國立陽明交通大學
資訊科學與工程研究所
碩士論文

Institute of Computer Science and Engineering
National Yang Ming Chiao Tung University
Master Thesis

# 採用 CUDA 圖型處理器 平行化改良 5G 軟體基地台之隨機存取通道流程

# An improved parallel implementation method of RACH procedure in 5G SDR system using CUDA GPU

研究生：王靖（Wang, Jing）

指導教授：許騰尹（Terng-Yin Hsu）

中華民國 一一○ 年 七月

July 2021

# 採用 CUDA 圖型處理器
# 平行化改良 5G 軟體基地台之隨機存取通道流程

# An improved parallel implementation method of RACH procedure in 5G SDR system using CUDA GPU


研 究 生：王靖　　Student： Wang, Jing

指導教授：許騰尹 博士　　Advisor： Dr. Terng-Yin Hsu

國立陽明交通大學

資訊科學與工程研究所


碩士論文

A Thesis
Submitted to Institute of Computer Science and Engineering
College of Computer Science
National Yang Ming Chiao Tung University
in partial Fulfillment of the Requirements
for the Degree of
Master of ○○
in
○○

July 2021
Taiwan, Republic of China

中華民國 一一○ 年 七月

# 摘　要

隨著 5G 逐漸於全球開始商轉，越來越多企業發現其中商機並相繼開發相關應用與服務，例如：無人機、物聯網、邊緣運算等，然而這些應用都需要基地台為其傳遞訊號才能正確運作，因此基地台本身的穩定與效能將是這一切的基礎。本論文即提出一改善方法以提升原基地台本身之運算效率使其能夠更穩定的提供服務。

無線行動網路近年快速發展，於是有軟體化基地台（Software-defined Radio, SDR）的概念被提出並運行提供服務，此概念即透過編寫軟體程式提供傳統基地台之服務，以應付行動網路技術規格之快速發展與變遷。本論文在此基礎之上針對基地台中提供使用者註冊接入網路與使用者裝置同步服務的隨機存取通道（Random Access Channel, RACH）流程，討論其傳統實作方法並提出一改善效率之方法與流程架構。本論文將研究使用圖型處理器（Graphics Processing Unit, GPU）加速平行 RACH 流程上的運算，並修改運算流程與方法使之更適合運行於 GPU。

透過本論文提出的架構設計，基地台的模擬測試運算執行時間可調降至大約原本的 10%～50%。本論文的架構亦提供彈性化設計，因此可一次處理多基地台接收之訊號，且由於本研究將所有運算拆開至不同運算單元上平行運算，所以即使需要處理的訊號增加，總處理時間也不會有太大的差異。藉此研究，軟體基地台運行時將能有更多閒餘的效能維持整體性之效能與穩定或是提供更多服務應用。

關鍵字：**隨機存取通道、統一計算架構、圖型處理器、第五代行動通訊新無線標準、軟體基地台**

# Abstract

With the commercialization of 5G in recent years globally, more and more companies have discovered business opportunities related to it. More and more related applications and services are developed or provided, such as drones, Internet of Things(IoT), edge computing, and more. However, the function of these applications requires base stations to work perfectly. Hence, the stability and performance of base stations form the backbone of the 5G relevant industries. In the following chapters, this paper presents a method to improve the performance and efficiency of the existing computation in the base station.

With the rapid development of the wireless mobile network in recent years, the concept of Software-defined Radio (SDR) has been proposed and conducted to provide services. This concept is to offer the traditional base station services through software programs. This idea also helps cope with the rapid development and changes in mobile network technical specifications. Based on this scheme, in the following studies, this article will introduce the traditional implementation method and propose an improved architecture for the RACH (Random Access Channel) process. Through the RACH process, the base station provides a way for the user equipment to register and access the mobile network and synchronize with it. This research is aimed to use the GPU (Graphics Processing Unit) to accelerate the computation of the RACH process, and to adjust the calculation process and algorithm to make it more suitable for the GPU.

Through the architecture design proposed in this paper, the execution time of RACH process simulation can be reduced to about 10% $\sim$ 50% of the original version. Due to the flexible architecture design, the signals received by multiple base stations can be processed simultaneously. Also, since this proposed method splits all operations into different computing units to compute in parallel, even if received signals that need to be processed are increased, the processing time will not be much different. As a result, base stations will have more backup performance to maintain overall availability and stability, while providing more application services.

Keywords: **RACH, CUDA, GPU, 5G New Radio, SDR**

# 目　錄

# 圖 目 錄

# 表 目 錄

# 第一章　緒論

## 1.1　研究背景與動機

自 2020 年 6 月 30 日台灣各行動通訊公司推出第五代行動通訊世代（5G）服務以來台灣的 5G 用戶數迅速成長，相關產業發展也隨之蓬勃。行動通訊技術最早發展於國家級的航空與國防工業，然而在技術逐年成長發展後，相關應用已更多樣化且轉向為一般民眾服務。第二代行動通訊技術推出時僅能提供行動通話與簡訊服務；發展至 4G 時不僅可透過行動通訊技術傳輸網路訊號，更可以透過網路技術提供行動通話的服務；而現今的 5G 則在向下相容原有的 4G 服務之基礎上，提供可乘載更大吞吐量的網路服務以提供現今人們無時無刻皆需使用網路的需求。也因為無線通訊技術趨於成熟，許多過去無法部屬有線網路的邊緣區域，現在都可以都過無線行動網路的服務連上網路，更有諸多服務應用藉此發展出來，如在過去因為行動網路只能專責服務一般民眾的上網需求，無法提供服務予其他應用，然而因為無線通訊技術的提升，行動網路已有多餘的效能可為其他應用服務，因此物聯網與邊緣運算等過去無法透過行動網路完成的服務開始出現。

據 Opensignal 於 2021 年 6 月發布的 5G 用戶體驗報告 [1] 指出：行動通訊公司目前提供最大可用率為 25.3%，且平均最快與最慢行動通訊公司服務的下載速度分別為 436.2 與 128.4 Mbps。由於 5G 服務較 4G 服務具傳輸速度更快、高頻寬、低延遲、低功耗等特性 [2]，因而有利發展大數據、人工智慧、物聯網等服務，進而帶動高品質視訊影音串流娛樂、智慧城市、自動駕駛、無人機等相關應用。隨著 5G 用戶數逐漸攀升且相關應用需適應高頻寬低延遲特性，行動網路基地台的處理效率也必須隨之提升。最簡單且直接的方法即為硬體升級：使用更高效能的訊號處理晶片以加速基地台的編解碼效率，雖然此方法效果良好，但是晶片的設計及製作成本極高，且重新佈署的人力更是不在話下，故此作法並不利產業持續更新升級。相反的，隨著近年硬體技術的發展，以往相當珍貴的運算資源變得唾手可得，於是近年逐漸推出一個新型態的設計概念：軟體基地台（Software-defined Radio, SDR）—透過個人電腦與無線傳輸（Radio Frequency, RF）介面即可提供手機行動網路服務 [3]。

一般行動網路基地台基地台所需提供的服務包含：接收／發送無限訊號、訊號編解碼、與核網（Core Network, CN）傳輸資料等；據此，軟體基地台的設計概念即為將接收／發送無線訊號的服務透過 RF 介面完成，其餘功能透過撰寫成軟體程式運行在一般個人電腦上。其中基地台在實體層（Physical Layer）所提供之服務即為訊號編解碼，此一服務之運算極大且需在規格所定時限內完成，然而軟體基地台畢竟是運行於 CPU（Central Processing Unit）與一般作業系統（Operating System, OS）之上，因此編解碼的效率相對於硬體較低，但相對的，軟體的重新修改、編譯、佈署，即為傳統基地台的硬體更新，此一流程可以大幅地降低傳統基地台的升級成本，未來也只需更新內部程式即可應付新版本的行動通訊標準。在這樣的優點之下，如何提升 CPU 的運算效能以符合規格要求，更甚追上硬體的運算效能，便成為最大議題；在此一架構下，最常提

出的優化改善方法多為使用 CPU 多執行緒的設計（Multithreading Programming）、針對 CPU 運算特性修改適合的演算法、系統效能優化等。

雖然一般 CPU 的運算效能在近年逐漸地提升，但其發展方向多為提升個別核心的運算能力，而非增加平行運算的能力。相對的，近年相當受人矚目的圖型處理器（Graphic Processing Unit, GPU）即致力在發展平行運算能力，雖個別核心的運算效能不及 CPU，但其可平行運算之單元則為 CPU 的好幾倍。在電腦上對於圖形、繪圖等運算多為矩陣相關的運算，而通常矩陣的單一元素可獨立計算，不涉及相鄰元素，故矩陣運算相當適合透過平行的方法加速，因此一開始，GPU 的開發旨在負責電腦上的繪圖相關運算，因而稱為圖型處理器。然而此一架構設計不僅可以做為圖型運算之用，更可使用於加速其他適合大量平行的運算，NVIDIA 更為此開發一系列工具方便一般開發者開發 GPU 相關程式，更有圖形處理器上之通用計算（General-purpose Computing on Graphics Processing Units, GPGPU or $GP^2U$）一詞以表示使用 GPU 進行大量適合平行的一般運算，如：大量相同運算、程序一致之計算。雖然 GPU 具備可大量平行運算之能力，但也有限制，如上述的個別運算核心能力較低，故須評估在部分條件下使用 CPU 多次運算，或許會更快；在 CPU 和 GPU 之間搬移資料的代價也相對高，若資料量不夠大，搬移時間或許以足夠讓 CPU 完成運算；GPU 的運算特性與 CPU 不同，GPU 適合浮點數，CPU 則對整數、位元操作運算較快，故須適當的切分工作以及重新設計相關演算法與計算。

基於以上說明，本論文將針對在軟體基地台中提供使用者接入網路的隨機存取通道（Random Access Channel, RACH）流程上提出一透過 GPU 實現改進架構，並提出相關於 GPU 實作上的改善演算法。希望能夠將現有的架構改為彈性的設計，使之能運作於各種通用運算的 GPU 且可應付大量的資料運算，最終使整體軟體基地台的運作更有效率且穩定。

## 1.2 研究貢獻

本篇論文的貢獻如下：

1. 針對軟體基地台中的 RACH 流程提出一透過 GPU 實現之改善架構。此一架構設計能將使整體運算時間降至原本的 10%～50%。
2. 透過模擬運行測試，改善架構不僅能彈性同時處理多接收來源之資料，其運算效率亦不受影響。
3. 新設計架構測試於不同圖型處理器，實際驗證新架構設計可不依賴於特定 GPU 規格要求。

## 1.3 章節概述

綜合以上所述，本篇論文將基於 5G NR 標準使用 GPU 以提升軟體基地台的效能為目標進行實驗與分析。本文各章節編排如下：

- 第一章為緒論，介紹本文的研究背景與相關應用，並且概述研究內容與貢獻。
- 第二章為背景知識，分別介紹本論文欲改善的問題與後續實作設計所需各項背景知識，例如，本論文採用的 5G NR 實體層規格、傳統實作的方法與原理、GPU 軟硬體設計方法與對應等。
- 第三章為研究架構設計，說明本論文所提出的優化設計方法與演算法應用，並解釋相關設計的原因與考量。
- 第四章為實驗結果與討論分析，提供優化後的各項實驗結果與討論說明。
- 第五章為結論與未來展望，提出本文的研究總結與未來相關研究之建議與參考方向。

