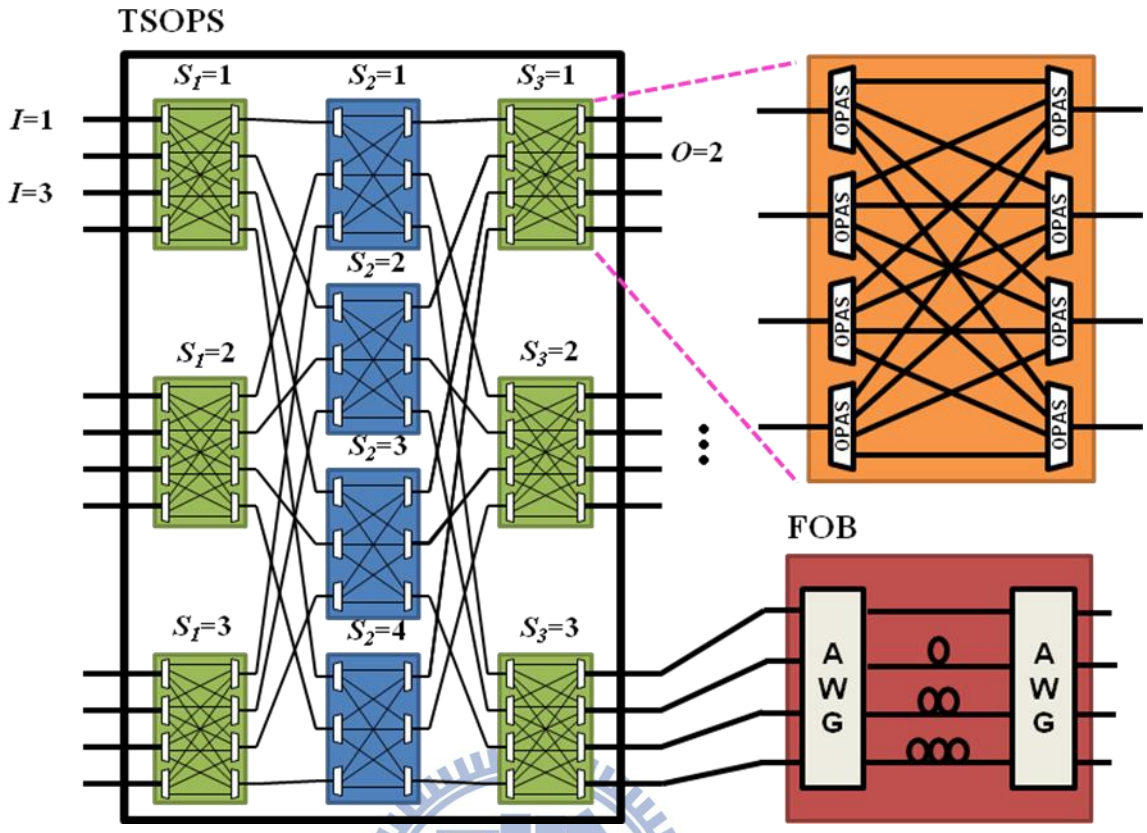


圖 3-2 TSOPS 架構圖及 FOB 架構圖

而在輸出緩衝區中，每個輸入波長都有一個由光纖延遲線(Fiber Delay Line, FDL)所組成的光緩衝區(FDL Optical Buffer, FOB)，且每個 FOB 又被每一個輸出端所共享。一個 FOB 是由一組陣列波導光柵（Arrayed Waveguide Gratings, AWG）並利用 F 條 FDL 連結所組成，所以每個 FOB 都擁有 $F \times L$ 個緩衝區位置(包含沒有延遲的位置)，其中 L 代表內部波長的數量。更具體地說，AWG 是純粹的被動元件且在 WDM 系統中常被當作光(解)多工器。正如圖 3-2 右下角所示，一個 4x4 的 AWG ，每一個輸入端可接收四個使用不同波長載波的封包，(如 λ_{11}、λ_{12}、λ_{13}、λ_{14} 代表從第一個輸入端分別使用四個不同波長載波的封包)，且路由(route)不同波長的封包到不同的輸出端，路由的規則如圖 3-4 示。值得一提的是，當封包連續通過兩個 AWG 時，一個從第一個 AWG 的第 i 個輸入端進入的封包，在由內部波長決定一個延遲後，將也會從第二個 AWG 的第 i 個輸出端離開緩衝區，因此對於任何一個 FOB，一個封包所使用的內部波長，將會成為決定它在 FOB 內延遲時間的唯一要素。

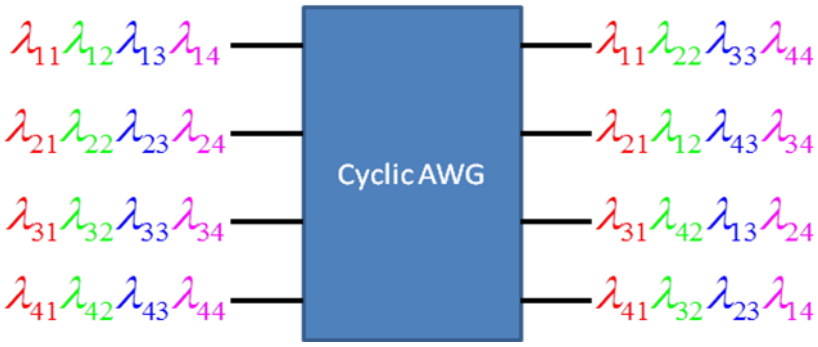


圖 3-4 AWG 路由規則

至於輸出部分，總共有 $N \times W$ 個 TOWCs，分別代表 N 條光纖的 W 個波長。在任何 FOB 的第二個 AWG 的輸出端，當出現多個封包試圖同時從同一個輸出端離開時，就會發生緩衝區競爭(buffer contention)。

最後是 SRNN 排程器的部分，它負責安排新來的封包一個安全且最低延遲的路徑，不只要可以完全避免交換器和緩衝區的競爭，且要同時達到整體系統最高吞吐量(throughput)及符合封包間的優先差別(priority differentiation)。

3.2 參數定義

在本節中我們要將上一節中各個硬體特性和發生競爭的情形，還有每個硬體相對應的參數，列出說明和做一些定義，並在接下來整篇論文中使用。

令 N 表示輸入(輸出)光纖的總數，L 表示內部波長的數量，W 和 K 分別表式 TSOPS 第一(三)階段和第二階段的交換器數量。參數 $I(O)$表示在交換器輸入(輸出)端的編號，且 I,O =1 to N。參數 s_1(s_3)則表示在 TSOPS 第一(第三)階段的交換器編號，且 s_1, s_3=1 to W。參數 s_2 則表示 TSOPS 第二階段交換器的編號，且 s_2 = 1 to K。參數 λ 表示內部波長的編號，且 λ=1 to L。而 F 則表示每個 FOB 有多少條 FDLs，如圖 3-1 架構，N=4，W=6，K=8，F=4。

有了以上的硬體架構參數，我們就可以來定義一些接下來會一直用到的排程參數。

[定義一]一條使用內部波長 λ 的路徑 $\mathbf{P}(I, s_1, s_2, s_3, O, \lambda)$，代表一條從第 s_1 個 TSOPS 的第一階段的交換器的第 I 個輸入端進入，並通過第 s_2 個 TSOPS 的第二階段交換器，最後從第 s_3 個 TSOPS 的第三階段的交換器的第 O 個輸出端輸出。一條路徑的狀態我們用參數函數 I_p 來表示，I_p=1 表示路徑 **P** 正被一個封包所佔用，反之則用 I_p=0 來表示。

[定義二]一個合理路徑集合 U，仔細來說就是所有符合 FOB 規則(任何一個封包從 FOB 的第一個 AWG 的第 i 個輸入端進入，就必須要從第二 AWG 的第 i 個輸出端輸出)的路徑，集合的定義如下：

$$U \equiv \{\mathbf{P} \mid ((\lambda-((\lambda-O) \bmod F+F) \bmod F) \bmod N+N) \bmod N = O\}$$

除此之外我們還要再定義幾個接下來會用到的函數：第一個是 d(**P**)，用來表示路徑 **P** 在緩衝區的延遲時間，且這個函數只和路徑的輸出端 O 還有內部波長 λ 有關($d(P) = ((\lambda - O) \bmod F + F) \bmod F$)。另一個是 b(**P**)，用來表示路徑 **P** 在 FOB 內緩衝區位置的狀態，且這個函數只和路徑 **P** 的 TSOPS 第三階段的交換器 s_3，還有輸出端 O 及內部波長 λ 有關。此外當 b(**P**)=1 時，就代表路徑 **P** 所通過的緩衝區位置已經有人佔用，反之則用 b(**P**)=0 來表示。

3.3 路徑競爭及硬體條件

有了上一節的參數和函數定義，在這一節內，我們就可以將交換器系統內，所有可能發生競爭(contention)的條件，及特殊的硬體條件都給數學化。其中包含了 TSOPS 內的路徑競爭、OPAS 的波長選擇性、緩衝區內的路徑競爭，及最後的單一分配及封包優先權條件。

3.3.1 TSOPS 內的路徑競爭及 OPAS 的波長選擇性

首先，在交換器最常發生競爭的情形就是，當兩個以上使用相同內部波長的封包，在 TSOPS 內同時要通過相同的路徑時所發生的競爭，因此連同 OPAS 的波長選擇性，要在 TSOPS 內達到毫無競爭的路徑排程，我們必須符合接下來的兩條規則。

第一條規則限制我們，當使用相同內部波長的封包從不同的入口進入交換器元件時，不可以同時被指定到相同的交換器元件出口，圖 3-5 舉出了三種可能的情況（相同的顏色代表不同的封包使用相同的內部波長），分別為紅色，綠色，藍色。第二條規則限制我們，由於 OPAS 的波長選擇性，當使用不同內部波長的多個封包同時進入 OPAS 元件時，必須被指定到不同的輸出，圖 3-5 右上角舉了一個違反規則的例子。基於以上兩條規則，要在 TSOPS 內達到毫無競爭的路徑排程，必須滿足六條式子 $R_1 \sim R_6$，其中 $R_1 \sim R_3$ 是為了避免違反第一條規則，而 $R_4 \sim R_6$ 則是為了避免違反第二條規則。

接下來為了方便表示這些路徑間的競爭情形，我們將定義一個新的集合及函數，定義如下：

[定義三]一個和路徑 **P** 因為違反規則 g 而產生的路徑碰撞集合，我們將他定義為 G_P^g 且 $P \notin G_P^g$

又可表示為：

$G_P^g \equiv \{P' \in U - \{P\} \mid C(g)\}$，$g$=1~6，且 $C(1) \sim C(6)$ 可表示如下：

$$C(1) = [(S_1', S_2', \lambda') = (S_1, S_2, \lambda)] \qquad C(2) = [(S_2', S_3', \lambda') = (S_2, S_3, \lambda)]$$
$$C(3) = [(S_3', O', \lambda') = (S_3, O, \lambda)] \qquad C(4) = [(I', S_1', S_2') = (I, S_1, S_2)]$$
$$C(5) = [(S_1', S_2', S_3') = (S_1, S_2, S_3)] \qquad C(6) = [(S_2', S_3', O') = (S_2, S_3, O)]$$

其中規則 $C(1) \sim C(3)$ 是違反上述第一條規則的條件，而 $C(4) \sim C(6)$ 則是違反 OPAS 的波長選擇性的條件，舉例來說，在圖 3-5 的紅色範例中，因為路徑 P 和 P' 擁有相同的 S_1, S_2 及 λ，符合 $C(1)$，所以 $P' \in G_P^1$ 或 $P \in G_{P'}^1$，因此當路徑 P 和 P' 同時被選取時，規則一(R_1)就會被違反而發生路徑競爭。而右上角的例子，則是因為路徑 P 和 P' 擁有相同的 I, S_1 及 S_3，符合 $C(4)$，所以 $P' \in G_P^4$ 或 $P \in G_{P'}^4$，因此路徑 P 和 P' 不能同時存在，否則就違反了 OPAS 的波長選擇性。而新的函數 $\eta(G_P^g, P') =$ 則表示 $P' \in G_P^g$，反之 $\eta(G_P^g, P') = 0$，換句話說，也就是當路徑 P' 和 P 同時被選取或存在時，會違反某一條規則 g，函數 $\eta(G_P^g, P') = \eta(G_{P'}^g, P) = 1$。而上續的六條為了達到毫無競爭的路徑排程的式子也可統一被表示成：

$$R_1 \sim R_6 : \sum_{P \in U} \sum_{P' \notin U} I_P I_{P'} \times \eta(G_P^g, P') = 0, \text{ for } g = 1 \sim 6$$

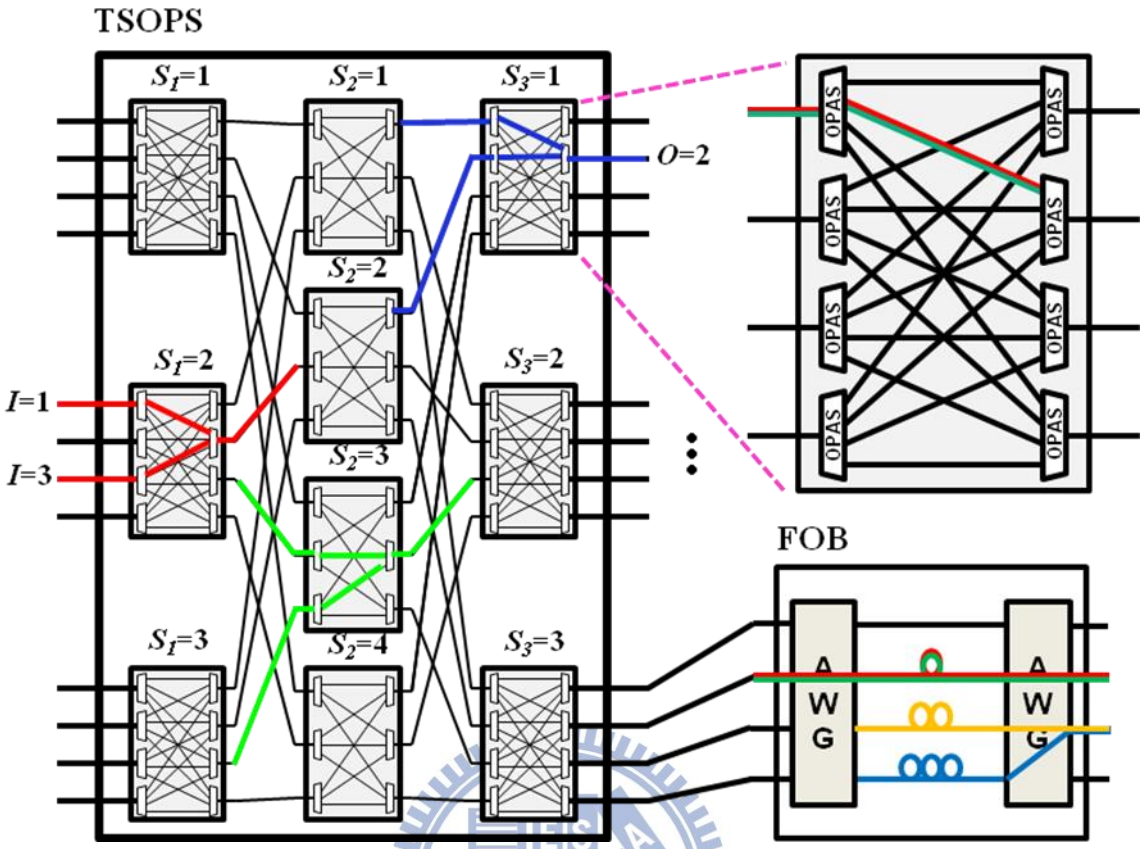


圖 3-5 路徑競爭說明圖

3.3.2 緩衝區內的路徑競爭

在看完 TSOPS 可能發生路徑競爭的條件後，我們要在把專注力放到緩衝區上。一般來說，緩衝區的路徑競爭都是因為當有兩個以上的封包，都企圖從同一個 FOB 內的第二 AWG 的輸出端離開時所造成的。而我們又可以將它再細分為兩種情況：一種是由兩個新來的封包所造成的，而另一種則是由一個新來的封包和一個已經在緩衝區內的封包所造成。接下來我們將分別提出規則 R_7 和 R_8 來避免以上的兩種路徑競爭。值得注意的是，規則 R_7 只有當 FOB 內的 FDL 數量小於輸入(輸出)的光纖數量時才會發生，也就是當 $F<N$ 時。我們分別將 R_7 和 R_8 表示為：

$$R_7: \sum_{P\in U}\sum_{P'\notin U} I_P I_{P'} \times \eta(G_P^7, P') = 0, \quad G_P^7 \equiv \{P' \in U - \{P\} \mid (S_3', O', d(P')) = (S_3, O, d(P))\}$$

$$R_8: \sum_{P\in U} b(P) \times I_P = 0$$

由式子可以很明顯的看出，R_8和$R_1 \sim R_7$有一個很不一樣的地方，就是它僅和路徑所通過的緩衝區位置有關，和其它參數皆無關。有了以上的八條規則$R_1 \sim R_8$，我們就可以定義出一條毫無路徑競爭(Contention Free, CF)的條件:

$$C_{CF} : \sum_{g=1}^{7} \sum_{P \in U} \sum_{P' \notin U} I_P I_{P'} \times \eta(G_P^g, P') + \sum_{P \in U} b(P) \times I_P = 0 \quad (1)$$

3.3.3 單一分配及封包優先權條件

如果只是單純為了滿足 CF 的條件，那麼一個封包可以選擇的路徑，可能從沒有到多個都有可能。但如果一個封包可以同時佔用多條路徑，那麼系統的整體吞吐量(throughput)就會因此而被降低了。因此，為了避免這樣的情形，我們希望每一個封包都只佔用一條路徑就好，另外又訂了一個條件稱為單一分配(Single Assignment, AS)：

$$C_{SA} : (\sum_{S_2, S_3, \lambda} I_P) - n(I, S_1, O) = 0 \quad (2)$$

也就是說當有路徑需求的時候，我們才指定一條路徑來滿足需求。其中函數$n(I, S_1, O)$=1，代表有一個從第S_1個第一階交換器的第I個輸入端的新進封包，要求從第 O 個輸出端輸出，換句話說也就是一個輸入(輸出)要求(I/O request)，反之$n(I, S_1, O)$=0。

到目前為止，我們已經說明了兩種一般限制 CF 和 SA，接下來我們將把優先權考慮進去，再提出兩種優先順序的限制。第一條為高優先權優先(Priority-First, PF)，限制同一批(batch)新來的封包，優先權較高的封包不會因為優先權較低的封包而限制了路徑的選擇，換句話說，假如當兩個同一批新進封包發生路徑競爭時，低優先權的必須讓給高優先權的封包。另外，當我們只考慮一批封包時，只要符合 SA 就可以滿足吞吐量的最佳化，但在真實情況，封包都會是一批接著一批的進來，而整個系統的吞吐量就會隨著緩衝區延遲下降而上升，因此我們提出第二條優先權限制：最小延遲(Minimum Delay, MD)，限制如果有較低緩衝區延遲的路徑可以使用，那我們就不能選擇高緩衝區延遲的路徑。

有了以上四條限制，我們就可以定義我們的有優先權的路徑排程問題：

[優先權排程問題定義]對於多批連續抵達 WOPIS 的封包，我們必須對每一批封包決定出一個路徑的集合，符合上述的兩條一般限制 CF 和 SA 及兩條優先權限制 PF 和 MD。換句話說，我們最大的挑戰就在於，我們必須達到毫無競爭且擁有優先順序的路徑排程，且同時實現吞吐量的最大化。

3.4 SRNN 處理 WOPIS 上有優先權路徑排程問題

為了能夠利用 SRNN 模型來解決 WOPIS 上的有優先權路徑排程問題，我們將每個神經元的輸出 V_P 都被對應的路徑指標 I_p ，其中 $P=(I,S_1,S_2,S_3,O,\lambda)$ ，而 V_P=1 則代表路徑 P 目前有被選取，而神經元間的連接權重和神經元本身的門檻我們則表示為 $W'_{PP'}$ 和 θ_P ， $P'=(I',S'_1,S'_2,S'_3,O',\lambda')$ 。

接下來我們先考慮之前式子(1)中的 CF 規則，而我將 CF 相關函數的定義如下：

$$f_{CF}^{(k)}=\sum_{g=1}^{7}\sum_{P\in U}\sum_{P'\in U}\eta(G_P^g,P')V_P^{(k)}V_{P'}^{(k)}+\sum_{P\in U}b(P)V_P^{(k)}.$$

再來是加入 SA 規則，但為了讓函數恆正，我們將原本的函數 $[\sum_{S_2,S_3,\lambda}V_P^{(k)}]-n(I,S_1,O)$ 取平方後，定義我們的 SA 相關函數如下：

$$f_{SA}^{(k)}=\sum_{P\in U}\sum_{P'\in U}\eta(G_P^8,P')V_P^{(k)}V_{P'}^{(k)}+\sum_{P\in U}[1-2n(I,S_1,O)]V_P^{(k)}+\sum_{I,S_1,O}n^2(I,S_1,O),$$

其中 $G_P^8\equiv\{P'\in U-\{P\}\mid(I',S'_1,O')=(I,S_1,O)\}$ ，且利用定理，我們可以得到相對應的神經元間權重和神經元本身門檻表示如下：

$$\begin{cases}W_{PP'}=-\sum_{g=1}^{7}2\eta(G_P^g,P')-2\eta(G_P^8,P')\\ \theta_P=-b(P)-[1-2n(I,S_1,O)]\end{cases}\tag{5}$$

接下來，我們要再加入優先權的考慮，在我們的神經元間權重和神經元本身門檻內加入等級激發項。首先我們先決定每個封包的優先權 $pr(P)$ 和每個緩衝區的延遲 $d(P)$ ，因為在定理中我們只有一個等級函數(rank function)，因此我們決定先滿足封包的優先權再滿足最低緩衝區延遲，並把它們合併成一個等級函數。再來，為了讓每個神經元都擁有不同的等級，我們在函數的最後再加上一個隨機實數 $\delta(0\le\delta<Num)$ ，最後我們每條路徑 P 的等級函數就可

以表示成：$r(P) = -[pr(P) \times Num \times F + d(P) \times Num + \delta]$，其中 F 為 WOPIS 內的 FDL 的個數，Num 為神經元的總數(交換器的總路徑數)，最後由上式和定理我們可以表示神經元間權重和神經元本身門檻為：

$$\begin{cases} W_{PP'} = \sum_{g=1}^{8} -2\rho_g u[r(P') - r(P)]\eta(G_P^g, P') \\ \theta_P = \rho_9 \times b(P) + \rho_8[1 - 2n(I, S_1, O)] \end{cases} \tag{6}$$

其中 $\rho_1 \sim \rho_9$ 為常數參數，後面我們會再說明如何決定它的值。根據定理，一條路徑要能夠成為路徑候選人，必須符合：(1)神經元本身門檻必須小於等於零($\theta_P \le 0$)，(2)路徑必須存在($n(I, S_1, O) = 1$)，且(3) 路徑所通過的緩衝區空間必須沒有封包佔用($b(P) = 0$)。而對應到式子(6)的第二式，我們可以得到下列四條不等式：

$$\begin{cases} \theta_P \mid_{n(I,S_1,O)=1 \text{ and } b(P)=0} = -\rho_8 \le 0; \\ \theta_P \mid_{n(I,S_1,O)=0 \text{ and } b(P)=1} = \rho_8 + \rho_9 > 0; \\ \theta_P \mid_{n(I,S_1,O)=0 \text{ and } b(P)=0} = \rho_8 > 0; \\ \theta_P \mid_{n(I,S_1,O)=1 \text{ and } b(P)=1} = -\rho_8 + \rho_9 > 0. \end{cases} \tag{7}$$

而從式子(7)中，我們可以發現 $\min_P\{\theta_P\} = -\rho_8 > 2\rho_g$。另外，根據定理 (規定 $\theta_P > -2\rho_g$ 對所有的路徑 P，且 $2\rho_g > 0$ $g = 1 \sim$)，我們可以再把條件簡化為 $-\rho_8 > 2\rho_g$。為了我們的 SRNN 路徑排程最後能夠收斂到穩定狀態，我們必須將我們的神經元間連接權重/神經元本身門檻的參數設計為:

$$2\rho_g > \rho_8 > 0 \text{ for all g=1~7, and } \rho_9 > \rho_8 > 0.$$

在這樣的設計下，那些擁有較高等級(較高的封包優先權及較低的緩衝區延遲)的路徑將會被激發，而其它在相同 CF 及 SA 碰撞集合的路徑，則會被壓抑下來。

4 計算統一設備架構(CUDA)

計算統一設備架構(Compute Unified Device Architecture，CUDA)是一個平行化的程式模型，是為圖形處理器(Graphic Processing Unit，GPU)計算設計的硬體和軟件框架。透過提供各種常用的程式語言，如 C, C++, Fortran 等的衍伸語法，來讓程式設計者很快的學習，而不需使用繪圖專用的程式語言(如: OpenGL)進行開發。現在的 NVIDIA GPU 皆包含必要的硬體組件，靈活的控制和分配到數百個處理器內核的工作量。而再詳細介紹 CUDA 之前，我們將先比較一下，GPU 和中央處理器(Central Processing Unit，CPU)的設計概念，以說明為何我們選擇 CUDA 來做為我們平行化的工具。

4.1 圖形處理器和中央處理器比較

GPU 主要是以多個流處理器(Streaming Processor，SP)所組成的平行系統，目前流處理器的數量，仍隨著莫耳定律快速地成長。面對市場上對快速運算的需求，發展成為高平行化、多執行緒(multithreaded)、多核心(multi core)處理器架構，具有非常高的運算能力及非常大的記憶體頻寬，如圖 4-1 及 4-2。

而 GPU 比 CPU 擁有更強浮點運算能力的背後原因是，GPU 是專門為計算密集，高度並行的計算所設計的，如圖形處理。因此在硬體的設計上，更多的晶體管用於數據處理(data processing)而非資料快取(data caching)或控制流程(flow control)，參考圖 4-3。更精確的來說，GPU 特別適合解決資料平行(data-parallel)的問題：多個資料元素(data element)平行的執行同樣的程式。因為同樣的程式在同一時間運算多個不同的 data element ，只需要較少的複雜控制流程(sophisticated flow control)，也因為算術運算的比例遠大於記憶體運算，因此記憶體的存取延遲可以透過大量的運算元件而非資料快取而被隱藏。

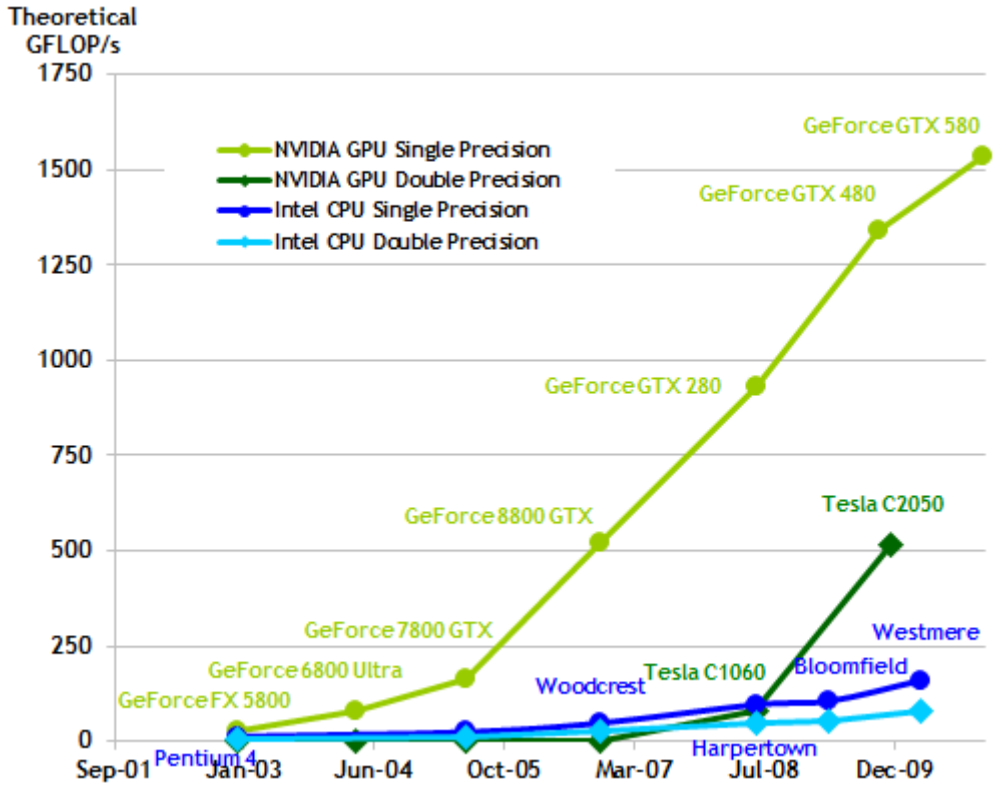


圖 4-1 CPU 和 GPU 運算能力成長圖

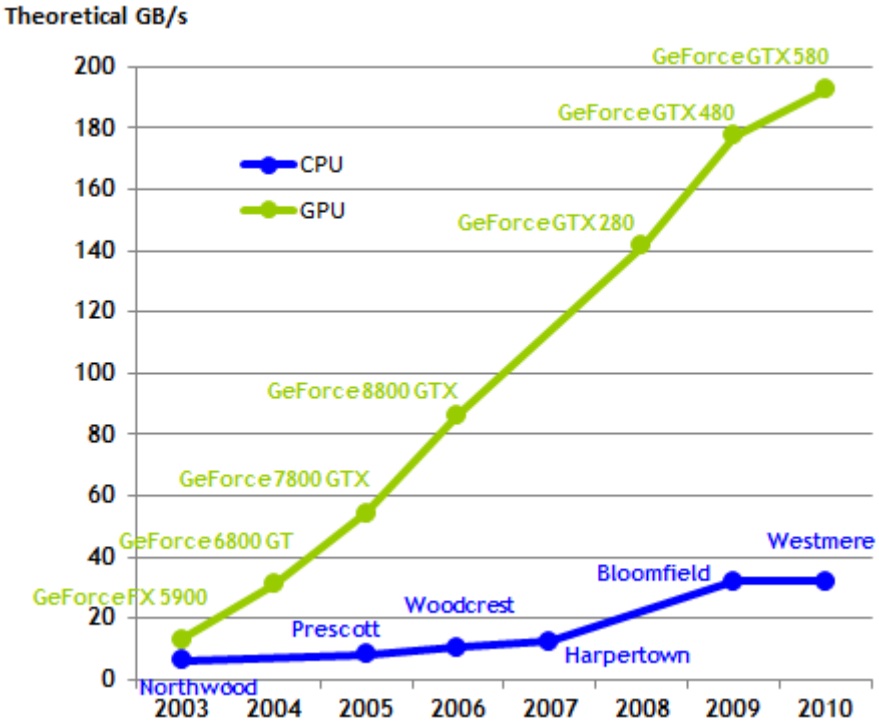


圖 4-2 CPU 和 GPU 記憶體頻寬成長圖

當然，並不是不能在 CPU 上執行平行化的程式，只是目前市售的多核心 CPU 核心數太低，如果想要執行平行度較大，且真的達到平行的效果，就必須還要透過像訊息傳遞介面(Message Passing Interface，MPI)，來幫助我們利用網路連結多台電腦的處理器，已達到擴充處理核心數目的目的。可是在這樣的架構下，對於不同執行續需要密切溝通的程式，通常平行化的優點會因為資料傳遞太緩慢，而沒辦法顯現出來。相反的，這點在 GPU 上就獲得了很大的改善，因為 GPU 本身就擁有大量的 SP(Ex:GTX580 擁有 512 個 SP)，所以所有的執行緒只需要在同一顆 GPU 上執行及可，因此執行緒間的溝通完全不需要透過網路的傳遞，只需要透過 GPU 上的全域記憶體(global memory)溝通及可。