# 第二章　背景知識

## 2.1　5G NR 規格介紹

第五代行動通訊世代（5G）服務中的 Physical Layer 規格主要來自於第三代合作夥伴計畫（3rd Generation Partnership Project, 3GPP）的規劃與設計。為使 5G 能夠向下相容 4G 網路且適應各種情況都能有效運作，其規格定義多樣複雜且不斷更新，因此以下將簡介 5G 實體層規格、本論文所參照之規格、規格所定之傳輸介面與訊號處理流程。

### 2.1.1　5G 實體層規格簡介

依 3GPP 定義，5G 基地台稱為 gNB（Next Generation Node B），其與使用者裝置（User Equipment, UE）間的通訊可分為上行鏈結（Uplink）與下行鏈結（Downlink），主要以 UE 角度做區分，Uplink 為 UE 傳送資料至 gNB 的連線，Downlink 則相反，表示 gNB 傳輸訊號至 UE 的連線。實體層為 OSI 模型（Open System Interconnection Model, OSI Model）當中的最底層，主要負責實際訊號的傳輸。行動網路的實體層相關規格定義在 3GPP 所釋出之規格文件中的 38.201、38.202 與 38.211～38.215，各規格文件關係與流程如圖2.1所示 [4]，本篇論文則主要探討 38.211 第 15 版 [5] 於軟體上實作與效能改善分析。

為了讓基地台與 UE 溝通順暢，通訊內容不只包含欲傳輸的資料，亦包含各種控制訊號以調整編解碼傳輸方式以應付多變的無線傳輸環境。因此 3GPP 定義多種傳輸資源與傳輸內容格式的介面，此即為實體層通道（Physical Channel），在 Uplink 有乘載實質傳輸資料的實體層上行共享通道（Physical Uplink Shared Channel, PUSCH）、乘載控制資訊的實體層上行控制通道（Physical Uplink Control Channel, PUCCH）與管理 UE 接入網路與同步的實體層隨機接入通道（Physical Random Access Channel, PRACH）[6]，本論文主要探討 PRACH 傳輸與格式。

上述各 Physical Channel 為 3GPP 對頻域（frequency domain）、時域（time domain）資源之分配與其溝通介面之定義。另外對於傳輸、接收、處理 Physical Channel 中的資料方法，則定義於傳輸通道（Transport Channel），其關係如圖表2.2所示，本論文主要探討 RACH 於軟體上之實作與改善優化。

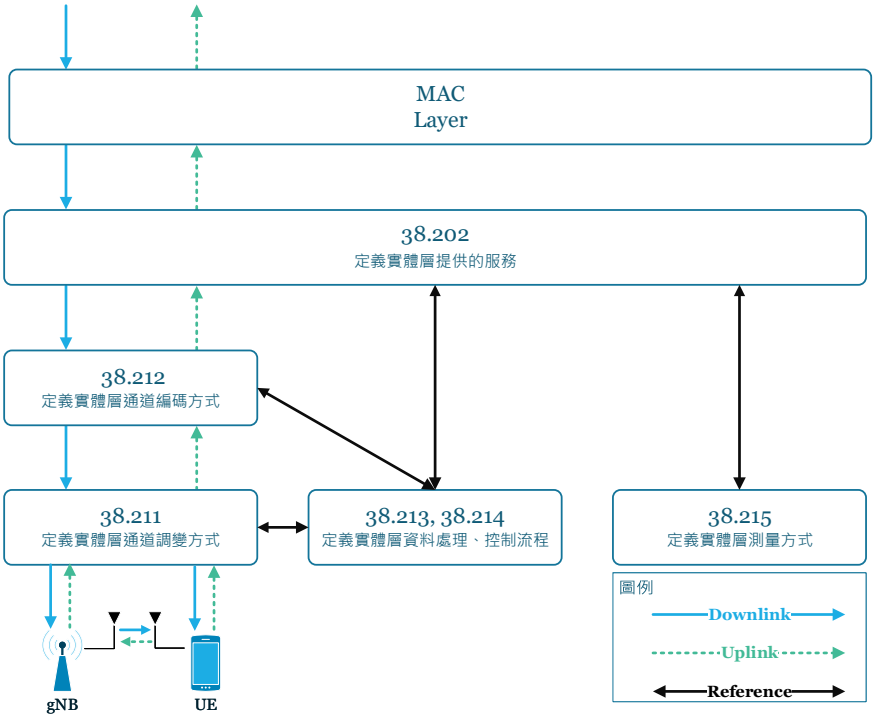
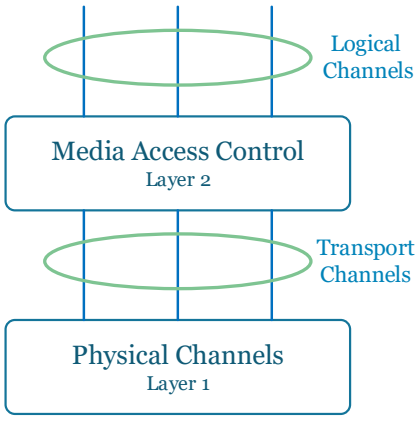



圖 2.1：5G 流程與實體層規格關係圖




(a)

| Physical Channel | Transport Channel |
|---|---|
| PRACH | RACH |
| PUCCH | N/A |
| PUSCH | UL_SCH |

(b) [1]

圖 2.2：Physical Channel 與 Transport Channel 對應關係

### 2.1.2 5G 基地台佈署架構選項

在過去發展 2G～4G 時，行動基地台一直被視為一獨立作業節點，不曾將其細部作業流程拆開分工，而在發展 5G 的新無線技術（New Radio, NR）時，為了能夠做到彈性化與虛擬化，廠商可依需求實作，5G 基地台作業流程被拆分成不同作業單元。依 3GPP 38.801 所述 [8]，5G 基地台訊號處理流程可切分至中央單元（Central Unit, CU）、

[1] 此表節錄自 3GPP 38.202[7] Table 6.1-1。

分散單元（Distributed Unit, DU）與無線電單元（Radio Unit, RU）不同節點上。而各節點則依規格所訂之介面規格溝通。各節點負責功能說明如下 [9]：

- RU：實際無線訊號的收發。
- DU：即時的 PHY 層和資料鏈結層（Data Link Layer）的排程作業，工作具有時效性。工作排程作受 CU 控制。
- CU：較不具時效性的網路層（Network Layer）以上的作業。

以上為各節點大略負責項目，實際各節點如何分配工作，則使用選項（Option）1～8 切分。Option 切分方法如圖2.3[8]。因本論文旨在探討無線網路實體層的實作與優化，故本論文將實作及改善 RACH 流程於 Low-PHY 與 High-PHY 的實作，即為 DU-RU 使用 Option 8 切分的版本。

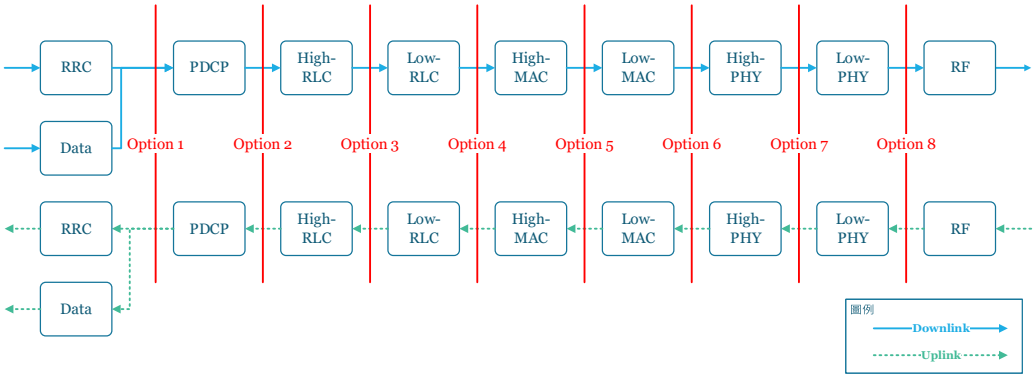



圖 2.3：5G 基地台佈署架構選項

## 2.2 PRACH 與 RACH 說明

因本論文主要將探討 RACH 在軟體上的實作與分析，以下將介紹 PRACH 的功能、格式與 RACH 的訊號處理流程。

### 2.2.1 PRACH 功能簡介

PRACH 全名為實體層隨機接入通道（Physical Random Access Channel, PRACH），顧名思義其使用時機主要在 UE 欲連接上行動網路或欲與基地台同步時所使用之介面。藉此介面，Core Network 端可得知有一 UE 欲連上網路，gNB 可對其回應以分配後續連線與資料傳輸所需資源。詳細流程可藉由圖2.4輔助說明。

1. 基地台會在固定時間廣播，發出系統資訊（System Information）。
2. UE 會在接上網前等待接收系統資訊，收到後會發送一段由系統資訊為參數所計算出的前導碼（Preamble）。

3. 基地台會在固定時間偵測是否有 UE 發送 Preamble，若有偵測到則發送回應（Response），告知所分配之時間與頻譜資源。

4. UE 在接收到確認回應後即可發送後續的資料。

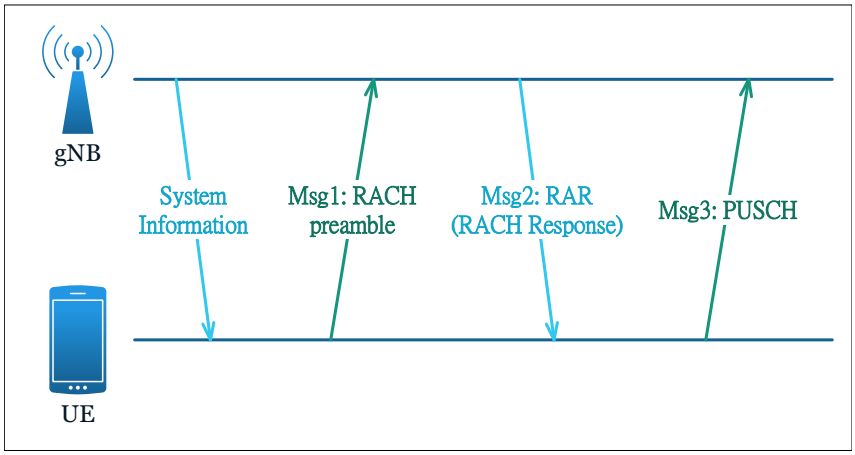



圖 2.4：使用者裝置接入行動網路流程

## 2.2.2 PRACH 介面說明

如前小節所述，UE 會發送一段由系統參數與隨機特徵值（signature）所計算的 Preamble，此 Preamble 即由 Zadoff-Chu 序列（Zadoff-Chu Sequence, ZC Sequence）組成，以下將介紹 PRACH 格式與 ZC Sequence 之生成方式與其特性。

### PRACH 格式

PRACH 的格式如圖2.5所示，在 time domain 上由循環前綴 (Cyclic Prefix)、前導碼序列 (Preamble Sequence)、保護帶（Guard Period）組成一組 Preamble，而 frequency domain 上則含有多組 Preamble。