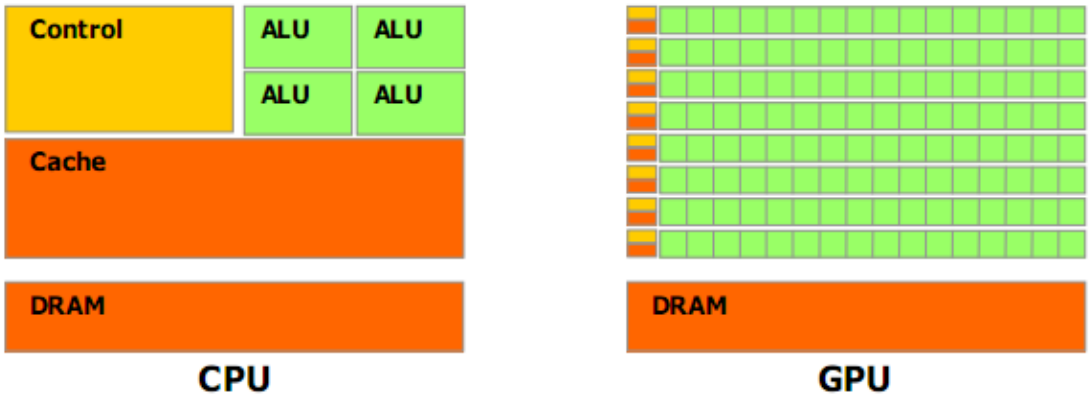


圖 4-3 CPU 和 GPU 硬體設計概念比較

4.2 硬體架構

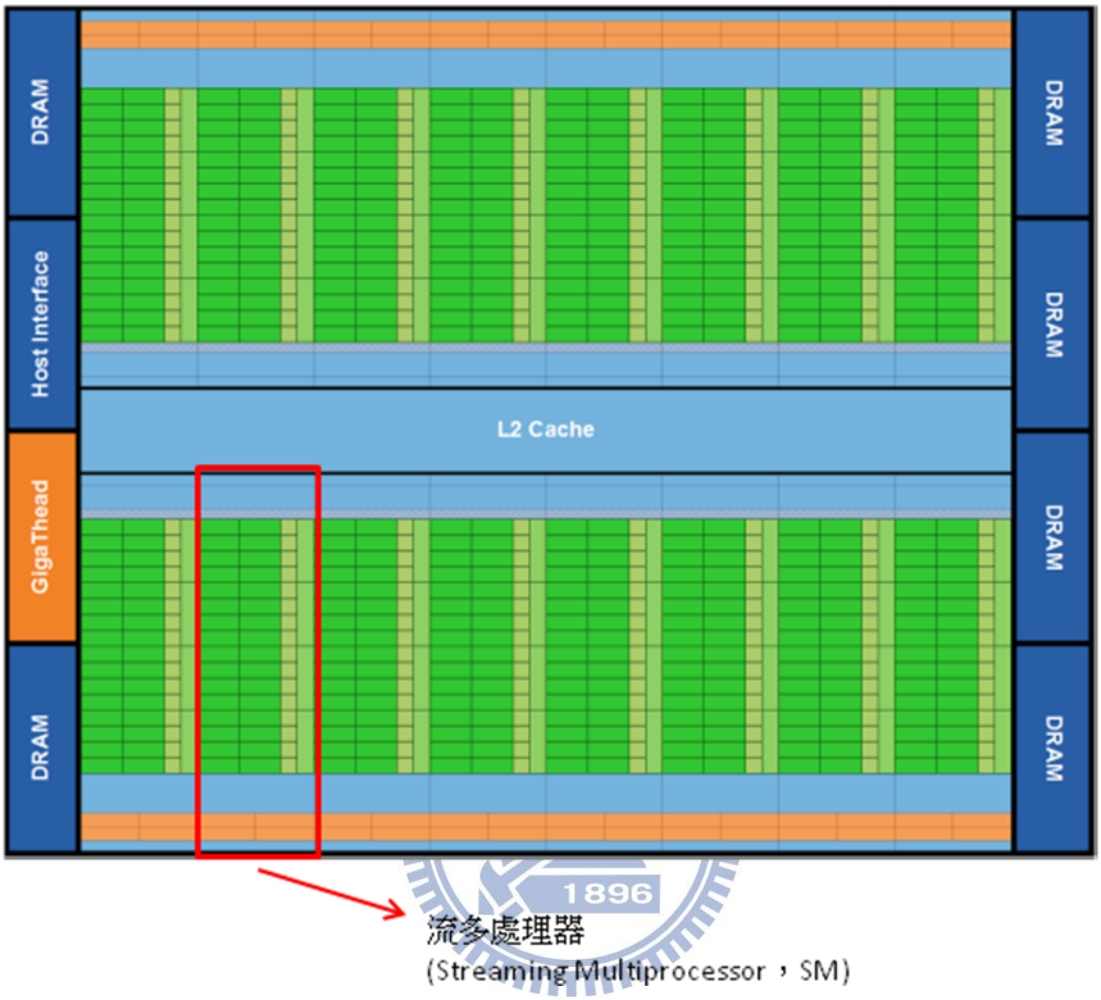


圖 4-4 CUDA Fermi 架構 GPU

和舊有的GPU 相比，CUDA 將部分傳統獨立的繪圖專用硬體，整合到SP內，相較於傳統的繪圖硬體電路，SP開始具有通用運算的能力，且在最新一代的NVIDIA GPU架構中(參考圖4-4)，還在GPU內加入了晶片內的L2及L1快取。以GTX 580來說，它擁有768KB的全域L2快取，及最高每個流多處理器(Streaming Multiprocessors，SM，如圖4-4)擁有48KB的L1快取。還有十六個SM，而每個SM內又包含了三十二顆SP。而這些SPs 會在同一時間，執行同樣的程式，不同的資料流，稱為SPMD(Signal Program Multiple Data Architecture)。

每個 SM 擁有五種不同的晶片內部記憶體(on-chip memory)：

- 一組可調大小(48KB/16KB)的L1 Cache，若L1 Cache為48KB則共享記憶體為16KB，反之亦然。
- 一組可調大小(48KB/16KB)的共享記憶體。在同一個Block中的所有Threads可以透過共享記憶體來通訊，而不需要經由晶片外部的記憶體。
- 一個唯讀的常數快取記憶體(read-only constant cache memory)，對於一維資料的讀取有加速的作用。
- 一個唯讀的紋理快取記憶體(read-only texture cache memory) ，對於二維資料的讀取有加速的作用。
- 32768個32-bit的Register

除此之外，SM 內還包含了兩個 Warp (由三十二個 Threads 所組成，是 CUDA 程式執行排程的最小單)排程器(scheduler)，分別控制十六個 SPs，且提供同時兩個 Warps 的執行及排程。另外，還有十六個記憶體存取(Load/Store) 元件及四個特殊函數元件(Special Function Unit，SFU)。而如果我們再將每個 SP 切開來看，裡面又包含了一個浮點(Float Point，FP)及一個整數(Integer，INT) 的計算元件，還有一個指令控制元件。

圖 4-5 SM 內部架構

4.3 程式設計模型

CUDA 程式執行時，是將 GPU 是為協同處理器(coprocessor)，主要程式還是在 CPU 端上執行，只要執行到需要平行化的部分，再利用呼叫函數的方式，將需要平行執行的程式碼寫成一個函數(在 CUDA 中稱為 kernel)，並載入到 GPU 內由不同的執行緒平行執行。而程式的設計模型主要包含三個概念：其一為層階式的執行緒群組， 其二為層階式記憶體，其三為同步(barrier synchronization)。第一種概念指的是 CUDA 程式執行緒模型由上到下分為 Grid、Block、Thread， 三種階層，一個 Grid 包括了多個 Blocks，一個 Block 又包含了多個 Thread。同一個 Grid 內的 Threads 會執行相同的核心函數(kernel function)，而同一個 Block 內的 Threads 則會被分配到同一個 SM 內執行。程式設計者必須善用這種階層關係，妥善分配每一層的比例，以達到

資源使用及程式執行效率的最佳化。第二種概念是配合第一種概念，CUDA 在不同階層的執行緒，提供不同可使用的記憶體，如各個 Thread 有自己的區域性記憶體(local memory)，而 Block 內的 Thread 有共享記憶體(Shared Memory，SMEM)能共用，以減少對外部記憶體的存取，而同 Grid 但不同的 Block 要做通訊，就必須利用在晶片外部的(off-chip)全域記憶體(Global Memory，GMEM)。除此之外，CUDA 還依資料的維度特性不同提供兩種唯讀的全域記憶體快取空間，分別為加速一維資料讀取的常數記憶體(Constant Memory，CMEM)，及加速二維資料讀取的質地記憶體記憶體(Texture Memory，TMEM)。第三種概念主要是程式必須由設計者來做同步，以避免資料不一致的問題，而 CUDA 提供了兩種同步的機制，一種是能同步同一個 Block 內的 Threads ，另一種則是能提供 CPU 端(在 CUDA 上稱為 Host)和 GPU 端(在 CUDA 上稱為 Device)間的同步。以上三點在後面的章節內會再一一仔細說明。

4.3.1 階層式 Thread

一個 CUDA 程式，包含了序列(series)部分及平行(parallel)部分。我們希望平行部分由 GPU(Device)端來執行，而序列部分則仍由 CPU(Host)端執行。CUDA 程式是藉由在序列部分的程式中，以呼叫函數的方式，來執行平行部分的程式，而這些被 GPU 平型執行的函數，在 CUDA 中被稱為 kernel。如圖 4-6，GPU 扮演著協同處理器的角色，與 CPU 進行合作，幫它執行需要平行執行的部分。

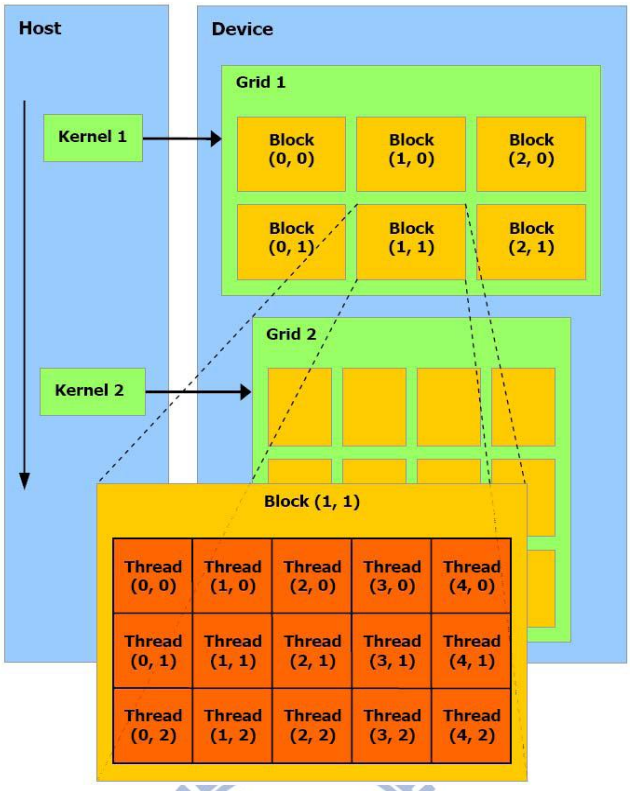


圖 4-6 CUDA 階層式 Thread 示意圖

而 kernel 則透過 Thread 的執行機制來執行。在 CUDA 中每個 Thread 都會被包在一個 Block 內，而在同一個 Block 內的 Threads 可以透過 SMEM 來溝通，並且利用 CUDA 內建的同步函數來達到同步的效果，以避免資料不一致的現象。且每一個 Thread 在 Block 內都有自己的一個 ID，使用者可以利用一個一到三維的索引來標示每個 Thread 的 ID，舉例來說：一個三維的 Block 三個維度的大小分別為 Dx、Dy，和 Dz ，那麼一個索引為(x,y,z)的 Thread ID 就可表示為 $x + y \times Dx + z \times Dx \times Dy$ 。且一個 Block 內最多可以包含 1536 個 Threads。

然而多個執行相同 kernel 的 Block 都會被包在同一個 Grid 內，也就是說，在同一個 Grid 內的所有 Threads 都會執行相同的 kernel，且同一個 Block 內的 Threads 都會被分到同一個 SM 內執行，而不同 Block 內 Threads 則不能存取彼此 Block 的 SMEM，只能透過全域記憶體做為溝通的橋梁。

而和 Thread 一樣，每個 Block 在 Grid 內也有一個自己的 ID，使用者一樣可以利用一到三維的索引來標示每個 Block 的 ID，而標示的方式在上面 Thread 時已經提過了，所以在這就不再多述。

4.3.2 階層式記憶體架構

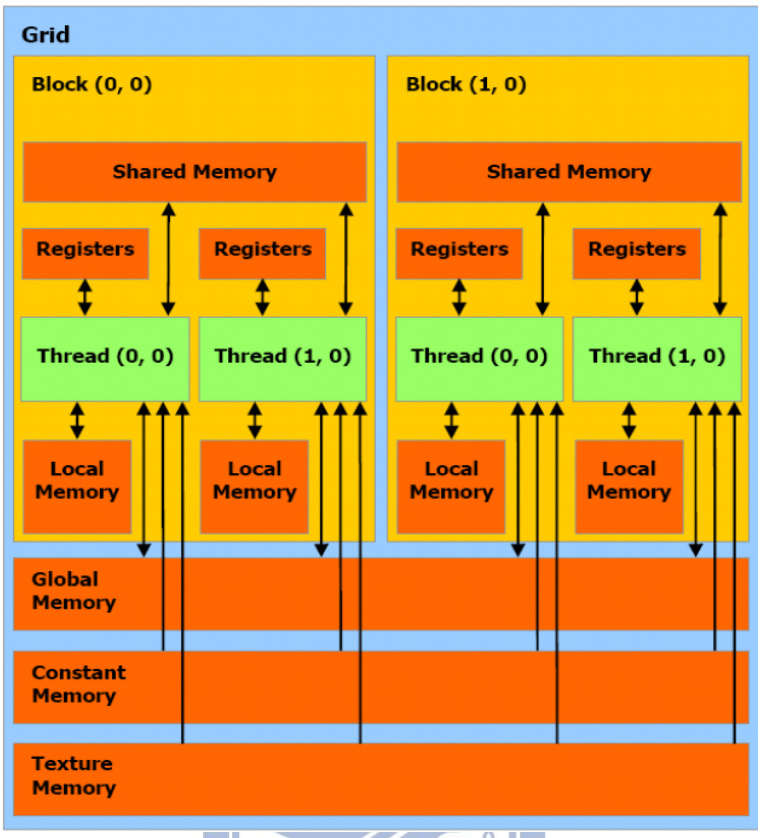


圖 4-7 CUDA 階層式記憶體架構示意圖

CUDA Thread 可以從多種不同的記憶體存取資料，且每一種記憶體都有它不同的特性及使用方法，如圖 4-7。每一個 Thread 有自己私有的記憶體(Local Memory)及暫存器(Register)。而每一個 Block 內的 Threads，則有一個可以共同存取，作為溝通媒介的 SMEM。

SMEM 在 CUDA 程式最佳化上佔有很重要的地位，我們可以把他視為一種使用者能夠自由控制的 L1 Cache，善用 SMEM 能減少對外部 GMEM 的存取，加快我們記憶體的存取速度。而為了能夠完全達到 SMEM 存取的最快速度，我們必須在存取時避免堤岸衝突(bank conflict)。因為 CUDA 會自動將 SMEM 分割成十六個相同大小的區塊，而每一個區塊都有一個專門的堤岸(bank)，負責存取的控制，且這十六個堤岸是可以同時運作的，換句話說 CUDA 可以一次執行十六個 Threads 對 SMEM 的存取。而堤岸衝突，則發生在當有兩個以上的 Threads，要使用相同的堤岸存取資料時，這些 Threads 變成需要以序列化的方式(一次只能一個 Thread 存取)，存取相同的 SMEM 區

塊，而導致時間上的浪費。但假如是十六個 Threads，同時對同一個堤岸做讀取(read)的動作時，CUDA 則可以用廣播(broadcast)的方式，在一個時間週期(clock cycle)內完成動作，圖 4-8 有各種情形的示意圖。

GMEM 是不同 Block 內的 Threads 互相溝通的唯一管道，因為只要是同一個 Grid 內的 Threads 都可以存取相同的 GMEM 區塊。而在最新一代的 NVIDIA Fermi GPU 架構中，突破性的在 GPU 內加入 GMEM 的 L2 及 L1 快取，雖然存取所需的延遲因此降低了不少，但因為這一部分的快取，不像 SMEM 可以由使用者自由控制，因此還是會有快取錯過(cache miss)的負擔。

另外，SMEM 與 GMEM 都會有資料一致性(data consistence)的問題。為了避免 SMEM 存取問題，使用者必須在 kernel 內，對 SMEM 存取的前(後)，加入 CUDA 提供的 Block 同步函數(__syncthreads())來同步。而對於 GMEM 則要在 Host 端程式使用同步 Host 端和 Device 端的函數(cudaThreadSynchronize())來同步，以確保當 Host 端要使用 Device 端運算的資料時，Device 端已將資料運算完，並且載出到 Host 端來。

除了 SMEM 與 GMEM 之外，CUDA 還另外提供了兩種唯讀記憶體空間：CMEM 及 TMEM。與 GMEM 相同，它們可以被所有的 Thread 讀取。但這兩種記憶體是基於 GPU 在繪圖運算所使用的，和其他記憶體相比，他們具有加速讀取的快取，通常在對 CUDA 程式做最佳化時使用，例如對一些不會在 GPU 內做更動的資料(只會被讀取)，我們就可以依資料的維度分別放入這兩個記憶體內(CMEM 只能存放一維的資料)。