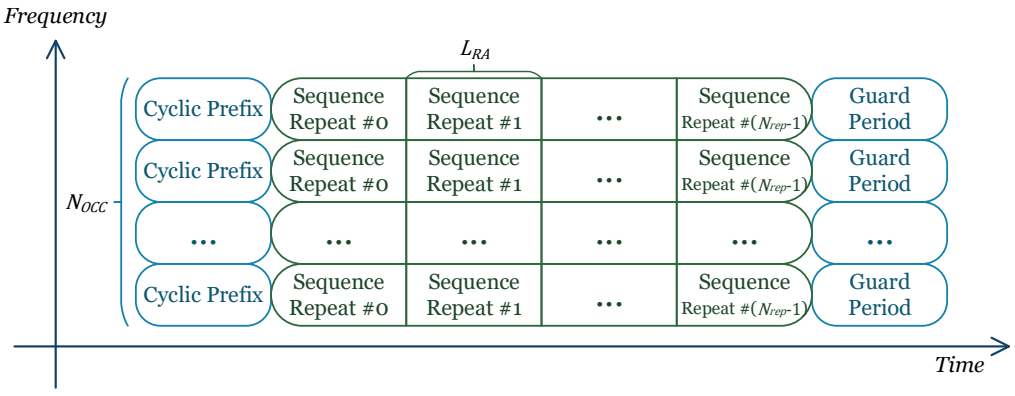



圖 2.5：Preamble 的組成

時域上的 Preamble 中段由 Preamble Sequence 組成，其中一 Sequence 的長度記為 $L_{RA}$，$L_{RA}$ 值可為 839 或 139，長度 839 之序列記為長序列（Long Preamble）；長度 139 之序列則記為短序列（Short Preamble），Long Preamble 的格式可再分為格式（Format）0/1/2/3；Short Preamble 的格式亦可再分為 Format A1/A2/A3/B1/B2/B3/B4/C0/C2[5]。圖2.5中顯示 Preamble Sequence 會重複（Repeat）數次，重複次數記為 $N_{rep}$，其值可藉由 Format 與表2.1得知。

| Format | $L_{RA}$ | $N_{rep}$ |
|---|---|---|
| 0 | 839 | 1 |
| 1 | 839 | 2 |
| 2 | 839 | 4 |
| 3 | 839 | 4 |

| Format | $L_{RA}$ | $N_{rep}$ |
|---|---|---|
| A1 | 139 | 2 |
| A2 | 139 | 4 |
| A3 | 139 | 6 |
| B1 | 139 | 2 |
| B2 | 139 | 4 |
| B3 | 139 | 6 |
| B4 | 139 | 12 |
| C0 | 139 | 1 |
| C2 | 139 | 4 |

表 2.1：Preamble 格式[2]

圖2.5中的垂直軸上有多個頻域傳送資源，此為 3GPP 於 5G NR 實體層規格新設計[5]。過去在 4G LTE 規格中 [10]，一 PRACH 時間區間（PRACH slot），僅能有一頻域資源可傳送 PRACH 資訊。而在 5G 中，gNB 可以依設定[3]在一 PRACH Slot 內於頻域上設定多個 PRACH 傳送資源（Occasion）。當設定的數量越多，UE 可以在一 PRACH Slot 內選擇傳送的 PRACH 的空間越多，如此在多個 UE 的情況下，UE 所挑選之 signature 發生碰撞的機率越低。而此設定的數量記為 $N_{OCC}$。

**Preamble Sequence 生成**

UE 在連上網路前或與基地台同步前，會不斷聆聽來自基地台所發出的系統資訊，此資訊包含：PRACH 配置參數（PRACH Configuration Index, `prach_ConfigIndex`）、邏輯根數列（Logical Root Sequence Index, `rootSequenceIndex`）、`zeroCorrelationZoneConfig`、可發送 Preamble 的時間點。透過這些參數，UE 可生成 64 組 Preamble Sequence，64 組 Preamble Sequence 皆為 Zadoff–Chu 序列（Zadoff–Chu Sequence, ZC Sequence），而 UE 會從中隨機擇一傳送。

$$x_u(i) = e^{-j\frac{\pi ui(i+1)}{L_{RA}}}, i = 0, 1, ..., L_{RA} - 1 \quad \text{(式 2.1)}$$

由式 2.1可得一組長度為 $L_{RA}$ 在 time domain 的 ZC Sequence，此一 Sequence 稱為根序列（Root Sequence）。UE 可藉由 `prach_ConfigIndex` 查表[4]取得 Format 進而得知 $N_{rep}$ 與 $L_{RA}$。$u$ 值為根序列數（Root Sequence Number、`rootSequenceNumber`），可藉

[2] 節錄自 3GPP 38.211[5] Table 6.3.3.1-1、Table 6.3.3.1-2。
[3] 3GPP 38.211[5] Table 6.3.3.2-2～Table 6.3.3.2-4
[4] 3GPP 38.211[5] Table 6.3.3.2-2～Table 6.3.3.2-4

由 rootSequenceIndex 查表[5]取得。其餘 Preamble Sequence 則使用此 Root Sequence 不斷做循環位移（Cyclic Shift）生成。

$$x_{u,v}(n) = x_u((n + C_v) \mod L_{RA}), n = 0, 1, 2, ..., L_{RA} - 1$$
$$C_v = \begin{cases} vN_{CS} & , v = 0, 1, ..., \lfloor L_{RA}/N_{CS} \rfloor - 1, N_{CS} \neq 0 \\ 0 & , N_{CS} = 0 \end{cases} \quad \text{(式 2.2)}$$

透過式 2.2，可由 Root Sequence 產生多組 Preamble Sequence，實際可產生的數量由 $N_{CS}$ 控制。$N_{CS}$ 可由 zeroCorrelationZoneConfig 查表[6]取得。若產生的數量不及 64 組，則增加 rootSequenceIndex，查表取得新 rootSequenceNumber 再生成一新 Root Sequence 做 Cyclic Shift，直到產生 64 組 Preamble Sequence 為止。

$$y_{u,v}(n) = \sum_{m=0}^{L_{RA}-1} x_{u,v}(m) \cdot e^{-j\frac{2\pi mn}{L_{RA}}}, n = 0, 1, 2, ..., L_{RA} - 1 \quad \text{(式 2.3)}$$

最後，再透過式 2.3將 time domain 的序列展開到 frequency domain。

至此，UE 將生成 64 組 Preamble Sequence，按生成順序編號為 [0, 63]，UE 將會隨機擇一並重複 $N_{rep}$ 次後加上 Cyclic Pregix 傳送，UE 所挑選的 Preamble Sequence 之編號即為 signature。

**Zadoff-Chu 序列特性**

依前項所述，Preamble Sequence 由 ZC Sequence 組成。ZC Sequence 取名自 [11] 與 [12] 之作者。因 ZC Sequence 具有振幅恆定且序列相關函數為零（Constant Amplitude Zero Autocorrelation, CAZAC）的特性 [11][12][13]，因此衍伸出許多在訊號處理上的優勢，進而被使用於 RACH 估測 [14]。

具 CAZAC 特性之序列對於 RACH 估測的最大優勢即為可透過 Cyclic Shift 產生多組彼此為正交（Orthohonal）的序列，且若將一經過 $N$ 點移位後的 Sequence 與原本的 Sequence 做相關函數（Correlation）後，其結果將於第 $N$ 點上出現最大值，其餘區域則為零相關區（Zero Correlation Zone, ZCZ）[15]。透過此特性，RACH 流程上並不需要逐一比對以找出 signature，只需要將收到的訊號與原始 Root Sequence 做 Correlation，即可找到對應的 signature。且因為移位過後的序列彼此為正交，故即使有多 UE 於同時域與同頻域傳送 Preamble，只要其所挑選之 signature 不同亦不會受彼此干擾。

### 2.2.3 傳統 RACH 訊號處理流程

如前小節所述，UE 會發送一段由系統參數與 signature 所計算的 Preamble，以下將介紹基地台如何透過收到的訊號解出是否有 UE 發送 Preamble，並正確找出 UE 所選擇的 signature。圖2.6上下分為 Low-PHY 與 High-PHY 二部分，此圖即為本論文欲比較之傳統架構程式與 RACH 流程之處理方式 [16][17]。

[5]3GPP 38.211[5] Table 6.3.3.1-3、Table 6.3.3.1-4
[6]3GPP 38.211[5] Table 6.3.3.1-5～Table 6.3.3.1-7

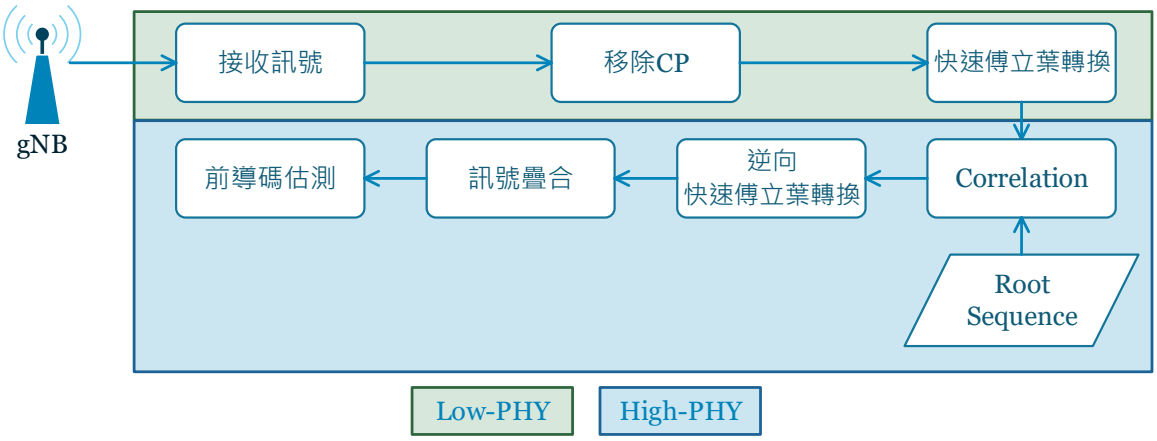



圖 2.6：RACH 訊號處理流程

**移除 Cyclic Prefix**

基地台收到的訊號格式就如圖2.5所示為一完整的 Premable，因此流程第一步即為移除 Cyclic Prefix（CP Removal），僅留下中間後續需要估測的 Preamble Sequence 部分。

**快速傅立葉轉換**

將收到的訊號使用快速傅立葉轉換（Fast Fourier Transform, FFT）（式 2.4），將訊號自 time domain 轉換至 frequency domain。

$$F(k) = \sum_{n=0}^{N_{fft}-1} f(n) \cdot e^{-j\frac{2\pi kn}{N_{fft}}}, k = 0, 1, ..., N_{fft} - 1$$
$$N_{fft} = \begin{cases} 49152 & \text{, Long Preamble} \\ 2048 & \text{, Short Preamble} \end{cases} \quad \text{(式 2.4)}$$

**Correlation**

RACH 流程主要目的即為找出此段接收到之訊號是否含有 UE 發送的 Preamble，且其所選用之 signature 為何。透過章節2.2.2關於 ZC Sequence 特性之說明，只要將收到的訊號與 Root Sequence 做 Correlation 即可判斷此訊號是否自此 Root Sequence 做 Cyclic Shift 產出與其 $C_v$ 值為何。而 Root Seqeunce 可於基地台啟動初始化時先行計算，後續在估測時，僅需帶入 Correlation 計算。Correlation 計算方式呈現於式 2.5，其中 $\overline{F(n)}$ 為 $F(n)$ 之共軛複數。

$$C_u(n) = \overline{F(n)} \times y_u(n), n = 0, 1, ..., L_{RA} - 1 \quad \text{(式 2.5)}$$
$$y_u : \text{Root Sequence in frequence domain}$$

**反向快速傅立葉轉換**

將 Correlation 的結果使用反向快速傅立葉轉換（Inverse Fast Fourier Transform, IFFT）（式 2.6）自 frequency domain 轉回 time domain。

$$c_u(k) = \sum_{n=0}^{N_{ifft}-1} C_u(n) \cdot e^{-j\frac{2\pi kn}{N_{ifft}}}, k = 0, 1, ..., N_{ifft} - 1$$

$$N_{ifft} = \begin{cases} 2048 & \text{, Long Preamble} \\ 1024 & \text{, Short Preamble} \end{cases} \quad \text{(式 2.6)}$$