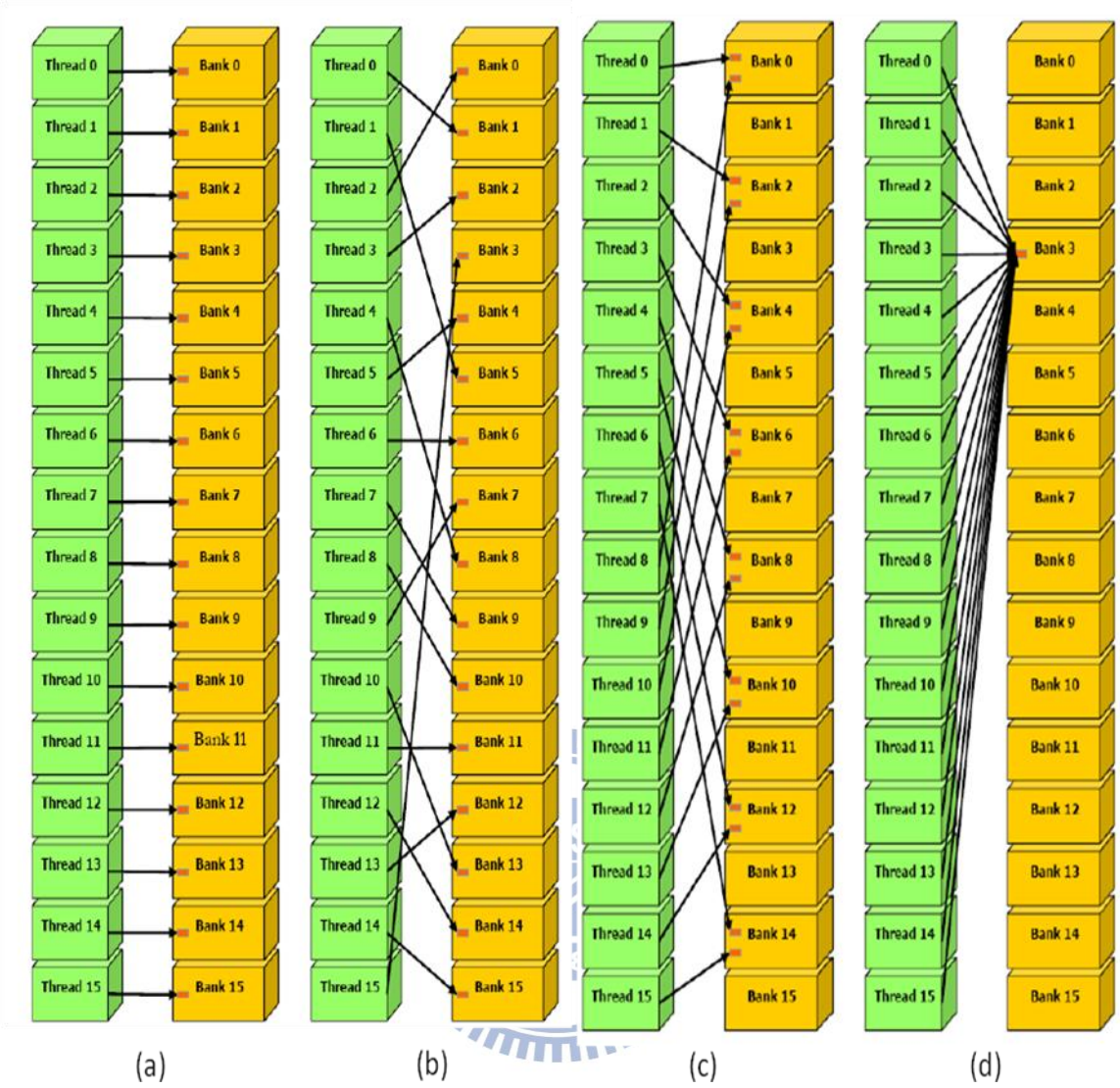


圖 4-8 (a)無堤岸衝突線性存取(b)無堤岸衝突隨及存取(c)有堤岸衝突 (d)廣播讀取

4.3.3 同步

在前面的章節有提到過幾次，CUDA 提供了兩種同步函數，分別可供 Host Device 間使用，及在同一個 Block 內的 Threads 使用。前者我們可以先參考圖 4-9，CUDA 程式執行的流程其實主要還是序列式的(serial)在 Host 端上執行，等到執行到我們需要平行的部分時，再將 kernel 和所需的資料載入到 Device 端內執行，這時在 Host 端假如我們沒有要求同步的話，程式是會繼續執行下去的，但當 Host 端程式執行到需要用到 Device 端的計算結果時，為了確保 Device 端真的已經執行完且也將資料載出完成，我們就必須在這加

上 Host Device 間的同步函數，以確保我們使用資料的正確性。

而後者最常被用到的情況是用來確保 SMEM 的資料一致性，雖然我們的 Threads 是平行的被執行(下個章節會介紹如何執行)，但還是會有先後的問題，因此當 Thread 對 SMEM 有存取的時候，為了確保資料的一致性，我們通常會在存取他之前或之後加上同步函數，以確保所有資料的一致性。

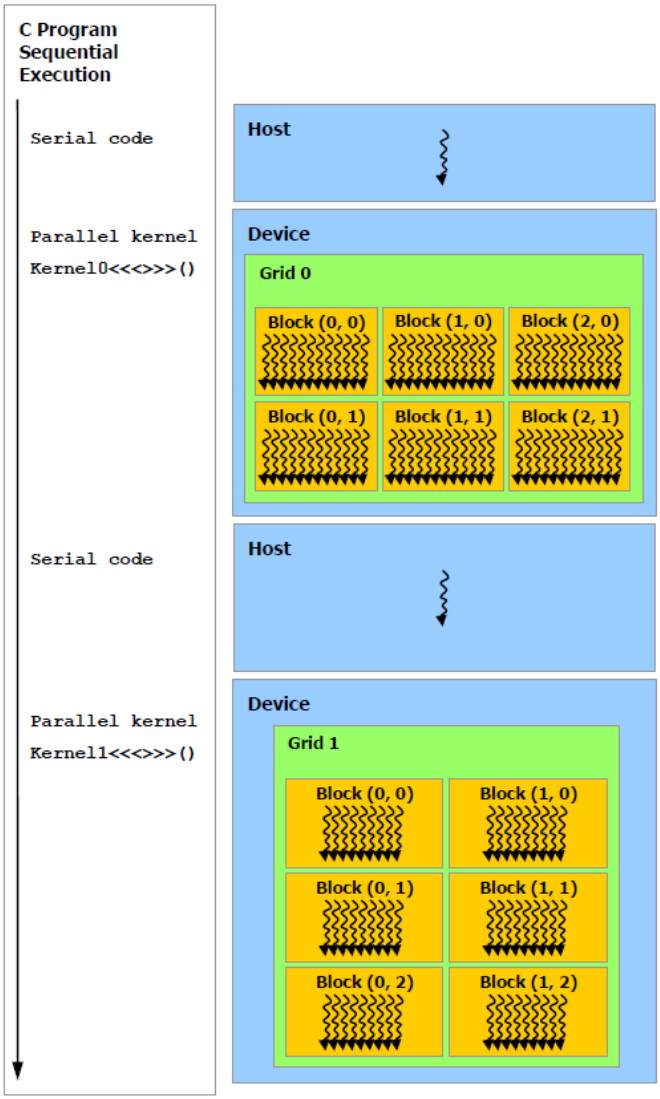


圖 4-9 CUDA Host 端程式執行

4.4 程式執行模型

在這章的前半部有提到，同一個Block內的Threads都會被分配到同一個SM來執行，而一個SM 同時間可能有多個Blocks在被執行，而一個SM 上能同時執行的Block數量，則主要取決於SM 的硬體資源，例如暫存器及SMEM，都會等分給各個Blocks，但扣除硬體資源不足的原因，還是有一個最高上限:八個Blocks。

當一個 SM 開始執行一個或多個 Block 時，SM 會依每個 Block 內的 Thread 編號，以三十二個 Threads 為一組分成一個 Warp，而 SM 則以 Warp 為排程及執行單位來執行我們的平行程式，且不同 Block 的 Warp 會一起在 SM 中排程及執行，而其中的排程則是由硬體來執行的。在我們宣告一個 Block 內的 Thread 個數時，盡可能讓它為 32 的倍數，這樣可以些許的降低資源的浪費，舉例來說：假設我們宣告一個 Block 內有六十五個 Threads，那麼系統會自動將這些 Threads 依序切成三個 Warps，其中第 0 到 31 號的 Threads 會被分到第一個 Warp，而第 32 到 63 號的 Threads 則會被分到第二個 Warp，最後剩下的那一個 Thread 系統會自動再加上 31 個沒有用的 Threads 使它也成為一個 Warp，因此我們等於浪費了 31 個 Threads 的空間。而 Warps 間的排程規則，則是由排程硬體隨機安排，或當某些 Warps 在做記憶體存取時，因為存取時間較久，且會造成運算資源(如 SP 或 SFU)的閒置，此時 Warp 排程器就會切換到下一個 Warp ，使它取得運算資源的使用權，以減少資源的浪費。

而在撰寫 kernel 時，為了讓平行程式執行得更有效率，我們必須避免同一個 Warp 內的 Threads 發生條件發散的情形，例如當存在 if、else 時，盡可能讓 Warp 內的 Threads 條件相同(即同為執行 if，或同為執行 else)，若發生條件發散時，Warp 內的 Threads 將會依條件的不同，被分為多個 Warps(一樣會補入沒有用的 Threads 到一個 Warp 32 個 Threads)，然後分別執行不同條件的程式後，再合併回同一個 Warp 繼續執行後面的程式。

另外，雖然我們的 Threads 是平行被執行的，但還是會有先後的的關係，而如上一段所說，在 Device 端我們的程式被切成好幾個 Warps 執行，而這些 Warps 內部又依單一指令多個執行緒(Single Instruction Multiple Threads)的方式被執行，也就是說同一個 Warp 內的 Threads 每次都一起執行相同的指令，

因此他們的執行進度會是一模一樣的。再來，這些 Warps 間也會有排程的問題。前面硬體簡介中有提到，因為一個 SM 擁有兩個 Warp 排程器，因此在一個 SM 內，可以同時有兩個 Warps 被執行，但哪一個 Warp 會被先執行卻是隨機的由硬體決定。從圖 4-10 中，我們可以看到 SM 內一個 Warp 排程的情況。

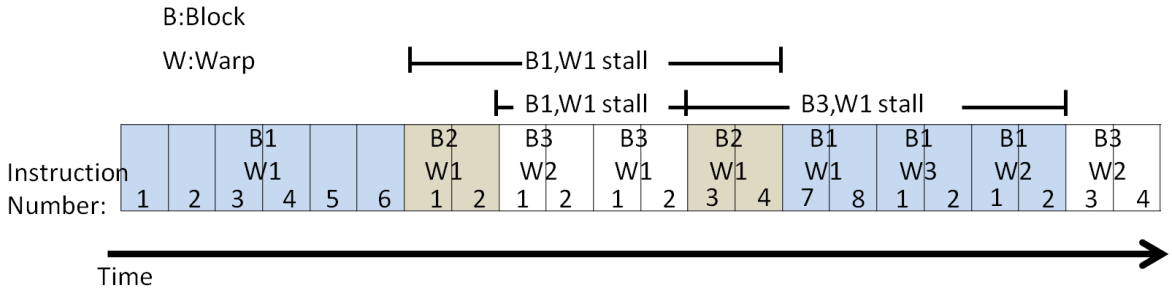


圖 4-10 SM 內的 Warp 排程

5 程式實作

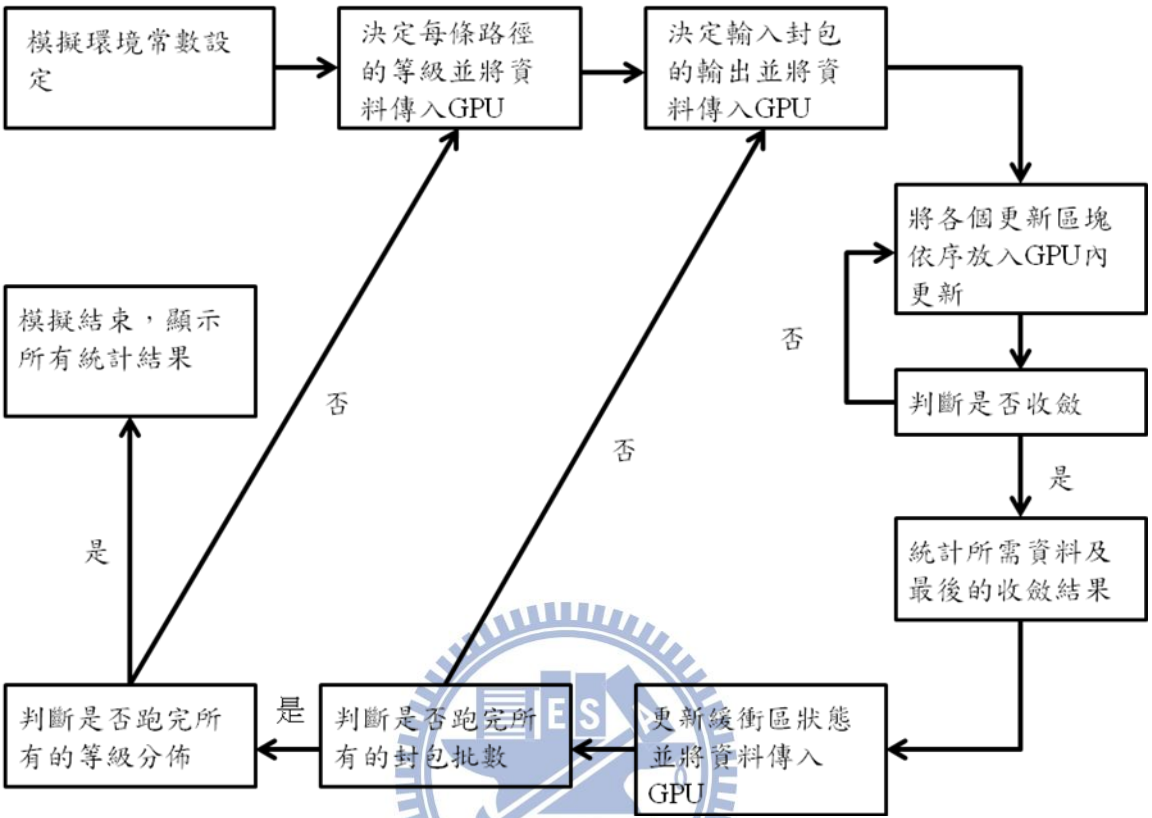


圖 5-1 模擬流程圖

在實作的最一開始，我們要先設定好我們整個模擬的環境常數(environment constant)，包含交換器的硬體架構，及 CUDA Thread 的安排及神經元的更新區塊個數。在交換器環境變數的部分，包含 N(輸入輸出的個數)，M(外部光纖的波長個數)，K(TSOPS 內第二階交換器個數)，L(交換器內部波長個數)，D(緩衝區光纖延遲線個數)，T(封包來的批數)，在這些變數設定完成後，也代表我們的交換器環境就確定了。在有了確定的交換器環境後，接下來，我們就可以來設定我們的 CUDA Thread 安排，包含 G(CUDA Block 的個數)、B(每個 Block 內的 Thread 個數)、S(每個 Block 所擁有的 SMEM 大小)，最後再決定我們要將我們的模型分成幾個更新區塊(I)來做更新。

在設定好我們所有模擬所需的常數後，我們的下一步就是要決定我們交換器內每一條路徑的等級，在前面介紹 SRNN 模型的章節中有提到，我們每條路徑的等級函數為：$r(P) = -[pr(P) \times Num_{H} + (1 - pr(P)) \times Num_{V}]$，但因為在我們

接下來所探討的問題裡，完全和封包的優先權無關，因此，為了簡化我們的問題，我們將每條路徑的等級簡化為：$r(P) = -[d(P) \times Num + \delta]$，其中 Num 為交換器總路徑個數，δ 為一整數且 $0 \le \delta < Num$，換句話說，也就是每個封包的優先權都相同。接下來我們再利用亂數來決定每一個封包的輸出端，值得一提的是，在這裡我們一樣為了簡化我們的問題，將我們模擬的每批封包附載(Load)都設定為 1，也就是說，每一批封包抵達輸入端，一定在每一個輸入端的每一個外部波常都會有一個封包進入。最後我們再將決定好路徑等級和每個封包要求的輸出端載入到 GPU 內，以備等一下我們利用 SRNN 來找到我們的最佳路徑安排。