**訊號疊合**

在此將代表相同資訊的資料加總平均，即先取所有輸入訊號的功率（power），再將一支天線內收到所有重複的訊號加總平均，最後再將所有天線的訊號加總平均（式 2.7）。

$$Y_r(k) = \frac{1}{N_{ant}} \sum_{a=0}^{N_{ant}-1} \left( \frac{1}{N_{rep}-1} \sum_{r=0}^{N_{rep}-1} |c_u(k, r, a)|^2 \right), k = 0, 1, ..., N_{ifft} - 1 \quad \text{(式 2.7)}$$

$N_{ant}$：基地台總天線數

**前導碼估測**

最後判斷此次接收訊號中是否有 UE 傳送 Preamble 及判定該 UE 所選擇之 signature 為何。方法即為將前項輸出之結果與事先計算之臨界點（threshold）做比較，若功率超過 threshold 即視為有 UE 嘗試傳送該 Presmable Sequence。接下來再看該尖峰（peak）出現在哪一段時間區間（Window），Window 長度定義如式 2.8[16]，根據章節2.2.2，個別 Window 即對應至 $C_v$，進而可求對應之 signature。

$$\text{Window 長度} = N_{CS} \cdot \left\lfloor \frac{N_{IFFT}}{L_{RA}} \right\rfloor \quad \text{(式 2.8)}$$

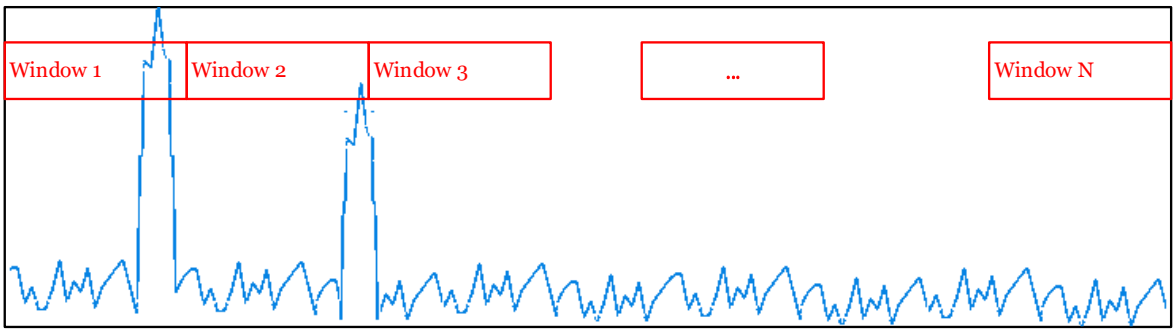



圖 2.7：Preamble 估測示意圖

除了需要偵測是否有 UE 傳送 Preamble 外，PRACH 亦有 UE 與 gNB 同步之功能。由於實際的傳輸環境相當複雜，使得 gNB 在收到 UE 的訊息可能會經過多種路徑反射再由 gNB 接收到並處理，由於此種無線訊號特性使 UE 傳送訊號的時間點可能不會對上 gNB 實際接收到訊號的時間點。在前段找出 peak 之後，除了可得其所對應之 $C_v$ 值外，亦可藉由該 peak 出現位置與其預期位置相差多少得一 $TA$ 值，此值將於後續 gNB 發送 PRACH Response 時傳送給 UE，UE 可藉此調整發送訊號之時機以達到同步之目的 [18]。

## 2.3 GPU 程式簡介

因本論文將使用圖型處理器（Graphics Processing Unit, GPU）平行化加速軟體程式，以下將簡介現有常見的 GPU 程式編寫方法與本論文所選擇之方法並深入介紹該方法之硬體與軟體架構。

### 2.3.1 GPU 程式編寫方法介紹

為了能夠透過可程式化的方式控制 GPU 運算，各生產 GPU 的公司會提供相應的工具方便工程師開發 GPU 程式，如 NVIDIA® 公司所提供的統一計算架構（Compute Unified Device Architecture, CUDA）。又因各 GPU 廠商在硬體上的設計架構不盡相同，若需工程師針對不同廠商編寫不同的程式語言，對於平行程式開發的成本會相當的高，於是出現開放計算語言（Open Computing Language, OpenCL）通用框架（framework）。

**CUDA**

CUDA® 是由 NVIDIA® 所開發一系列平行計算工具，以實現圖形處理器上之通用計算（General-purpose Computing on Graphics Processing Units, GPGPU or GP$^2$U）。雖然此系列工具皆專為 NVIDIA 出品之 GPU 開發，但其軟體與硬體具有最佳之相容性。且工程師得以借助 CUDA 使用高階語言描述 GPU 運作指令與其記憶體。開發時，工程師可以使用 C/C++、Python、JAVA 等流行語言控制 CPU 行為，而對於 GPU 的相關操作，CUDA 則有提供對應語言之擴充功能（extension）[19]。綜合以上，雖然 CUDA 的執行範圍有限，但因其公司規模與開發效率，此工具仍為常見之 GPU 程式開發系列工具。本論文即使用此一工具鍊（toolkit）進行軟體基地台實作。

**OpenCL**

OpenCL™ 為一透過 Khronos Group® 發布之跨平台運算框架，該組織僅定義在軟體上的架構與應用程式介面（Application Programming Interface, API）行為參數，故開發者能夠按其架構編寫 GPU 程式並與主程式溝通，實際 GPU 硬體架構與 API 行為則由各 GPU 廠商實作，提供開發者使用 [20]。

### 2.3.2 CUDA 軟體架構

在軟體運算資源上，CUDA 最小的執行單位為執行緒（Thread），Thread 會由區塊（Block）管理，Block 再由網格 (Grid）管理，整體架構如圖2.8。一個 Block 內可以使用最多 1024 個 Thread，這 1024 個 Thread 可以透過速度較快的共享記憶體（Shared memory）溝通；Block 之間若要溝通則需要透過速度較慢的全域記憶體（Global memory）；因目前 CUDA 對於個別啟動的工作（work），只允許啟動一個 Grid，故 Grid 可視為一個工作的運算資源單位。另外，Block 和 Thread 都可以視管理需求，描述為三維的運算資源，但一個 Block 內的 Thread 總數限制仍為 1024 個。

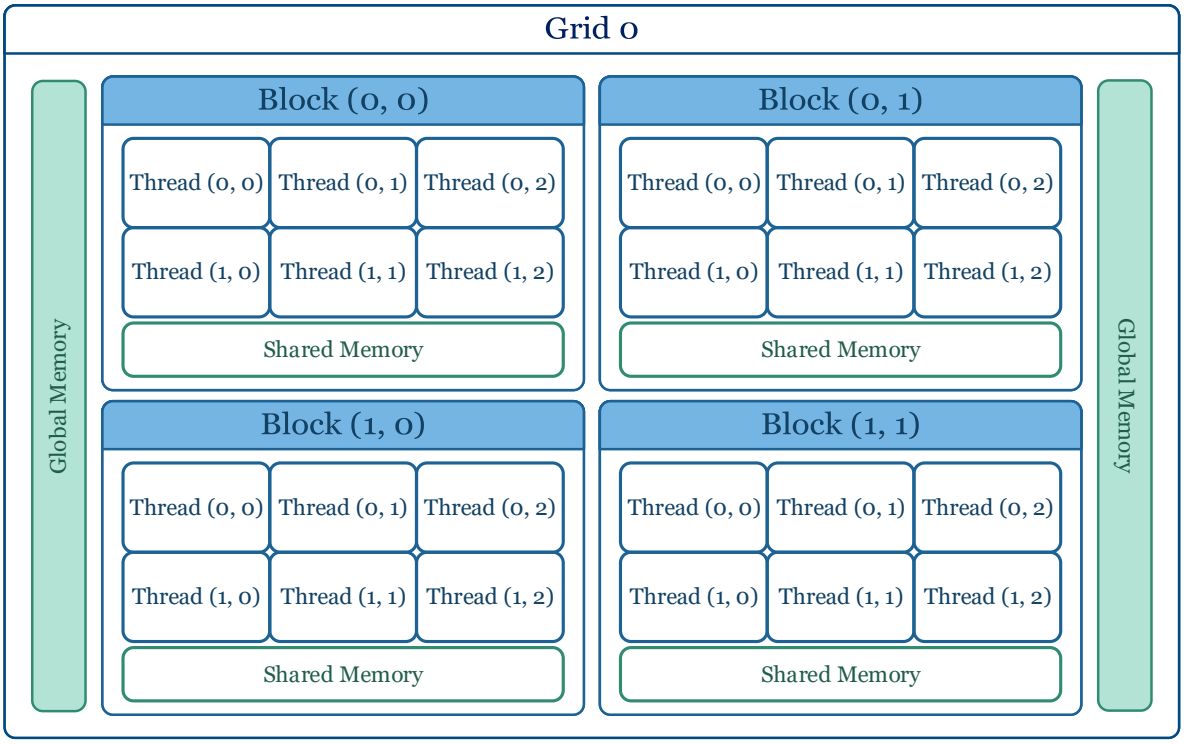



圖 2.8：CUDA 軟體架構

GPU 與 CPU 的協作流程可表示如圖2.9，一開始程式由 CPU 啟動，當 CPU 流程執行到需要 GPU 的協助時，會將 GPU 運算所需要的資料複製到 GPU 的 Global Memory 上，接下來 CPU 會呼叫編寫給 GPU 執行的程式，此段予 GPU 執行的程式稱為核心函式（kernel function）。在呼叫 kernel function 時，亦須提供核心設置參數（kernel configuration parameters），設置參數包含：要啟動多少 Block、每個 Block 內需執行多少 Thread 等，透過這些參數告知 GPU 該如何分配運算資源。最後，在 GPU 運算完畢後，亦須將結果傳回 CPU。在 GPU 運算的同時，CPU 亦可進行其他適合 CPU 的運算。

雖然在 Grid 當中內的運算資源可以同時平行運作，但是 Grid 和 Grid 之間，亦即不同的 work 之間，卻無法同步執行，因此，除了上述的所提的資源管理單元外，CUDA 還有提供稱為串流（stream）的管理功能，可以將不同的 work 賦予不同的 stream 編號，同一 stream 編號的工作會依序執行，而不同 stream 編號的 work，GPU 會視工作負載同步處理。如部分工作正在等待資料傳輸讀寫，其運算單元即處於閒置狀態，此時將調用此部分資源支援其他 work 的運算 [21]。

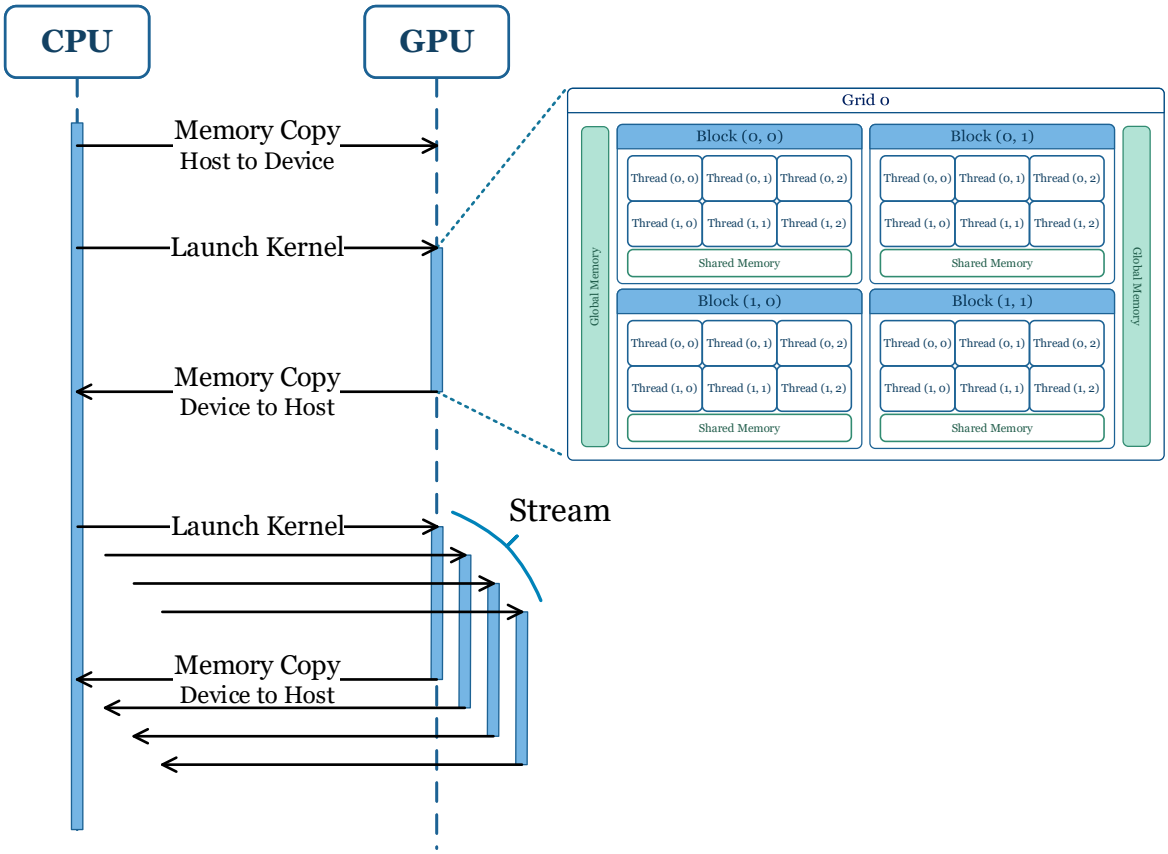



圖 2.9：CPU 與 CUDA GPU 協作流程

### 2.3.3 CUDA 硬體架構

雖然我們所撰寫的程式必須按照上述的軟體架購編寫，並使用 CUDA toolkit 編譯交由硬體執行，但若能理解軟體與硬體之間的對應與硬體架構特性，則可以將此觀念帶入軟體設計當中，進而提升運算效率。以下將依序介紹主要硬體元件與其架構。

#### SM

GPU 當中最重要的計算資源管理單位為串流多元處理器（Stream Multiprocessor, SM），其架構如圖2.10所示，在 SM 當中有 1、2 個 SM 處理塊（SM Processing Block, SPB），其中含有多種最小硬體運算資源：

- 核心（Core）：又稱串流處理器（Stream Processor, SP），所有的運算指令都會交由給 Core 處理，浮點數運算精準度為單精度。
- 雙精度核心（Double Precision Unit, DP Unit）：可執行雙精度的浮點數運算。
- 載入與儲存元件（Load/Store, LD/ST）：負責處理與 Memory 存取的工作。

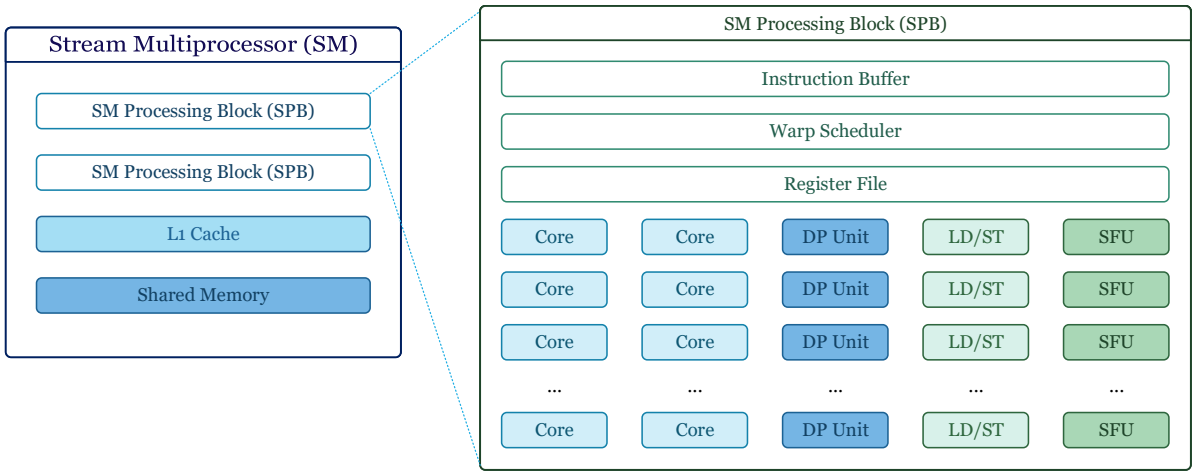



圖 2.10：SM 架構

- 特別運算單元（Special Function Unit, SFU）：負責處理如 `sin`、`cos`、開根號等特別數學運算。

這些運算單元的數量比例則會因不同 GPU 核心而有不同的設計規劃，本論文所使用之 GPU 的詳細規格會於後續實驗環境中介紹。

**Warp**

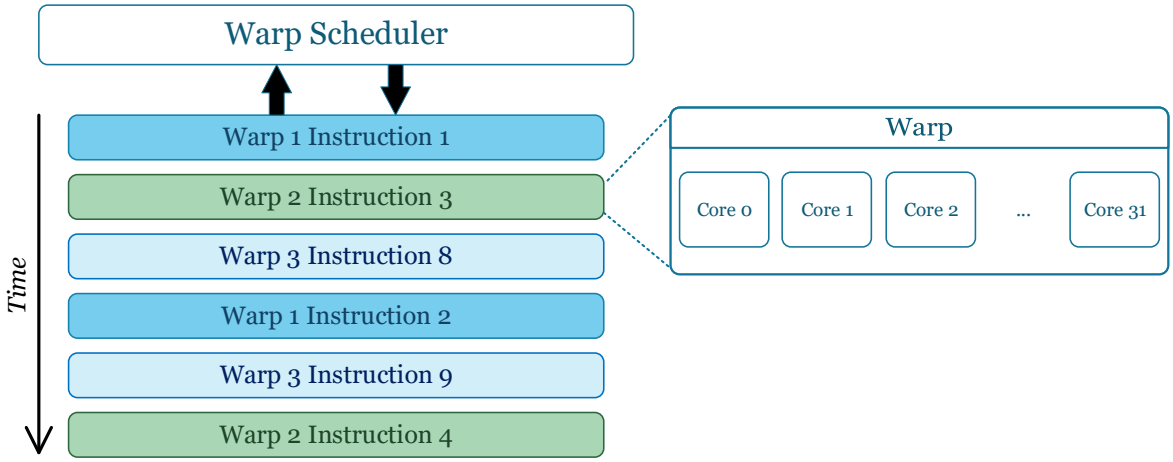



圖 2.11：Warp 執行流程架構

在透過軟體程式啟動 GPU 時，GPU 會將需要執行同樣指令流程的 Core 安排在一個 Warp 當中，一個 Warp 內會啟動 32 個 Core，這 32 個 Core 會在安排的時間同步執行，進行單一指令多種資料（Single Instruction Multiple Data, SIMD）的運算，而 Warp 就是 GPU 執行時的最小排程單位。這些 Warp 會交由 Warp Scheduler 安排執行順序，流程架構如2.11。而 Warp 不一定會一次把所有指令都執行完，當遇到如存取 Global Memory 需要等待的指令時，Warp Scheduler 即會切換其他 Warp 運算，藉此避免等待而

浪費時間。而 Warp 間互相切換的成本相當低，因為 Core 並沒有自己的暫存器，而是共用 SM 內部的暫存器空間（Register File），故當需要做 Warp 切換時，只需將 Register 指向 Register File 的不同區塊 [22]，如圖2.12所示。因此最好的 GPU 執行設置為 SM 有足夠多的 Core 可以交替執行，輪流運算或存取資料，避免 SM 所有資源都在等待運算或是等待資料。

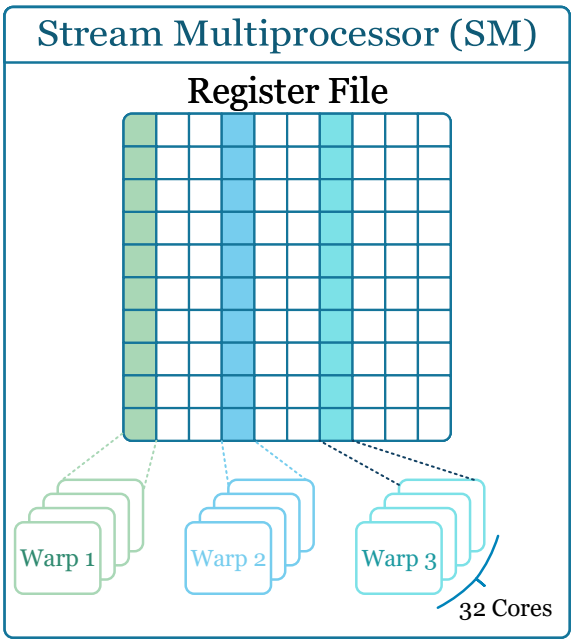



圖 2.12：Register File 切換示意圖

### 2.3.4 軟體與硬體運算資源對應

CUDA 中的軟硬體運算元件並無標準一對一對應，但大致可表示為2.13，軟體上的 Thread 對應至硬體上的 Core；軟體上的 Block 對應至硬體上的 SM；軟體上的 Grid 對應至整個 GPU。但是這種對應並不能完整表達，因為實際硬體中的 SM 內的 Core 數量不如軟體上 Block 中可有 1024 個 Thread，因此當在軟體設定的 Thread 數量過多時，CUDA 硬體會啟動虛擬化機制，讓空閒或是在等待資料讀寫的 Core 改處理其他 Thread 的運算。又如在軟體上容許的 Block 數量也遠大於實體 SM 的數量，若在啟動時設定的 Block 數量大於 SM 數量時，亦會啟動虛擬化機制，一個 SM 可能會輪流執行多個 Block 的任務，或是多個 Block 會被放進一個 SM 內同步執行。最後因為在軟體上還有提供 stream 可以同步執行多個 work 的功能，GPU 也會尋找有空閒的 SM 輪流計算。

綜合上述，在編寫 GPU 軟體程式時有諸多需考量及優化的項目：

- 雖然在實際硬體中一個 SM 內的 Core 可以輪流運算，但是一個 SM 所包含的 Shared Memory 與 register 是固定也無法切換做使用的，所以每個 Block 配置的 Thread 越多，每個 Thread 所能使用的 register 越少。而每個 Block 所使用的 Shared Memory 越多，一個 SM 能同時執行的 Block 就越少。