在上面所需要的資料都載入到 GPU 內後，我們就可以依我們之前設定的 CUDA Thread 安排方式和神經元的更新分割方式，依序將我們的 Thread 載入到 GPU 內執行，值得一提的是，在我們將 SRNN 套用到我們的 WOPIS 問題上時，我們是將交換器內的每一條路徑對應到一個神經元，那在 CUDA 上實作時，我們又將 SRNN 的每一個神經元對應到 CUDA 上的一個 Thread。另外還有一點，在實作時，我們是等所有的神經元都更新完一次後，才來判斷 SRNN 是否已達收斂狀態。因為判斷是否收斂的程序必須在 CPU 端完成，因此我們必須先把每個神經元的狀態載出到 CPU 端後，才能執行判斷是否收斂的程序。所以如果我們在每一個區塊更新完，就判斷是否達收斂狀態，會因此浪費很多時間在載出資料上(後面會提到，這一段的資料傳輸是頻寬最低的)，舉例來說:如果們的 SRNN 要 30 次重複才會收斂，且我們將我們的模型分割成四個區塊來更新，那麼總共就需要載出：30*4=120 次資料，但假如我們只在所有的神經元都更新完一次後才載出資料，那麼總共就只需要載出 30 次資料，也就是說載出資料的個數會隨著我們分割的更新區塊增加而增加。

除了上續兩點外，我們在 CUDA 上實作時還會遇到兩個問題，一個是隨著我們的 CUDA Thread 安排的方式，會影響我們程式的平行度上限。另一個是 CUDA 內部只提供我們同 Block 內 Thread 間的同步，及 Host 和 Device 端之間的同步函數，但很明顯的，我們的 CUDA 程式在執行時，會需要用到跨 Block 間的 Thread 同步，而如何解決這兩個問題我們會在下一節中探討。

等到 SRNN 最後收斂了，我們就可以統計我們所需的資料(收斂的重複次數及所需時間)，還有收斂的結果(最後所挑出來的路徑)，然後再把緩衝區的狀態更新並傳回 GPU 內，以備下一批封包選擇路徑用。等封包批數跑到我們

所要的次數後，我們就可以再載入下一個不同的等級分佈，重複我們上緒的流程，直到所有的等級分佈都跑完。

5.1 實作問題及解決辦法

如前面提到的，在實作上我們還有兩個問題需要解決，一個是 CUDA Thread 的分配所導致的平行度上限問題，另一個則是 CUDA 無法提供我們跨 Block 間的 Thread 同步。在接下來的文章裡我們將依序透過範例，講解這兩個問題來原及解決辦法。且在在正式進入討論之前，我們先列出 SM 內所有的硬體資源上限，即對於 Block 及 Thread 執行個數的限制，以備接下來的內容使用。

每個 SM 同一時間的最大執行 Block 數	8
每個 SM 同一時間的最大執行 Thread 數	1536
每個 SM 所能提供的 32-bit 暫存器個數	32K
每個 SM 所能提供的 SMEM 大小	48KB

5.1.1 平行度上限

在 CUDA 裡每個 GPU 都有它的最大平行執行上限，如我們的 GTX580，它的最大平行執行是 16(SM 個數)x1536 = 24576 個 Threads，但往往我們會因為其它硬體需求的不足(如暫存器，SMEM)或是 CUDA Thread 的分配方式不好，而導致我們無法達到每個 GPU 的最大平行度上限。在下一段，我們將以我們的 GTX580 為範例，依序舉出四種情況，因分別違反上述表格的四個規則，而無法達到最大平行度:24576 個 Thread，並提出相對應的解決辦法。

- 範例一：

  假設我們將一個 Block 內的 Thread 個數設為 32，這樣總共會產生 768(24576/32 = 768)個 Blocks，但因為一個 SM，同一時間的最大執行 Block 數為八個，所以我們一個 SM 一次就只能同時執行 8*32 = 256 個 Threads，而整體 GPU 的平行度就只能為 256*16 = 4096 個 Threads。

- 範例二：

  假設我們將一個 Block 內的 Thread 個數設為 1024，這樣總共會產生 24 個 Blocks，但因為一個 SM 同一時間的最大執行 Thread 數為 1536，且同

一個 Block 內的 Threads 都會被分到相同的 SM 內執行，所以我們一個 SM 一次就只能執行一個 Block(1024 個 Threads)，而整體 GPU 的平行度就只能為 1024*16 = 16384 個 Threads。

在上面兩個範例中，都是因為 Thread 的分配方式不好，而導致無法達到硬體的最高平行度。但其實解決的辦法都很簡單，面對第一個範例，我們只需要簡單的將 Block 內的 Thread 個數提高到 192 個即可，這樣總共就只會產生 128 個 Block，剛好可以讓我們的十六個 SM，都達到 Block 數量的最高附載：八個。而範例二則是提醒了我們，一個 Block 內的 Thread 個數也不可以太多，不然反而會造成，因為超過一個 SM 所能同時執行的 Thread 個數上限，而塞不下一個完整的 Block，因而損失平行度。因此在範例二，我們只要將 Block 內的 Thread 個數降低到 512 個即可，這樣不只產生的 Block 個數(四十八個)，可以同時讓十六個 SM 附載(一個 SM 附載三個 Blocks)，且 SM 附載的 Thread 個數，剛好也達上限 1536 個。接下來我們要將硬體資源一起納進來考慮，看看對我們的平行度還會造成怎樣的影響。

- 範例三：

  假設我們每個 Thread 要用到 32 個暫存器，那麼因為一個 SM 所能提供的 32-bit 暫存器個數為 32x1024 個，因此我們一個 SM 一次就只能執行 1024 個 Threads，而整體 GPU 的平行度就只能為 1024x16 = 16384 個 Threads。

- 範例四：

  假設我們將一個 Block 內的 Thread 個數設為 256，且每個 Block 需要用到 16KB 的 SMEM，那麼因為一個 SM 所能提供的 SMEM 大小為 48KB，因此我們一個 SM 一次就只能執行三個 Block(768 個 Threads)，而整體 GPU 的平行度就只能為 768*16 = 12288 個 Threads。

面對範例三的問題，這是目前一個很大的矛盾點，因為每個 Thread 所使用的暫存器個數，是由 CUDA 的編譯器決定，雖然 CUDA 有提供，可以強制將每個 Thread 所使用的暫存器個數，壓到我們想要範圍內的函數，但程式的執行效率也會因此被大大降低了，因為它只是將原本我們存在暫存器的資料，改存在 GMEM 內，但這樣會大大提高了我們記憶體的存取所需的時間，因此目前還沒有一個很好的方式，能夠處理這一個問題，只能靠設計者在撰

寫程式時，盡量避免人為上的暫存器浪費。而在我們自己的 WOPIS 排程的程式內，這個問題也是主宰了我們平行執行上限的原因，因為這個問題，我們的平性執行上限被壓縮到只有 16384 個 Threads。至於範例四，目前有兩種不同的解決方法，第一種較簡單的方式，是將我們 Block 內的 Thread 個數提高到 512，這樣就算一個 SM 一次還是只能執行三個 Block，也可以達到平行度的最高上限。但是這是當我們提高 Block 內的 Thread 個數時，所需要的 SMEM 不會跟著提高的情況，假如當我們將 Thread 個數提高到 512 後，每個 Block 需要的 SMEM 跟著提高，變成 32KB 時，上述的方法就無法使用了。接下來我們將就這個問題，就我們 SRNN 模型運作的特性，提出另一個解決辦法。

在我們的 SRNN 模型上，其實它的運作方式很簡單，就是每一個神經元去讀其它神經元對自己的影響權重，最後把它加總起來判斷自己最後是否可以激發，在這樣的運作模式裡，我們為了加速我們的程式執行速度，會先花一點時間將所有神經元的狀態，從 GMEM 先搬進 SMEM 內，以便提升接下來的存取速度，而最直覺的方式，就是每個 Block，都需要一塊和神經元總數相同的 SMEM(幾個神經元就需要幾個 bytes 的大小)。但當發生 SMEM 不夠時，我們就必須利用有限的 SMEM 空間，來存取所有的神經元狀態。這時，我們可以就 SRNN 模型的運作特性，一次搬一部分的神經元狀態進入 SMEM 內即可，等到算完這些神經元對 Block 內神經元的影響權重後，在搬入下一部分的神經元狀態。這樣我們就可以利用有限的 SMEM 空間，重複使用，已達到不因 SMEM 的空間需求，而限制了我們的平行執行的最大上限。

5.1.2 同步

在我們的 SRNN 模型裡，因為我們的神經元更新函式是一個遞迴式的函式。因此，此刻神經元的狀態，會和上一刻其它所有神經元的狀態有關，因此我們永遠要等到所有的神經元都更新好某次的狀態後，才能再進入更新下一次狀態。也因為如此，不管是全平行更新或是分塊更新，都需要用到多個跨 Threads 間的同步，而這多個 Threads 的數量，往往都會超過我們一個 Block 所能容納的 Threads 個數上限。因此，我們無法使用 CUDA 提供的 Block 內 Threads 的同步函數，但我們可以利用 CUDA 另一個內建同步函數：Host 及

Device 間的同步函數，來達到我們的同步需求。我們可以在 Host 端用一個迴圈，將我們的 CUDA kernel 包住，每載入一個更新區塊的神經元進入 Device 端更新，就呼叫這個同步函數，直到所有前一個區塊的神經元都更新完畢，我們再載入另一個更新區塊的神經元，直到所有區塊都更新完畢，再跳出載入更新區塊的迴圈，判斷是否收斂，如果沒有收斂，就再回到載入更新區塊的迴圈，相對應的程式碼示意圖如圖 5-2。

```
do
{
    foreach update blocks
        kernel<<Grid,Block>>;
        cudaThreadSynchronize();
    end of foreach
}while(not converged)
```

圖 5-2 更新區塊同步程式碼

這樣的運作方式雖然直接解決了我們跨 Block 間的 Threads 同步問題，但因為利用的是 Host 及 Device 間的同步函數，所以相對應需要付出的負擔(overhead)也較大。但這是目前 CUDA 能夠提供跨 Block 間的 Threads 同步問題的唯一解。

5.2 程式優化

關於 CUDA 程式的優化，這裡提供三個基本的策略及方向，分別為：

- 最大化平行執行
- 最大化記憶體頻寬
- 最大化指令流量

以上三點我們會在下面的文章中一一描述。

5.2.1 最大化平行執行

從前面的章節我們可以知道，CUDA 程式的編輯原則，就是將一個大的程式，分割為許多小程式，並且將這些小程式一一分配給各個獨立執行的 Threads。除此之外，如何去分配一個 Block 內的 Threads 個數，及一個 Grid 內的 Blocks 個數也是非常重要的，因為這不只會影響我們程式的執行效率，也和我們的硬體資源分配有密不可分的關係。

資源分配的影響在上一節裡已經有提過了，在這裡我們只針對執行效率的部分。因為在同一個 Block 內的 Threads 都會被分配到同一個 SM 內執行，因此我們通常都會把 Block 的數量定為 SM 個數的整數倍，這樣才不會造成不同 SM 有不同負荷(執行不同個數的 SM)的情況，且因為我們會用到一些同 Block 內的 Threads 間的同步，為了能夠隱藏同步所產生的延遲，我們至少要分配每個 SM 兩個以上的 Blocks，這樣當一個 Block 因為同步需求進入延遲等待時，SM 還可以有另一個(以上)Block 可以執行，且一個 SM 內的 Block 數量越多，延遲的隱藏效果就越好。

雖然同一個 Block 內的 Threads 都會被分配到同一個 SM 內執行，但每個 Thread 在 SM 內執行時，又會被分割為 32 個 Threads 一組的 Warp，且一個 SM 同時可以執行兩個 Warps，因此一個 Block 內的 Thread 數量最好也能夠是 32 的整數倍，且 Block 內的 Thread 數量越多，也越可以幫助我們隱藏，因為記憶體存取所產生的延遲。當我們某一個 Warp 內的 Threads 執行到記憶體存取的部分時，往往等待記憶體存取完畢的時間，都可以拿來讓我們執行數個基本的運算了，因此這個時候 SM 內的 Warp 排程器，就會將等待中的 Warp 匯出，且匯入一個需要使用運算元件的 Warp，這樣可以讓我們 SM 內的運算元件永遠保持忙碌(Busy)，因此為了讓我們的 SM 內擁有足夠的 Warps，我們會盡量安排多一點的 Thread 數量在一個 Block 內。

5.2.2 最大化記憶體頻寬

我們知道，CUDA 非常適合用來執行大型資料平行的程式，但也因為如此，資料的傳輸速度變成一個，影響程式執行效率非常重要的要素。在 CUDA 程式中有兩個地方的資料傳輸路徑，是我們經常用到，但資料的傳輸頻寬卻是非常低的：

1. Host 和 Device 間的資料傳輸
2. GPU 外部的 GMEM 存取

而第一個的頻寬又比第二個來得低很多，且通常在不改變演算法的前提下，我們往往很難去降低對於第一個資料傳輸路徑的使用。至於對第二個資料傳輸路徑的使用，因為在 CUDA 上，對於晶片內部(on-chip)的記憶體存取，至少都比直接對晶片外部(off-chip) GMEM 存取快 100 倍以上，因此我們可以藉由先將 GMEM 的資料搬到 SMEM 上，等到使用完畢後，如果有需要，再將它們寫回 GMEM。而面對唯讀的資料，我們則可以利用 CUDA 提供給我們的常數記憶體(Constant Memory，CMEM)，及加速二維資料讀取的質地記憶體(Texture Memory，TMEM)，來降低對 GMEM 的存取，提高我們資料傳輸的頻寬。

舉例來說，假設我們對於 GMEM 的存取時間就剛好是晶片內部記憶體的 100 倍，那麼，如果我們對於一個晶片內部記憶體的存取需要花一個時脈週期(clock cycle)，且我們的程式需要對記憶體存取十次，那在都不使用晶片內部記憶體所花費的時脈週期就會是 10*100 = 1000 個時脈週期，假若我們先將資料載入到晶片內部記憶體內，再直接對它做存取，那麼所需要的時間會是 1*100+10*1 =110 個時脈週期。

5.2.3 最大化指令流量

對於指令流量的最大化，我們主要可以從兩個部分下手，一個是降低特殊函數的使用，另一個則是避免同一 Warp 內部 Threads 分歧(divergence)的情況發生。

在前介紹硬體的章節中有提到，每個 SM 內部都只有四個特殊函數原件(Special Function Unit, SFU)，雖然特殊函數原件對於計算一些特殊的函數，會比一般計算原更有效率，但還是比一般運算要多花上不少的時脈週期，且因為一個 SM 只有四個特殊函數原件，因此一個 Warp 要被分成八次才能使內部的 Threads 都計算完畢，由此我們就可以看出計算特殊函數是非常耗費時間的。

而在前面執行模型的章節中我們也有提到，CUDA 的執行是利用將 32 個 Threads 分割為一個 Warp，並且讓它們同時執行相同的指令，但是如果今天裡面有其中一個(或以上)的 Threads 因為發生條件分歧，必須執行不同的指令時，CUDA 為了堅守這個執行原則，會將執行相同指令的 Threads 又分割為一個 Warp，並且再塞入一些沒有用的 Threads，使得一個 Warp 還是維持 32 個 Threads，等到他們各自執行完不同的指令後，再將他們合併回一個 Warp，因此，當 Warp 內部有 Threads 發生分歧的情況時，是需要花非常多時脈週期去處理的。

以我們的 SRNN 模型為例，當它在 CUDA 上執行時，為了避免 Warp 內的 Threads 產生分歧情況，我們原本會用一個 if 的判斷式，來處理當神經元讀到自己的狀態時，就不要做權重的運算。但到了 CUDA 上，因為 Warp 內的 Threads 的讀取資料是同步的，因此為了避免有某神經元因讀到了自己的狀態而產生分歧，我們會將這個 if 的判斷式拿掉，改在後面多乘上一個邏輯判斷式，只要讀到自己的狀態就會乘以零，等於沒做這個計算。相反的，我們卻會保留判斷讀到的神經元狀態是零還是一(激發還是壓抑)的判斷式，因為不管是零還是一，Warp 內的 Threads 作的動作都會是一樣的，且當讀到零時，可以直接跳過，讀取下一個神經元狀態，省去一個權重的計算量，相對應的程式邏輯請參考圖 5-3。

```
foreach neural state
    if(neural state)
        net += weight*(neural state != my state)
    end of if
end of foreach
```