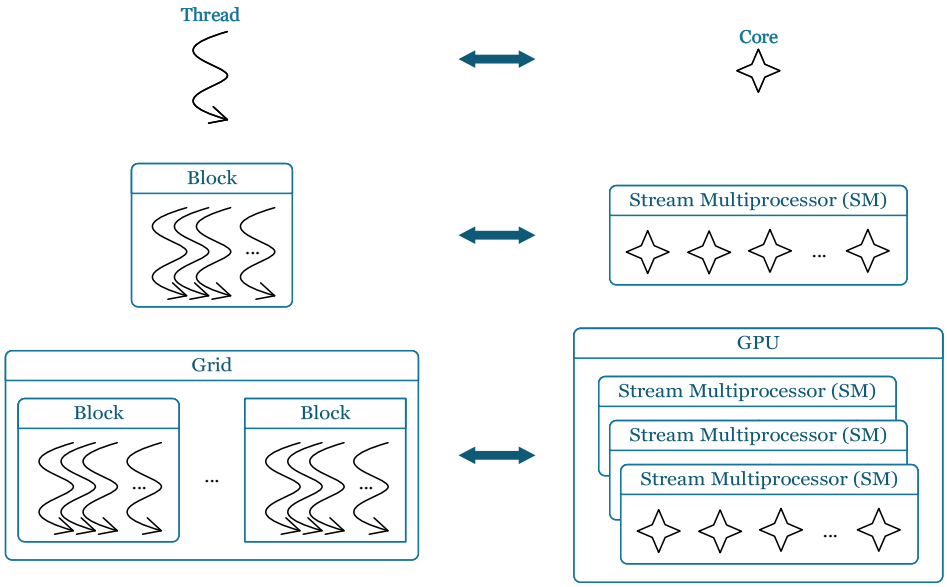



圖 2.13：CUDA 軟硬體運算元件對應

- 由於 SM 中的 Warp 切換幾乎不耗時間，要根據程式設計與硬體設置適當的 Block 與 Thread，藉以隱藏需花費較久的指令的延遲，但亦須注意資源的使用。
- 在同一 Warp 內的 Core 所執行的指令必須相同，故對應到軟體設計時，如程式碼中常有的 `if/else`，即會造成同個 Block 中的 Thread 所執行的程式碼不同，進而造成等待，此即為 Warp 分歧（Warp Divergence）問題 [23]。

### 2.3.5 記憶體架構

為了要讓多筆資料同時運算，GPU 在硬體上有對應的記憶體階層（Memory Hierarchy）設計 [24]，在程式編寫上也有基於實體記憶體之抽象表示與使用。

#### 實體記憶體

CUDA 實體記憶體架構可參考圖2.14。在最外層 GPU 透過快捷外設部件互連標準第三版（Peripheral Component Interconnect Express 3.0, PCIe 3.0）與 CPU 溝通。CPU 在喚醒 GPU 運算前，必須先將欲運算的資料複製到 GPU 的 DRAM 上，後續在其上的運算元件才可以使用這些輸入資料。

DRAM 可對應至軟體上的 Global Memory，DRAM 為在 GPU 上最大但是最慢的全域記憶體，因此減少讀取 DRAM 次數是一大優化重點 [23]。當記憶體控制器（Memory Controller）存取 DRAM 時，會一次存取連續且對齊大小為 32～128 byte 的區段記憶體，而如果多個 Core 所存取的記憶體區段是連續的，Memory Controller 會將這些請求合併為一次對 DRAM 的存取 [24]。因此最佳存取方法即將同步運算的 Core 所需要之資料擺放在一起並對齊的位置，此優化方法則稱為合併記憶體存取（Coalescing memory access）[23][25]。以圖2.15為例，最左邊為為最佳存取情境，可以在一次 DRAM 存取就取得所有資料；中間情境雖然是存取連續的資料，但是因為存取並沒有對齊，所以

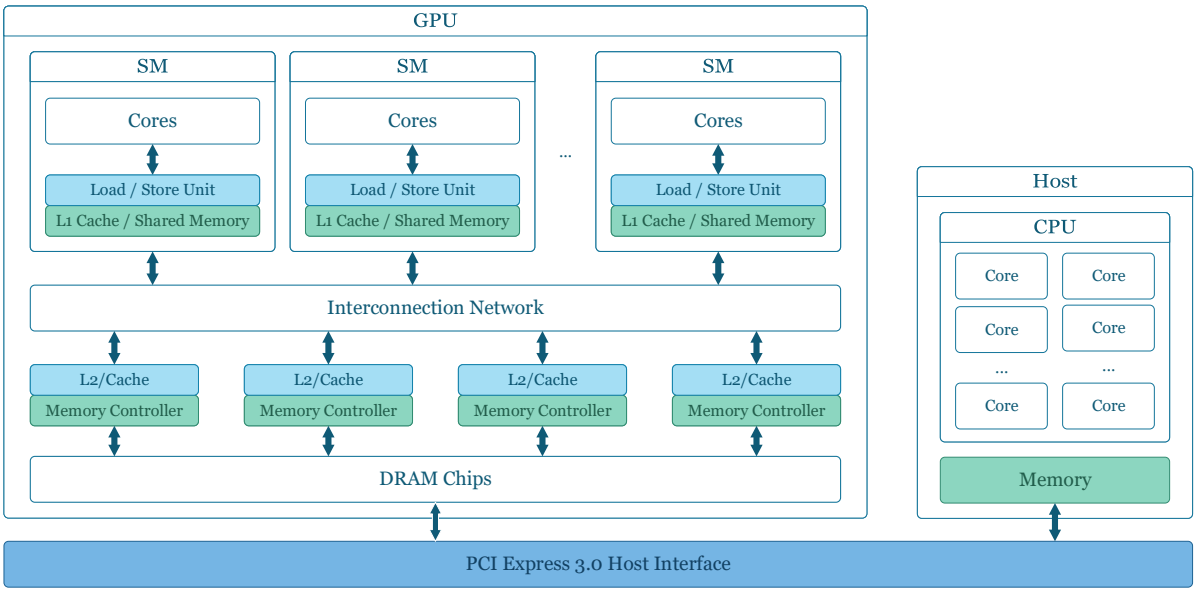



圖 2.14：CUDA GPU 記憶體階層架構

需要二次存取才能取得所有資料；最右邊情境為最差情況，因為存取沒有對齊也不連續，故需要三次存取才能取得所有資料。

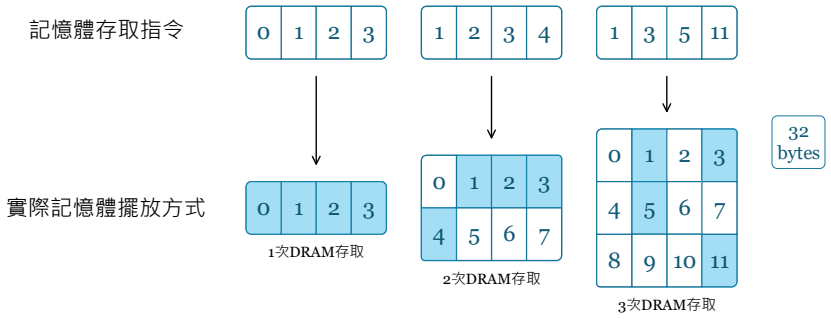



圖 2.15：合併記憶體存取

相較於 DRAM 為 off-chip memory，在每個 SM 中還有一個 on-chip memory，稱為 Shared Memory 與軟體上的名稱相同，速度較 DRAM 快上許多。而 Shared Memory 主要是提供給同個 SM 中的 Core 交換資料，因此為了能夠讓每個 Core 拿到所需資料的時間越短越好，Shared Memory 的存取機制與 DRAM 大不相同。存取機制如圖2.16所示，Shared Memory 的存取不再是將多個請求組成依連續資料的存取，而是將需要存取的 Shared Memory 切成一樣大小的區塊，此一區塊稱為 bank，每次記憶體存取可有多個 bank 同時執行記憶提存取，但若要在一次存取當中若存取一個 bank 當中的不同位置，則會被拆開成二次的存取，此即稱為 bank 衝突（bank conflict）[24]，但是有一例外為：一個 Warp 當中所有的 core 都存取同一個 bank 中的同一個記憶體位置，這樣子會轉換成只讀取一次並將結果廣播給所有的 core。綜合以上，最佳存取方法為，一個 Warp 當中所有 core 存取的 Shared Memory 落在不同的 bank 當中，或是一個 Warp 當中所有 Core 所存取的記憶體位置皆相同。而在較新的 CUDA 版本，bank 的大小也可以透

過程式修改為 4 bytes 或 8 bytes，一次最多可以有 32 個 bank 同時運作，且所有的 bank 必須是連續的 [23][26]。

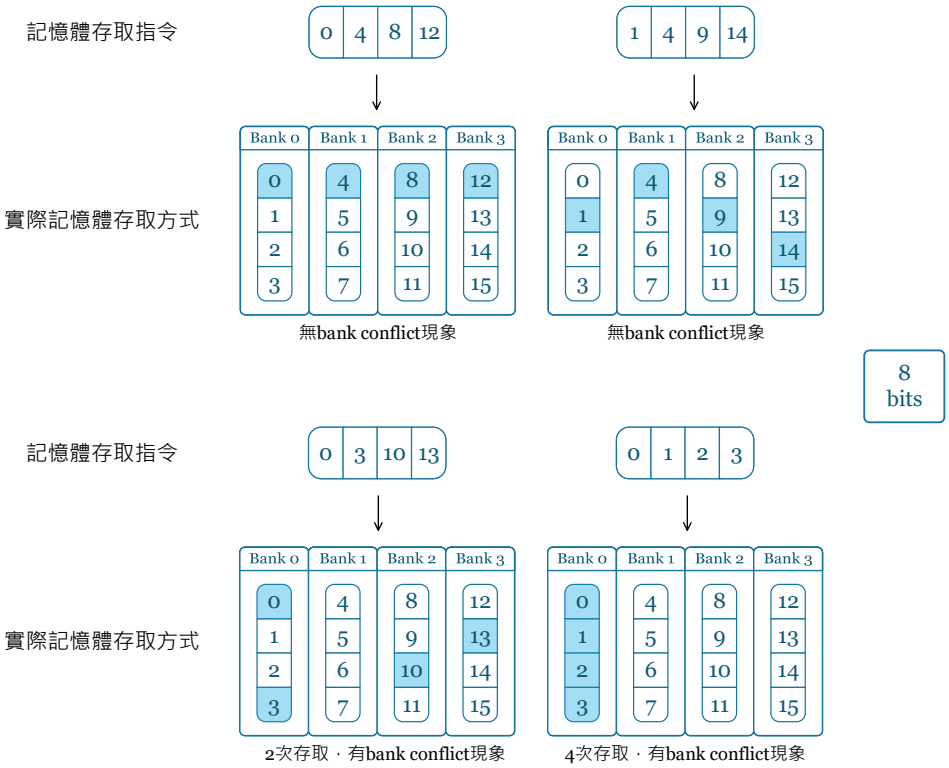



圖 2.16：Bank Conflict 示意圖

### 程式編寫之抽象化記憶體

在編寫 CUDA 程式時，於章節2.3.2有說明可依記憶體的共享程度區分為 Global Memory 與 Shared Memory。另有一分類方式為依記憶體實際所在區域而分為 Device Memory 與 Host Memory，Device Memory 即實際位於 GPU 上的記憶體，通常指其 Global Memory，而 Host Memory 則為 CPU 端的記憶體 [27]。

按以上設計區分即代表，GPU 的運算指令不可操作 Host Memory 上的資料；CPU 的運算亦不可存取 Device Memory。因此在編寫 GPU 程式上需有明確的記憶體複製操作指令說明如何將位於 Host Memory 上欲運算的輸入資料複製至 Device Memory 中的特定記憶體位址，複製資料完後才能喚醒 GPU 協助計算；在 GPU 運算完畢後，亦需經過此繁複的動作才能將運算結果複製回 Host Memory 中供 CPU 讀取。

由於上述的程式編寫方法過於複雜且無效率，CUDA 在 6.0 版本之後對於硬體計算能力版本（Compute Capability version）[7]3.0 以上之 GPU 提供統一記憶體架構（Unified Memory）之記憶體存取方法 [19]。圖2.17即為此記憶體存取方法之運作架構圖。此一記憶體架構提供開發者可宣告使用一 Unified Memory 區塊，程式運行至此宣告時，會在 Device Memory 與 Host Memory 各配置一段相同大小之記憶體空間，並命其為同一段虛擬記憶體位址，後續 CPU 與 GPU 皆可操作指令將資料寫入至或讀取自此段虛擬記

[7]NVIDIA 公司依此版本號碼區分硬體支援的功能，亦稱為 SM 版本（SM version)。

憶體位址，此時若指令運行於 GPU，則會存取上述宣告之 Device Memory；反之，若操作指令運行於 CPU，則會存取上述宣告之 Host Memory。

Unified Memory 使程式編寫者在開發 GPU 程式時可更關注於重要的商業邏輯，而非不斷小心的處理資料所在的記憶體空間，除此之外，由於 Unified Memory 會在 GPU 與 CPU 空閒時，將 Host Memory 與 Device Memory 中的資料同步，不需要等到整個 GPU 運算完畢才一次將所有計算結果傳回 CPU，如此資料搬移的時間將可與運算的時間重疊，從而隱藏資料搬移的時間延遲。

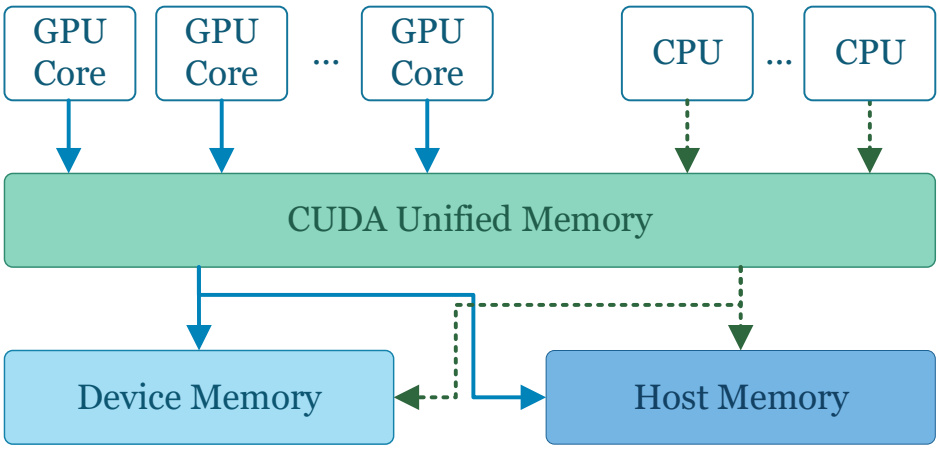



圖 2.17：CUDA Unified Memory 存取架構

# 第三章　研究架構設計

本章節將說明本論文提出的 RACH 優化架構與各種實作細節。由於提出的架構將借助使用 GPU 優化，故部分實作流程方法將與前章敘述有所差異。

圖3.1概略地呈現本論文實作架構與流程。在計算流程上，本論文將訊號疊合的計算改為 High-PHY 第一步驟，此修改順序會使後續 Correlation 與逆向傅立葉轉換計算的資料來源產生變化，然而因為這二項運算皆為線性的運算，故將不會造成後續的判讀出現錯誤，且可以大幅降低後續運算的資料量，進而提升整體運算效能。

另外，RACH 流程的目標是要找出是否有 UE 欲連上網與其所選擇之 signature 為何並提供 UE 同步的資訊，因此中間對於訊號處理的精準度要求並不高，故本論文在實作上配合 GPU 的硬體運作特性，所有浮點數運算皆使用單精度浮點數（占用 4 Bytes）[28] 計算。

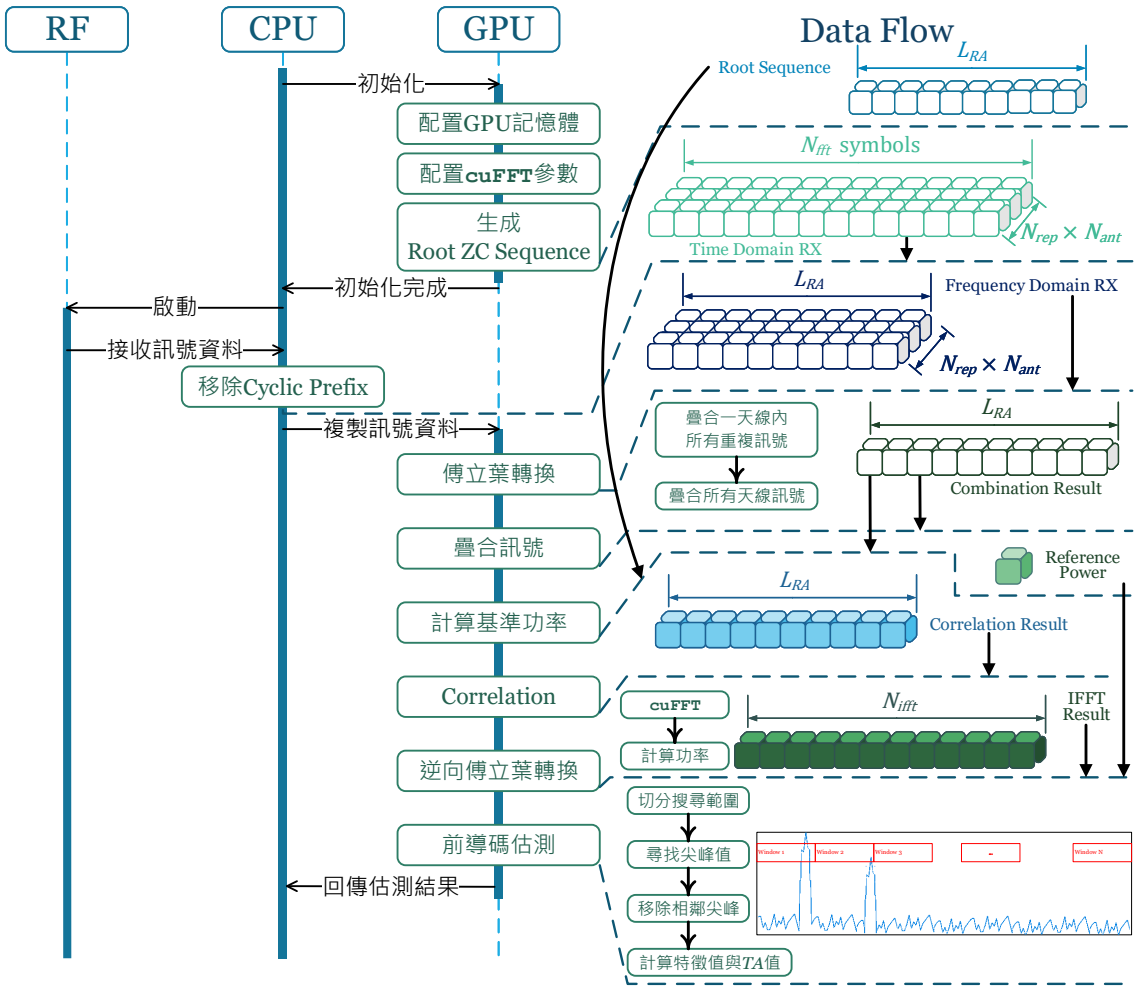



圖 3.1：RACH 在 CPU 與 GPU 上之協作流程架構

## 3.1 Root Zadoff-Chu 序列生成

如在章節2.2.3所述，軟體基地台在啟動後的初始化階段，即會先計算所有的 Root Sequence 以於後續訊號處理時可以直接比對。雖然在生成 Root Sequence 時，基地台仍處於初始化階段尚未開始接收處理訊號，將此步驟移至 GPU 運算對於整體軟體基地台在處理訊號上並不會有顯著效能的提升，然而在第一次啟動 CUDA 程式時 GPU 會先執行一系列初始化的動作後才執行實際運算，初始化行為包含選擇要使用的 GPU、記憶體初始化、載入 CUDA 程式碼等，這些工作都會在第一次啟動時才執行，一般將此一流程稱為暖機（Warm Up）[29]。故若將此 Root Sequence 的計算於 GPU 上執行，後續第一次訊號處理時即不會有多餘的 Warm Up 延遲時間。

### 3.1.1 GPU 資源分配

如在章節2.2.2所述，UE 傳送的 Preamble Sequence 共有 64 組可能，而這 64 組序列則透過多組 Root Sequence 分別使用 Cyclic Shift 取得。藉由式 2.2推算出一個基地台需要多少組的 Root Sequence 與每組 Root Sequence 可產生的 ZC Sequence 數量，表達為式 3.1。

$$N_{root} = \lceil 64/N_{C_v}(0) \rceil$$

$$N_{C_v}(i) = \left\{ \begin{array}{ll} \lfloor L_{RA}/N_{CS} \rfloor & , N_{CS} \neq 0, i < N_{root} - 1 \\ 64 - i \times N_{C_v}(i-1) & , N_{CS} \neq 0, i = N_{root} - 1 \\ 1 & , N_{CS} = 0 \end{array} \right\}, i = 0, 1, ..., N_{root} - 1$$

$N_{root}$：需要產生的 Root Sequence 數量。

$N_{C_v}(i)$：每組 Root Sequence 可產生的 ZC Sequence 數量。

(式 3.1)

此階段在 CUDA 程式設計上使用的資源如表3.1所示。如同章節2.3.2說明過，Block 與 Thread 皆可依需求安排為三維的結構，故表3.1在欲產生長度為 139 的 Sequence 所使用的 Block 數量表示為 ($N_{root}$, 139, 1)，即代表在三個維度上個別所需的 Block 數量，故共需使用 $N_{root} \times 139 \times 1$ 個 Block，在 Thread 的使用數量表示方法亦同。另外，Shared Memory 的欄位則代表一個 Block 內所有 Thread 共用的記憶體大小。

| $L_{RA}$ | Block 使用數量 | Thread 使用數量 | Shared Memory (Bytes) |
|---|---|---|---|
| 139 | ($N_{root}$, 139, 1) | (256, 1, 1 ) | $4 \times 2 \times 256$[1] |
| 839 | ($N_{root}$, 839, 1) | (1024, 1, 1 ) | $4 \times 2 \times 1024$ |

表 3.1：Root ZC Sequence 生成使用 GPU 之資源

### 3.1.2 GPU 累加演算法

因為在生成序列的最後步驟有累加運算，故在此先介紹本論文如何在 GPU 上有效率地計算累加 [30]，此方法亦可運用於後續於於一輸入序列中得一輸出結果的類似運算

[1] 長度為 256 的複數陣列，一複數由實數與虛數組成，個別需要 4 Bytes。

（Reduction Operation）

一般在 CPU 上累加一長度為 $n$ 的數列，通常會使用一迴圈計算，其時間複雜度為 $O(n)$。然若在 GPU 上使用相同方法將會非常的沒有效率，且無法善用 GPU 的多核心計算優勢。

若需累加長度為 $n$ 的序列，則啟動 $K$ 個 Thread，$K$ 與 $n$ 的關係表示如式 3.2。演算法流程呈現於圖3.2與演算法3.1。首先，每個 Thread 先將其所負責的資料從 Global Memory 載入至 Shared Memory 中，而編號 $\geqq n$ 的 Thread 則將 0 放入 Shared Memory。接下來執行一迴圈，迴圈每執行一次，實際參與運算的 Thread 就減半，而該輪負責運算的 Thread 即自 Shared Memory 中取二數相加，再將結果存回 Shared Memory。藉此，總時間複雜度將降為 $O(\lg n)$，因此當欲累加的序列越長，使用 GPU 執行運算的所節省的時間就越明顯。

$$k = \lceil \lg n \rceil$$
$$K = 2^k \qquad \text{(式 3.2)}$$

此算法的流程會需要大量的 Thread 同步（Threads Synchronization）與 Shared Memory 的使用，故在存取資料與 Thread 使用需特別注意。若每輪迴圈實際參與運算的 Thread 不相鄰，會造成在章節2.3.3所提到的 Warp Divergence 問題；若一 Thread 選取欲相加的二數在 Shared Memory 是緊鄰的也可能會有 Bank Conflict 問題。另外，當實際運算的 Thread 少於 32 時，剩下的 Thread 在硬體上即會被規劃在同一 Warp 運算，故當執行至此，可不再做同步的動作，進而加速整體計算的效率。