圖 5-3 SRNN 避免 Warp 內產生分程式碼

6 模擬結果與討論

在這個章節中，首先我們會先介紹一下我們模擬的環境，然後比較一下用 CUDA 平行化前後，程式執行的效率，及利用 CUDA 提供的最佳化策略，最佳化後的效能提升。再來就我們之前討論，影響收斂所需的更新週期因素，依序套用到我們 WOPIS 交換器排程問題上，看看更新週期是否如我們之前所推測的下降了。最後，在我們以 CUDA 為我們平行化工具的基礎下，就分塊更新會降低更新週期的特性，看看是否能在更新週期和平行度上(分塊更新會降低程式平行度)，找到一個最佳的平衡點，使我們的模型達收斂所需的時間最短。

6.1 模擬環境

程式執行環境:

CPU	Intel i7 2600k
RAM	8GB
OS	Windows 7 64bits
GPU	NVIDIA GTX580
CUDA Version	Version 4.0

在接下來的模擬中，我們將全部以我們的 WOPIS 交換器排程問題，做為我們探討 SRNN 模型各項因素的基礎。其中為了簡化我們的問題，我們將把各個封包的優先權拿掉，使得我們的等級函數變為：$r(P) = -[d(P) \times Num + \delta]$，其中 Num為模型神經元的總數(交換器的總路徑數)。還有我們將把所有可能的路徑以一個六維矩陣表示：[N][M][K][M][N][L]，而六個維度，則分別代交換器內各個不同的硬體路徑。因此每一條路徑，都有它自己在模型中唯一的編號。而在分塊更新時，我們則直接依這個編號，依序分割我們的模型成數個更新區塊。且為了能夠準確測量計算量與時間的關係，我們會將因為讀到神經元狀態是壓抑態，就不做權重運算的判斷式拿掉，已備我們測量出來的時間不會因為實際計算量的不同，而產生差別。

6.2 序列更新、平行更新、優化平行更新的比較

在這個章節裡，我們將先給大家看看，經由 CUDA GPU 平行化執行的程式，和單純使用 CPU，以序列化方式執行，在一個更新週期所需時間的差別。再來比較在 CUDA 上，有無做最佳化，對整體執行效能的影響。

首先，我們先提供所有的樣本點數據，如表 6-1。從表 6-1 我們可以很清楚的看見，經過 GPU 平行執行的程式，和單純使用 CPU 執行的程式，在一個更新週期所需的時間上，隨著模型的神經元總數的增加，時間的差距會越來越大，在最小的模型大小(六十四個神經元)時，兩者的差距大約為三到四倍，隨著我們的神經元越加越多，到了最大模型大小(16384 個神經元)時，差距已經來到了五百倍以上，而這也和我們所期望看到的結果差不多。以我們的 GTX580 來說，它擁有了 512 個 SP，所以就硬體上的直接對應(不考慮處理器的時脈等細部因素)，我們本來就期待它可以帶來大約五百倍的加速。只是當模型還不夠大時，本來平行的好處就很難完全展現，且我們的 SRNN 模型還有同步的問題，所以在一開始才會只有三四倍的差距，但隨著模型的增大，平行的好處也展現得越來越好。

Model size	CPU sequential update(ms)	GPU no optimization parallel update(ms)	GPU with optimization parallel update(ms)
64	0.5378	0.1859	0.1305
128	2.4643	0.2449	0.1763
256	6.8831	0.3434	0.2735
512	26.9690	0.5712	0.4701
1024	98.5854	0.9634	0.8061
2048	385.1981	1.8803	1.6194
4096	1259.1106	3.3798	2.8308
8192	5040.4679	10.2273	8.7415
16384	18633.6991	36.1996	30.9136

表 6-1 序列更新、平行更新、優化平行更新，一個更新週期所需時間數據

我們知道，隨著我們模型內的神經元個數增加，模型的整體計算量是以平方增長得。而因為 CPU 是序列化的執行程式，所以所需要的時間基本上和計算量是成正比的，因此，這可以用來解釋為何圖 6-1 CPU 是一條接近平方的曲線。但相反的，按照道理來說，假如程式是依平行的方式執行，這條曲線應該會變成一條線性的直線。但從圖 6-2，我們可以發現 GPU 的圖形，大概在模型大小過了兩千個神經元後，就開始變成比較像指數的曲線關係。這是因為 CUDA 的平行機制，本來就不像一般在 CPU 上的如此單純，一個處理核心專門負責一個 Thread。再來，以我們的 GPU GTX580 來說，它也只擁有 512 個計算核心，因此當模型大小還沒超過兩千個神經元時，它還可以勉強保持平行的特性，使所需時間和計算量成 $1/n$ 的關係。但隨著模型大小越來越大，它也就越難維持這個特性了。

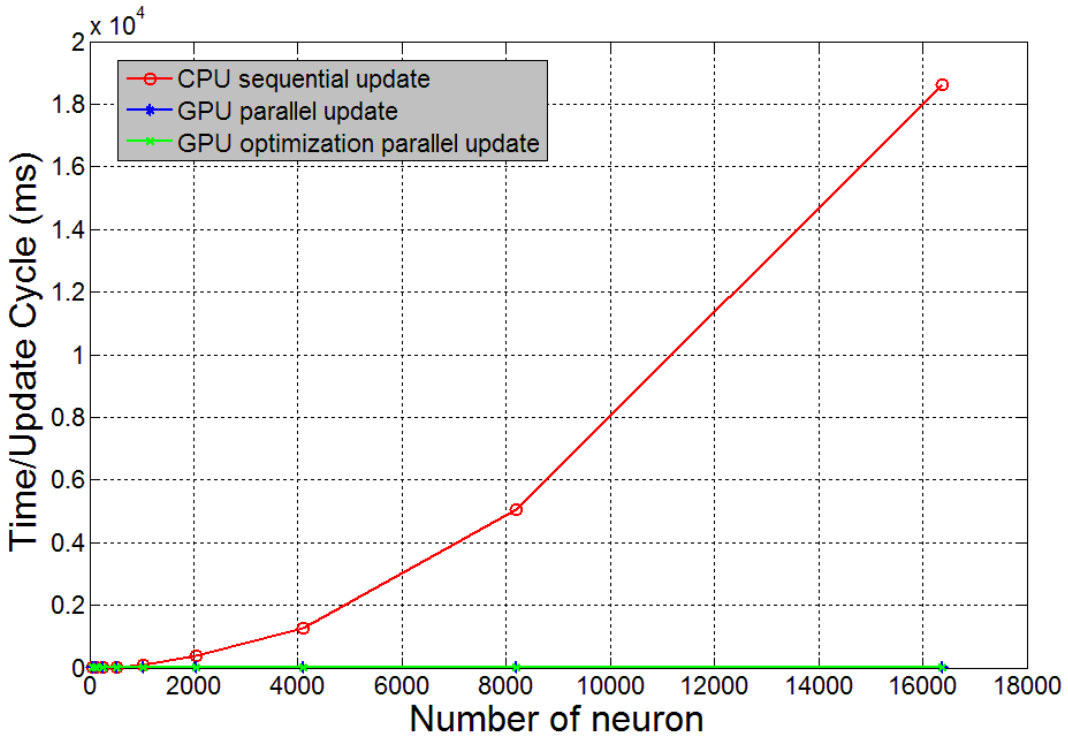


圖 6-1 CPU 序列更新及 GPU 平行更新比較

在比較完 CPU 和 GPU 後，接下來我們來看看，我們的 CUDA 程式優化到底對執行效率造成了多大的影響。不管是從表 6-1 的數據或是，圖 6-2 的曲線，我們都可以看出，優化後的程式，在每一個取樣點，大概都讓我們省去了 15%以上的時間。這反應了，隨著 CUDA GPU 硬體效能的提升(我們使用的 Fermi 架構已經能夠在 GPU 內加入快取)，程式優化還是對程式的執行效率，有相當的影響力。且不要小看這區區的 15%，我們可以從圖 6-2 看見，隨著模型的神經元個數增加，這 15%所代表的時間也會跟著放大。

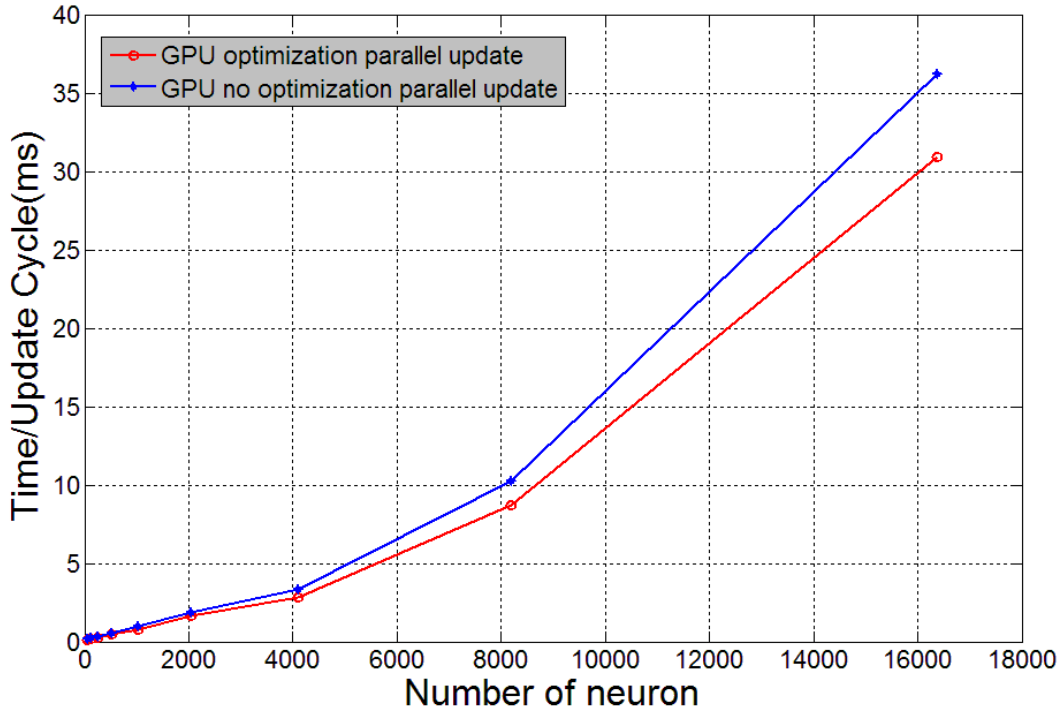


圖 6-2 GPU 平行更新及優化平行更新比較

6.3 全平行更新和分塊更新

在這個章節中，我們將探討神經元的更新方式，對我們 SRNN 運作在 WOPIS 交換器排程問題上的收斂所需更新週期的影響。在前面的探討中有提到，分塊更新有助於我們降低收斂所需的更新週期，且分得越多塊效果越好。因此，接下來我們將探討，在一個相同的模型內，更新的區塊個數不同，對所需的更新週期次數的影響。

在這個因素的討論中，為了維持問題的一般性，我們將把我們神經元等級函數的最後一項常數 δ，以隨機的方式安排，並且提供兩種不同的模型，做為兩種不同模型大小的比較。在下面表 6-2 中，我們提供了我們所有樣本點的數據。其中在表格最下面那一列的 sequential 項，是指當我們將每一個神經元都視為一個更新區塊時，收斂所需的更新週期個數。

	Model(N,M,K,L,F)	
Number of Blocks	(4,8,3,4,4)	(4,4,3,4,4)
1	16.01	11.94
2	13.25	10.01
3	12.39	9.28
6	11.51	8.58
12	11.01	8.11
24	10.53	7.47
sequential	7.97	6.33

表 6-2 分塊更新數據

從圖 6-3 我們可以發現，收斂所需的更新週期如我們之前所預料的，隨著我們分割的更新區塊增加而減少。且在分割的初期(更新區塊還沒分那麼多時)，更新週期掉的幅度明顯比後期多，且由表 6-2 可以看到，當我們將模型分割為二十四個更新區塊時，其實所需要的更新週期，已經和序列更新的相距不遠了。雖然我們不能說序列更新所需的更新週期一定會是最少，但至少機率會是最高的。且分塊更新在實作上，其實是在犧牲平行度及增加同步的負擔，因為我們一次更新的神經元個數變少了，但所需的同步次數卻是增加

的(每一個更新區塊都需要同步一次)。因此，假如單就模型達收斂所需的時間做考量，其實不是分越多個更新區塊越好，儘管它可以降低我們達收斂所需的計算量，但卻相對應的，犧牲了我們硬體所能達到的最大平行度，及額外付出了因所需的同步次數增加，的同步等待時間。

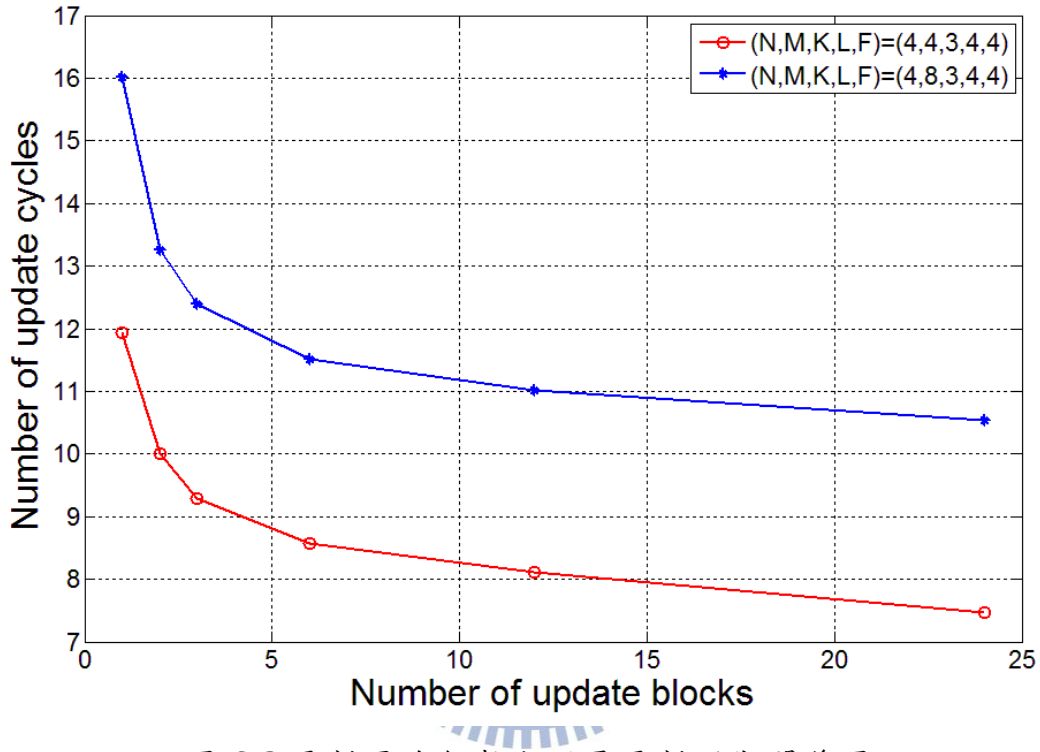


圖 6-3 更新區塊個數和所需更新週期關係圖

在模擬討論的最後，我們將試著找出我們的 GPU(GTX580)，在不同的平行度下，執行的效率，並希望依此找出每一種模型，就最少收斂所需時間為前提，找到一個最有效率的更新區塊個數。換句話說，也就是就分塊更新可以降低收斂所需計算量，但卻會降低硬體的執行效率，及付出額外的同步時間，兩者間的權衡(tradeoff)下，找出一個可以達最低收斂時間的更新區塊個數。

6.4 隨機常數神經元等級安排及序列常數神經元安排比較

在這個章節中，我們將探討神經元等級安排，對我們 SRNN 運作在 WOPIS 交換器排程問題上的收斂所需更新週期的影響。值得注意的是，在前面我們有提到，在我們的模擬中，我們將神經元的等級函數簡化為：$r(P) = -[d(P \times) N \omega + n$。而我們這裡所謂的序列或是隨機等級安排，指的是對最後一項常數 δ，我們是直接指定它為自己的路徑編號，或是隨機給予一個 0 到 Num 且不重複的常數。而從我們簡化後得到的等級函數可以看出，整個函數的大小，是被路徑延遲所主導(dominant)，而從第一章介紹 WOPIS 架構的內容可知，我們的路徑延遲，是由路徑的輸出(O)及內部波長(λ)的編號差所唯一決定的。因此，我們對於等級函數的最後一項常數 δ 的安排，只有當路徑的輸出及內部波長的編號差相等時，才會產生作用。

	Model(4,8,3,4,4)	
Number of Blocks	Sequential Constant	Random Constant
1	34.85	16.01
2	32.14	13.25
3	29.88	12.39
6	25.80	11.51
12	22.15	11.01
24	18.72	10.53
48	15.16	9.77
sequential	3.61	7.97

表 6-3 Model(4,8,3,4,4)數據

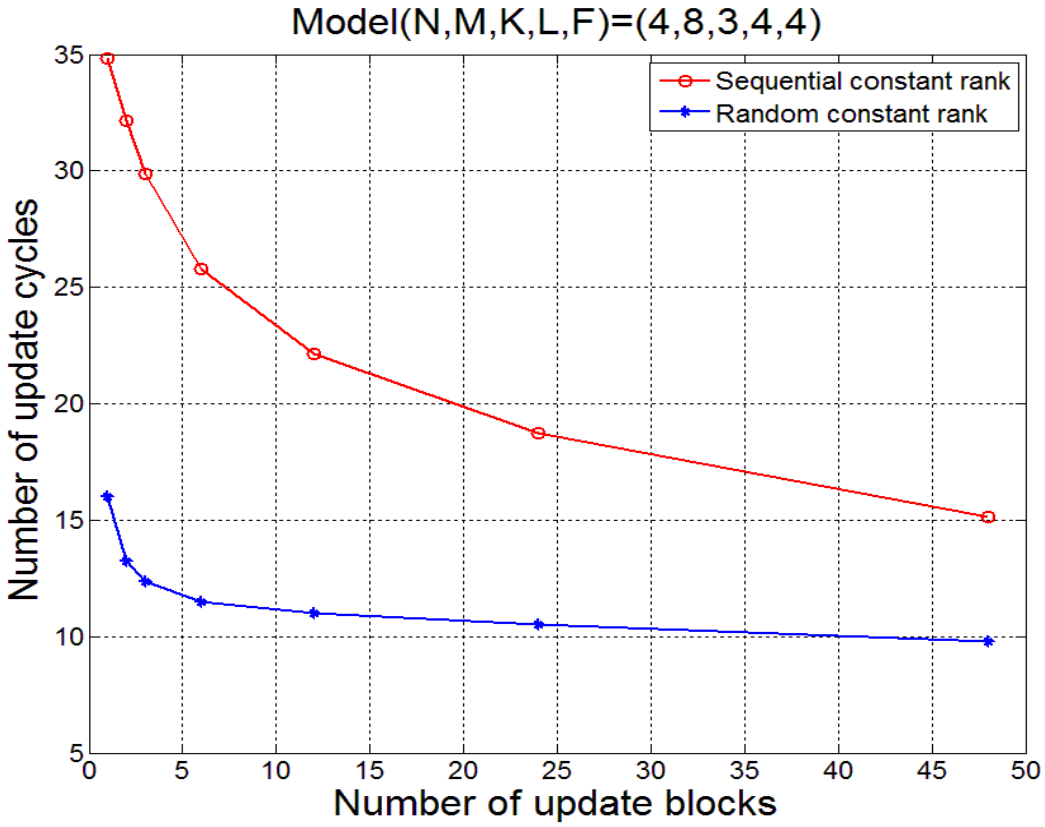


圖 6-4 Model(4,8,3,4,4) 隨機常數等級安排及序列常數等級安排

無論是從圖 6-4 的 Model(4,8,3,4,4)或是圖 6-5 的 Model(4,4,3,4,4)，我們都可以很清楚的發現，隨機常數等級安排所需的更新週期，都比序列常數安排來得少，這和我們前面討論神經元等級安排，對收斂所需更新週期影響的結果相互呼應。但值得注意的是，隨著我們將更新區塊切得越來越多，序列常數等級安排會比隨機常數等級安排，所需的收斂更新週期數量來得少，而這點我們也可以從表 6-3 及 6-4 最下面的序列更新得到驗證。而這也再次反映了，一個模型達收斂所需的更新週期數量，不是靠一個單一因素就可以決定的。如我們模擬出來的結果，我們無法說序列常數等級安排所需的更新週期，一定比隨機常數等級安排來得多，因為這和我們分的更新區塊個數也有關係。

另外還有一個值得注意的地方是，我們前面提到，隨著我們將更新區塊切得越來越多，序列常數等級安排會比隨機常數等級安排，所需的收斂更新週期數量來得少。這是因為我們的序列等級安排，如果在不考慮延遲對神經原等級影響的前提下，其實就是將我們的神經元依它的編號，給予由大到小

的神經元等級，也就是說，編號越前面的神經元，等級也越高。因此，雖然同一個更新區塊內還是會包含各種不同延遲的神經元，但隨著更新區塊越切越多，區塊內的神經元個數也越少，假如我們只看不同更新區塊間，延遲相同的神經元，我們的更新方式也會隨著更新區塊的增加，越來越像由大到小的序列更新。

	Model(4,4,3,4,4)	
Number of Blocks	Sequential Constant	Random Constant
1	18.44	11.94
2	16.06	10.01
3	14.66	9.28
6	12.22	8.58
12	9.82	8.11
24	8.49	7.47
sequential	3.46	6.33

表 6-4 Model(4,4,3,4,4)數據

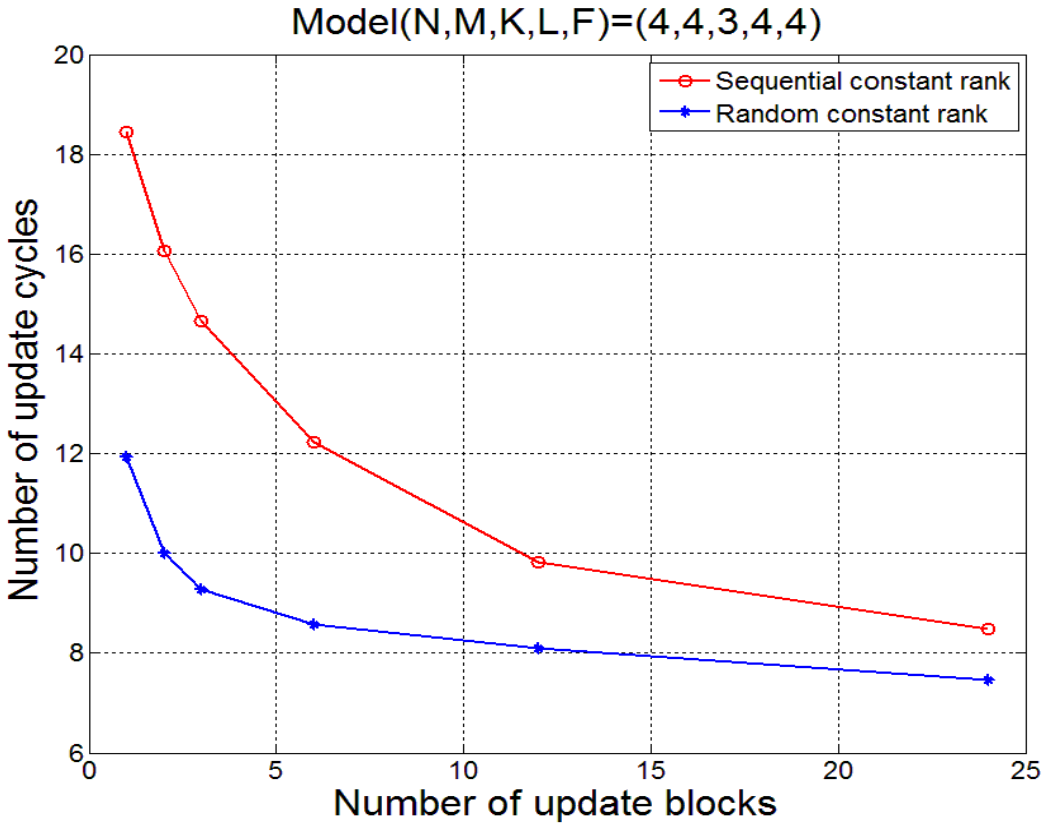


圖 6-5 Model(4,4,3,4,4) 隨機常數等級安排及序列常數等級安排

6.5 平行度及每個更新週期所需時間之關係

為了能夠找到最有效率的更新區塊個數，在這一個章節中我們將探討，在不同的平行度之下，我們一個更新週期所花的時間。但在正式進入討論前，我們要先說明一下，這裡的平行度所代表的含意。我們這裡定義的平行度為：一次同步執行的 Thread 個數/硬體能夠一次同步執行的最大 Thread 個數。前面章節有提到，分塊更新會犧牲我們的平行度，那是因為分塊更新的本意，就是將我們的模型分成好幾個區塊，一個一個做同步的更新。也就是說，假設我們的硬體最高能同時執行一千個神經元，如果我們將一個一百個神經元的模型，使用全平行更新(所有神經元同步更新)，我們的平行度就是 0.1，但假如將它分割成五個更新區塊，那麼平行度則變成只剩下 0.02 了，且在一個更新週期內，我們還要多負擔，四次同步所耗費的時間。而在接下來的模擬討論中，我將把同時執行 16384 個(受暫存器個數影響)神經元定為平行度 1。還有，為了驗證我們的硬體，在不同大小的模型中，執行效率是一樣的，我

們將利用對各種不同大小模型的，平行度及每個更新週期所需時間的關係圖(如表 6-5)，做常規化(normalize)，看看最後所有的關係圖會不會重疊在一起。更仔細一點來說，因為我們模型的計算量和模型大小的關係為$O(n^2)$，因此我們將對不同大小模型的更新週期執行時間，依n^2的關係，常規化為相同的模型大小。

Model(4,8,3,4,4)		Model(4,8,4,4,4)		Model(4,8,5,4,4)	
Parallelism	Time/UC	Parallelism	Time/UC (ms)	Parallelism	Time/UC (ms)
0.75	18.32	1	31.16	0.625	51.10
0.25	23.01	0.5	31.82	0.3125	57.61
0.125	42.21	0.25	40.02	0.25	65.55
0.0625	83.60	0.125	74.27	0.125	118.61
0.03125	169.62	0.0625	148.16	0.0625	235.25
		0.03125	298.16	0.03125	477.01

表 6-5 不同大小模型單一更新週期所需時間和執行平行度關係樣本

首先從圖 6-6 中我們可以看到，由上到下，我們依序提供了三種由大到小的模型，且因為 Model(4,8,5,4,4)的神經元總量，超過了我們硬體所能負荷的最高平行執行數量，因此只能從將模型切為兩個更新區塊(平行度 0.6251)，開始觀察，而第二大的模型，Model(4,8,4,4,4) 的神經元總量，則剛好為我們硬體能夠負荷的的最高平行執行數量。且不管是哪一個大小的模型，我們都可以發現在平行度大於 0.25 後，隨著平行度的增加，一個更新週期所需的執行時間幾乎是線性的向下遞減。反觀平行度小於 0.25 的，隨著平行度的減少，一個更新週期所需的時間，則是呈現指數向上遞增。

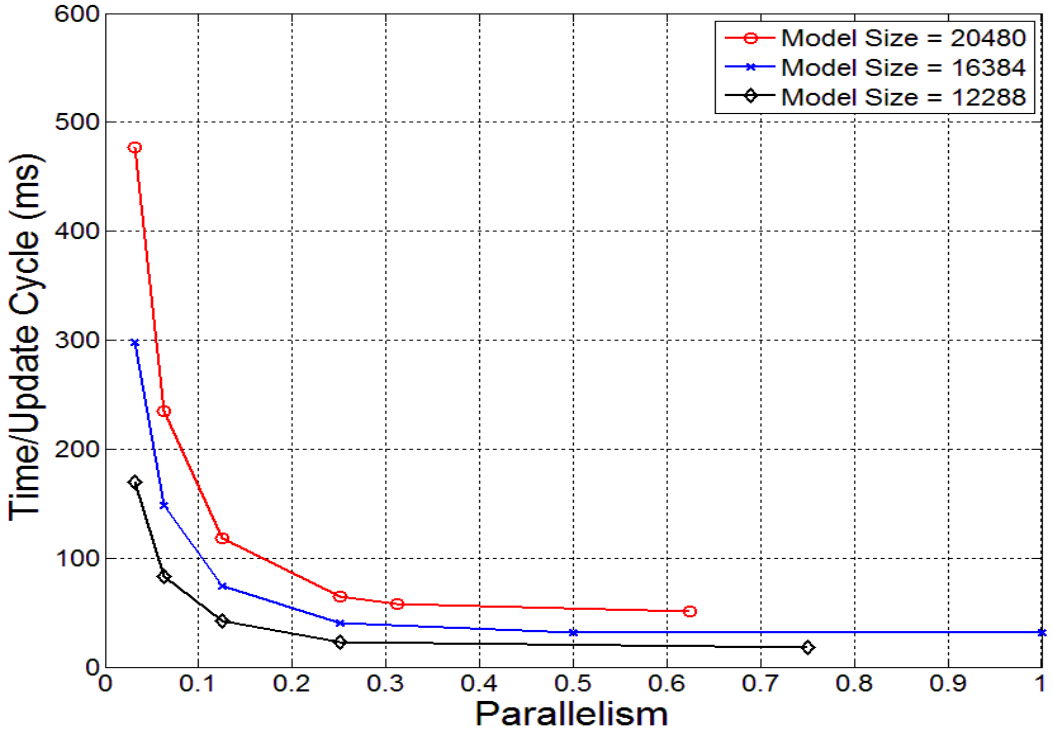


圖 6-6 不同大小模型單一更新週期所需時間和執行平行度關係

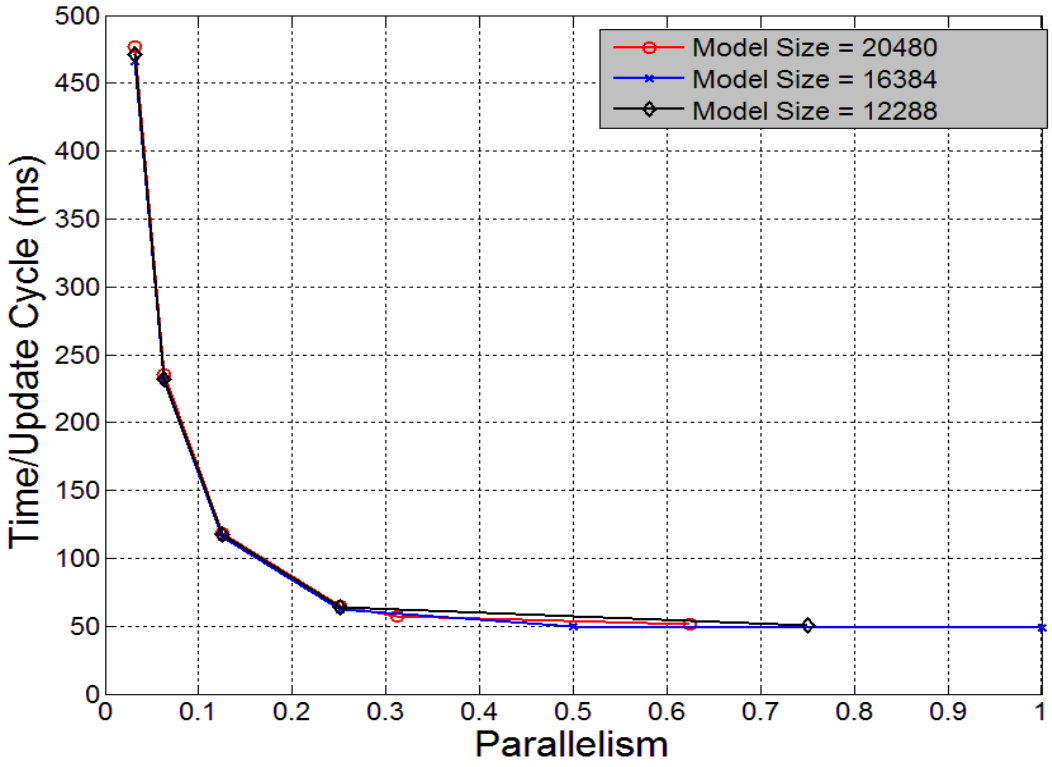


圖 6-7 各種大小模型常規化後單一更新週期所需時間和執行平行度關係

而接下來在圖 6-7 上，我們將所有模型的時間，都乘上模型大小比值的平方，常規化為最大模型 Model(4,8,5,4,4)。舉例來說，如第二大的模型 Model(4,8,4,4,4)，因為他和最大模型的大小比值為 1.25，所以我們就將它的每一個樣本點都乘上 1.25 的平方(1.5625)。而從圖可以輕易看出，經常規化後的三條曲線密切的交疊在一起，這反應了，我們的硬體在各個平行度下的執行效率，和所需的計算量是呈現密切相關的。另外，不管是從圖 6-7 或是 6-6 上都可以很清楚的看見，當平行度小於 0.25 後，一個更新週期所需的時間會快速增加，這樣的結果暗示著，在平行度及達收斂所需更新週期數量的權衡中，平行度 0.25 可能會是一個門檻，因為當平行度小於 0.25 後，因為分塊更新所降低的更新週期數量，可能很難與犧牲平行度所造成的執行效率下降，達成平衡。

接下來，為了能夠讓大家清楚看到，收斂所需更新週期和一個更新週期的執行時間，隨平行度的變化。我們分別將三個參數依序放在兩個 Y 軸及 x 軸上，如圖 6-8、6-9。並且利用前面討論過的兩種神經元等級安排，做為兩種不同神經元等級分佈的比較，如表 6-6 圖 6-8 為序列常數等級安排，而表 6-7 圖 6-9 則為隨機常數等級安排。另外值得一提的是，因為這裡的橫座標變成了平行度，而非前面討論收斂所需更新週期的更新區塊個數，所以代表更新區塊變化量的曲線和前面討論的圖長得會不太一樣，且因為我們的範例模型大小剛好和我們硬體所能支援的平行量相等，因此，圖中平行度 1 就剛好代表全平行更新，而平行度 0.5 則代表分成兩個更新區塊，以此類推。

從圖 6-8 和 6-9 的比較中我們可以發現，序列常數神經元等級安排，在平行度較大時，收斂所需更新週期下降的幅度，明顯比隨機常數神經元等級安排來得少，如平行度從 1 到 0.5 時，序列常數神經元等級安排的收斂所需更新週期大概只下降了 7%，但反觀隨機常數神經元等級安排，則下降了將近 17%。而無論哪個等級分佈，它們的一個更新週期執行時間隨平行度的變化曲線是相同的(因為是同樣的模型)，且當平行度從 1 降低到 0.5 時，所需的時間大概只上升 2%了。因此，可以預知的，無論是序列常數神經元等級安排，還是隨機常數神經元等級安排，將模型切成兩個更新區塊，都會比全平行更新花費較少的時間，而我們也從表 6-6 及 6-6 中得到了驗證。

Model(4,8,4,4,4) Sequential constant rank			
Parallelism	Time/Update Cycle (ms)	Number of update cycles	Total model converge time (ms)
1	31.16	35.03	1091.63
0.5	31.82	32.63	**1038.36**
0.25	40.02	29.31	1173.04
0.125	74.27	25.33	1881.31
0.0625	148.16	21.64	3206.27
0.03125	298.16	19.03	5673.96

表 6-6 Model(4,8,4,4,4)序列常數神經元等級安排數據

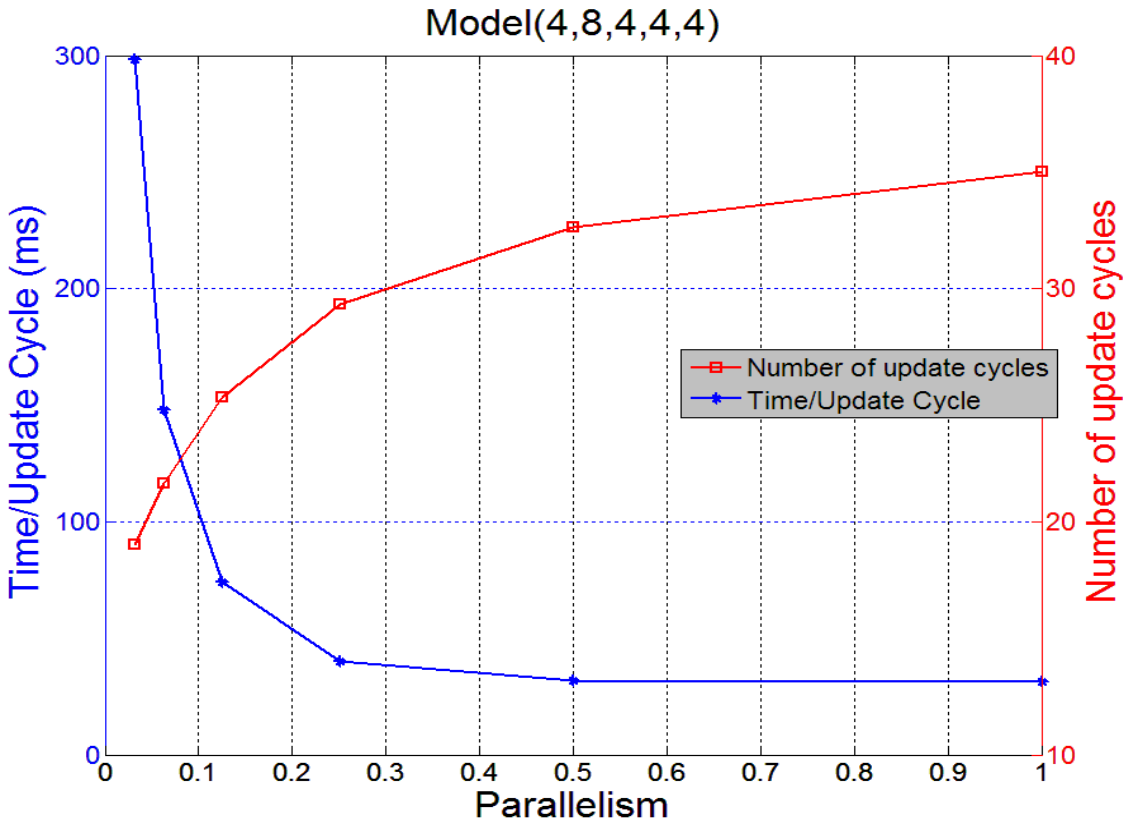


圖 6-8 Model(4,8,4,4,4)序列常數神經元等級安排綜合比較圖

Model(4,8,4,4,4) Random constant rank			
Parallelism	Time/Update Cycle (ms)	Number of update cycles	Total model converge time (ms)
1	31.16	16.21	516.60
0.5	31.82	13.50	**446.13**
0.25	40.02	12.23	508.26
0.125	74.27	11.75	937.44
0.0625	148.16	11.42	1748.4
0.03125	298.16	11.25	3492.76

表 6-7 Model(4,8,4,4,4)隨機常數神經元等級安排數據

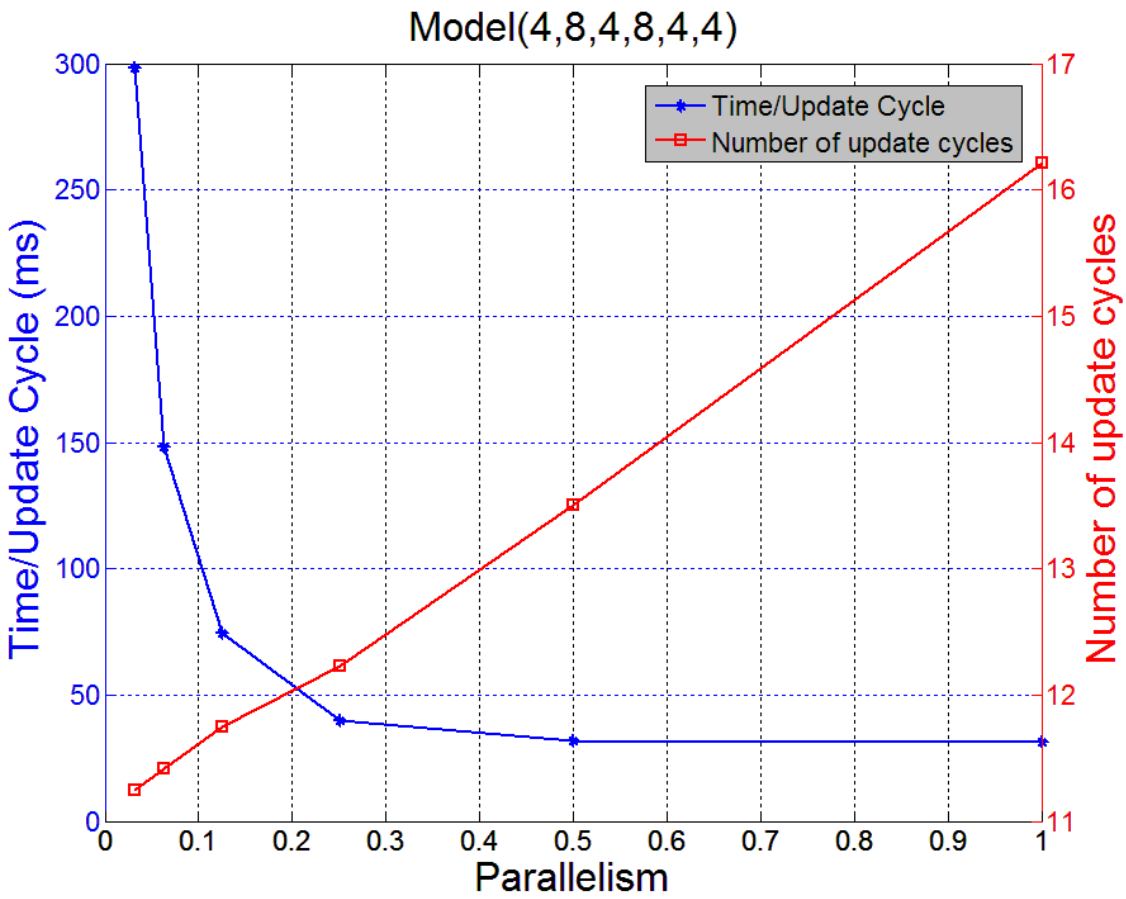
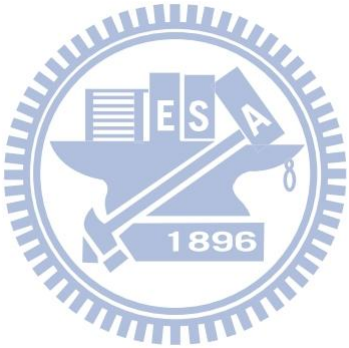


圖 6-9 Model(4,8,4,4,4)序列常數神經元等級安排綜合比較圖

且從表格中我們也可以清楚看見，當平行度小於 0.25 後，一個更新週期所需的執行時間就幾乎是以倍數的方式在成長，而這也驗證了，我們在前面的單一更新週期所需時間和執行平行度關係圖中的觀察。雖然我們目前還無法準確的預測，收斂所需更新週期和更新區塊個數之間的準確關係。但是，可以得知的是，我們無法犧牲太多的平行度，來換取收斂所需更新週期的下降，因為當平行度小於 0.25 後，一個更新區塊所需的執行時間會快速上升，而上升的幅度是遠大於更新區塊下降的幅度。但就如我們模擬出來的結果，在全平行和平行度 0.25 之間，我們還是有機會找到一個比全平行更新還好的更新區塊個數。



7 結論及後續工作

在本篇論文中，我們利用 SRNN 模型可高度平行更新的特性，來抵銷 SRNN 模型因遞迴式的更新函數，而不利於平行化的特性。且因為這兩個特性，我們捨棄使用須透過網路連結，才能提供高度平行執行的 CPU-based 平行化工具，改用可以在單一晶片上，就可以提供高平行執行的 CUDA GPU，藉此大幅降低 SRNN 模型達收斂所需的時間。並且透過三種程式最佳化策略，使我們的 CUDA 程式執行得更快更有效率。而在 SRNN 模型上，我們則討論各種影響模型達收斂狀態所需更新週期的因素，藉此以降低模型達收斂狀態所需的更新週期個數，減少整體模型的計算量。最後利用 WOPIS 交換器路徑排程問題，作為我們實作 SRNN 模型的處理問題，並且藉由觀察 CUDA 在不同平行度下的硬體執行的效率，得知當平行度大於 0.25 後，平行度和一個更新週期所需時間，會呈現比較接近線性的關係。最後再利用 SRNN 模型分塊更新，可降低達收斂狀態所需更新週期個數的特性，找出一個最佳的更新區塊個數，使我們可以花費最短的時間，讓模型就達收斂狀態。

在未來，我們可以針對套用 SRNN 模型處理的問題特性，就我們討論影響收斂所需更新週期個數的因素，如神經元等級安排、更新區塊的切割方式等…，做一些特別的安排，使我們的 SRNN 模型能夠在較短的更新週期內達收斂狀態，甚至可以達到準確預測中間的權衡關係。並且，透過在 CUDA 上實作平行更新的經驗，嘗試在其它平台上(如 Field-Programmable Gate Arrays，FPGA)或是更新版本的 CUDA 上實作，相信隨著平行計算的技術日新月異，總有一天我們能夠提供即時的(real time)SRNN 模型排程系統。

8 Reference

[1] C. Lin and C. S. Lee, “Neural Fuzzy Systems A Neuro-Fuzzy Synergism to Intelligent Systems,” *Prentice Hall* 1996.

[2] X. Hu and B. Zhang, “A New Recurrent Neural Network for Solving Convex Quadratic Programming Problems With an Application to the k-Winners-Take-All Problem,” *IEEE Trans. Neural Networks*, vol. 20,no. 4, April 2009, pp. 654-664.

[3] P. Tien and B. Ke, “A new ranked Hopfield neural networks approach to QoS parallel scheduling for WDM optical interconnection system, ” *IEEE High Performance Switching and Routing (HPSR), July 2011, pp.* 276 - 281

[4] X. Gao and L. Liao, “A novel neural network for a class of convex quadratic minimax problems,” Neural Computing, vol. 18, no. 8, Aug. 2006, pp. 1818–1846.

[5] X. Hu and J. Wang, “A recurrent neural network for solving a class of general variational inequalities,” IEEE Trans. Syst. Man Cybern. B, Cybern., vol. 37, no. 3, Jan. 2007, pp. 528–539.

[6] Q. Liu and J. Wang, “A one-layer recurrent neural network with a discontinuous hard-limiting activation function for quadratic program.,” IEEE Trans. Neural Networks, vol. 19, no. 4, Apr. 2008, pp. 558–570.

[7] Y. Li, Z. Tang, G. Xia, and R. Wang, “A Positively Self-Feedbacked Hopfield Neural Network Architecture for Crossbar Switching,” IEEE Trans. Circuits and Systems, vol. 52, no. 1, Jan. 2005, pp. 200-206.

[8] NVIDIA, “NVIDIA CUDA C Programming Guide,” Version 4.0, 20011.

[9] NVIDIA CUDA Technology in Adobe Premiere Pro and the New Mercury Playback Engine

[10] NVIDIA CUDA “Reference Manual,” Version 3.2 Beta,Aug.2010.

[11] A. Karanth Kodi, and A. Louri, “Multidimensional and Reconfigurable Optical Interconnects for High-Performance Computing (HPC) Systems,” J. Lightw. Technol., vol. 27, no. 21, Nov. 2009, pp. 4634-4641.

[12]S. Scott, “Optical Interconnects in Future HPC System,” IEEE/OSA Optical Fiber Communication Conference, 2011.

[13]A. Wonfor, H. Wang, R. V. Penty, and I. H. White, “Large Port Count High-Speed Optical Switch Fabric for Use Within Datacenters,”IEEE/OSA Journal of Optical Communications and Networking, vol. 3,no. 8, Aug. 2011, pp. A32-A39.

[14]T. Wang et al., “All-Optical Switching Data Center Network Supporting 100Gbps Upgrade and Mixed-Line-Rate Interoperability,” IEEE/OSA Optical Fiber Communication Conference, 2011.

[15]M. Yuang, Y. Lin, J. Shih, J. J. Chen, Po. Tien, S. S. W. Lee, and S.Lin, ”A QoS Optical Packet Switching System: Architectural Design and Experimental Demonstration,” IEEE Comm. Mag., May 2010, pp.66-75.

[16]I. M. Soganci et al., “Monolithically Integrated InP 1x16 Optical Switch With Wavelength-Insensitive Operation,” IEEE Photon. Technol. Lett.,vol. 22, no. 3, Feb. 2010, pp. 143-145.

[17]J. Gripp, et al., “Architectures, Components, and Subsystems for Future Optical Packet Switches,” IEEE J. Sel. opics Quantum Electron., vol.16, no. 5, pp. 1394–1404, Sept./Oct. 2010.

[18]B. Sarker, T. Yoshino, and S. Majumder, “All-Optical Wavelength Conversion Based on Cross-Phase Modulation (XPM) in a Single-Mode Fiber and a Mach-Zehnder Interferometer,” IEEE Photon. Technol. Lett.,vol. 14, no. 3, March 2002, pp. 340-342.

[19]S. Liew, G. Hu, and H. Chao, “Scheduling Algorithms for Shared Fiber-Delay-Line Optical Packet Switches─ Part II: The Three-Stage Clos-Network Case,” J. Lightw. Technol., vol. 23, no. 4, Apr. 2005, pp.1601-1609.

[20]Mark Harris, “Optimizing Parallel Reduction in CUDA,” NVIDIA Developer Technology,2007.