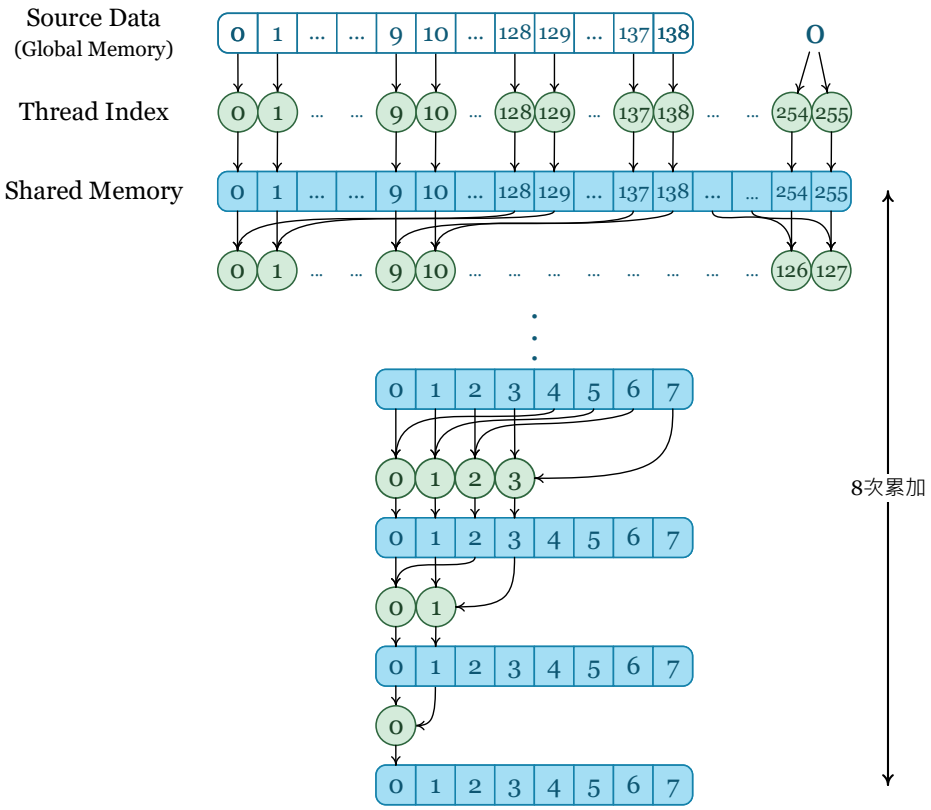



圖 3.2：GPU 累加演算法

**演算法 3.1:** GPU 累加演算法

**輸入：** 長度為 $n$ 的 Sequence $Seq$
長度為 $K$ 的 Shared Memory $S$

**輸出：** $Seq$ 的總和

$T_i \leftarrow$ Thread Index
$offset \leftarrow K/2$

**if** $T_i < n$ **then**
  $S[T_i] \leftarrow Seq[T_i]$
**else**
  $S[T_i] \leftarrow 0$
**end**

**while** *offset > 0* **do**
  **if** $i < offset$ **then**
    $S[T_i] \leftarrow S[T_i] + S[T_i + offset]$
  **end**
  $offset \leftarrow offset/2$
  Threads Synchronization
**end**

**return** $S[0]$

### 3.1.3 序列生成

在生成序列時，一個 Block 底下的所有 Thread 會合作計算出最後輸出頻域序列中的一個數值，即一個 Block 會計算出式 2.3當中 $y_{u,v}$ 序列中的一數；一個 Thread 負責計算一組 $x_{u,v}(m) \cdot e^{-j\frac{2\pi mn}{L_{RA}}}$，最後再按照前小節所述的演算法執行累加計算。

觀察表3.1中的 Block 的使用量，以及章節2.3.4提到的軟硬體架構對應，可以發現在軟體上宣告使用的 Block 數量其實是遠大於實際硬體的 SM 數量，故可推得實際程式運行時，SM 應會大量的切換實際執行的 Block。而一個 Root Sequence 會由 139 或 839 個 Block 計算，這些 Block 在前半部分計算 $x_{u,v}(m) \cdot e^{-j\frac{2\pi mn}{L_{RA}}}$ 時的運算是重複的。如此乍看之下，此為一相當沒有效率且耗用資源的規劃，然而在此設計之下，每個 Thread 最多只計算一次的複數乘法與 $\lceil \lg 1024 \rceil = 10$ 次的複數加法，計算含量相當低，所以此步驟最大的時間瓶頸應為累加所需要的 Thread 同步動作；而又因為啟動夠多的 Block 與 Thread，故當有 Thread 需因同步而暫時無法計算時，該 Core 可切換去執行其他 Thread 的運算，且因每個 Block 最多只需要 8192 Bytes 的 Shared Memory，故一個 SM 中將可容納大量交互切換計算的 Block，進而可隱藏因 Thread 同步所造成的延遲。

## 3.2 Cyclic Prefix 移除

由此開始，即為基地台在收到 Preamble 後要做的一系列序號處理之流程步驟。Preamble 的組成如圖2.5所示，因此此步即為移除前段的 Cyclic Prefix，保留中間段的 Sequence 資料。在軟體程式實作上，即為清除儲存 Cyclic Prefix 資料的記憶體區段，而本論文的目標為將 RACH 的處理流程使用 GPU 改善實作，故此步進而簡化成：只複製需要計算的中間段資料進入 GPU 執行後續計算。以此方式實作將有以下優點：

- 在 GPU 上若存取不連續或非對齊的資料時，將無法享有於章節2.3.5提到的合併記憶體存取所帶來的好處，且會降低資料存取之效率。而若只複製需要計算資料即可確保後續存取資料時是連續且對齊的，進而降低後續資料存取成本。
- CPU 與 GPU 間的資料傳輸是依靠 PCIe 完成，其傳輸速度為本論文架構中最慢的資料途徑，故減少 CPU 與 GPU 間的資料傳輸大小與次數亦為一優化重點。

## 3.3 快速傅立葉轉換

將傳輸至 GPU 的訊號使用快速傅立葉轉換（Fast Fourier Transform, FFT），使訊號自 time domain 轉換至 frequency domain。在實作上，本論文借助 CUDA toolkit 內提供的 cuFFT 程式庫完成此運算。

根據 CUDA 官方文件 [31] 所述，cuFFT 內所提供的運算可表達為式 3.3，此運算式與式 2.4相同，故可直接使用不需修改。另外，cuFFT 提供的方法可一次同時計算多條一樣大小的序列，且會根據不同的輸入序列大小，配置不同的硬體資源運算，而此配置可預先設定並保存，每次執行時僅需輸入資料，不必執行多次的初始化配置。最後，cuFFT 提供的演算法可將計算時間複雜度降至 $O(n \lg n)$，故綜合考量以上優點，本論文於此流程使用 cuFFT 程式庫完成實作。

$$X(k) = \sum_{n=0}^{N_{fft}-1} x(n) \cdot e^{-2\pi i \frac{kn}{N_{fft}}}, k = 0, 1, ..., N_{fft} - 1 \quad \text{(式 3.3)}$$
$$X, x \in \{\mathbb{C}\}$$

## 3.4 訊號疊合

由此開始為 High-PHY 的運算，雖然本論文之實作設計包含 Low-PHY 之運算，亦即章節2.1.2中提到的使用 Option 8 切分 RU 與 DU 之設計，但因本論文於實作設計上考慮未來擴充彈性，因此亦支援使用 Option 7 的切分方法，故在此引進一新變數 $N_{RU}$，此變數即代表可同時支援的 RU 數量。另外，自此亦會開始加入 $N_{OCC}$ 變數的討論，此變數即為在章節2.2.2提到於頻域上可容納的 PRACH 介面數量。

### 3.4.1 計算方法

如同章節一開始所述，為降低後續運算的資料量，所以在新的流程設計上先做訊號疊合的動作；將多天線與重複的 Sequence 訊號疊加計算平均的動作。也因為後續的 Correlation 與逆向快速傅立葉轉換都是線性運算，故先做訊號疊合的計算並不會影響到後續的運算。運算表示為式 3.4。

$$c_{R,O}(k) = \frac{1}{N_{ant}} \sum_{a=0}^{N_{ant}-1} \left( \frac{1}{N_{rep}} \sum_{r=0}^{N_{rep}-1} X_{R,O}(a,r,k) \right) \quad \text{(式 3.4)}$$
$$R = 0,1,...N_{RU}-1 \quad O = 0,1,...N_{OCC}-1$$
$$k = 0,1,...L_{RA}-1 \qquad c, X \in \{\mathbb{C}\}$$

### 3.4.2 GPU 資源分配

在此所使用的 GPU 資源呈現於表3.2。一個 Block 負責計算出所有的重複序列疊合，一個 Thread 負責計算並輸出一個序列中的一數的結果。由於需要相加的量並不多，因此若使用章節3.1.2的方法計算，計算次數之差將不足以隱藏大量的 Thread 同步時間，故不採用該演算法，而是直接計算加總。

| $L_{RA}$ | Block 使用數量 | Thread 使用數量 | Shared Memory (Bytes) |
|---|---|---|---|
| 139 | ($N_{RU}$, $N_{OCC}$, 1) | (139, 1, 1 ) | 0 |
| 839 | ($N_{RU}$, $N_{OCC}$, 1) | (839, 1, 1 ) | 0 |

表 3.2：訊號疊合使用之 GPU 資源

圖3.3為一訊號疊合之計算於 GPU 上的資源分配示意圖。因為實際輸入的資料來源組成相當多，分別有來自不同 RU 的訊號、一 RU 內多個天線的輸入訊號、一天線內一 PRACH Slot 內不同頻域上的多個 Preamble、以及一個 Sequence 內多次重複的訊號，以上為一次運算的所有資料來源，這些輸入資料標示於圖的最下方區塊中，在此區塊中，沿向上垂直軸所相疊的方格即為乘載相同資料之訊號資料（一方格代表一複數，占用 8 Bytes），所以此軸上所有的訊號將被疊合，因此在疊合後，原本四維的輸入資料，輸出時將僅餘三維的資料，而輸出的資料擺放規劃則標示於圖的上方區塊中。在圖的中間區塊則為 GPU 的資源規劃，此規劃是依最後輸出的資料量分配，一 Thread 將會計算出一格輸出資料，因此一 Thread 最多需計算 $N_{ant} \times N_{rep}$ 次的複數加法，計算量亦像當低。另外依此設計，相鄰 Thread 自 Global Memory 提取與儲存的資料位置皆相鄰，因此對照至實際的 DRAM 存取位置也皆相鄰且對齊，故此規劃方法的記憶體存取成本相當低，一 Thread 在運算過程中所占用的時間大部分都在做實際加法運算。

## 3.5 基準功率計算

在原始 CPU 版本程式中，對於訊號噪音（Noise）的計算相當繁複，該方法參考了訊號機率分布，以及各種 PRACH 設置場域等多種參數進行計算。在本篇論文中，採用

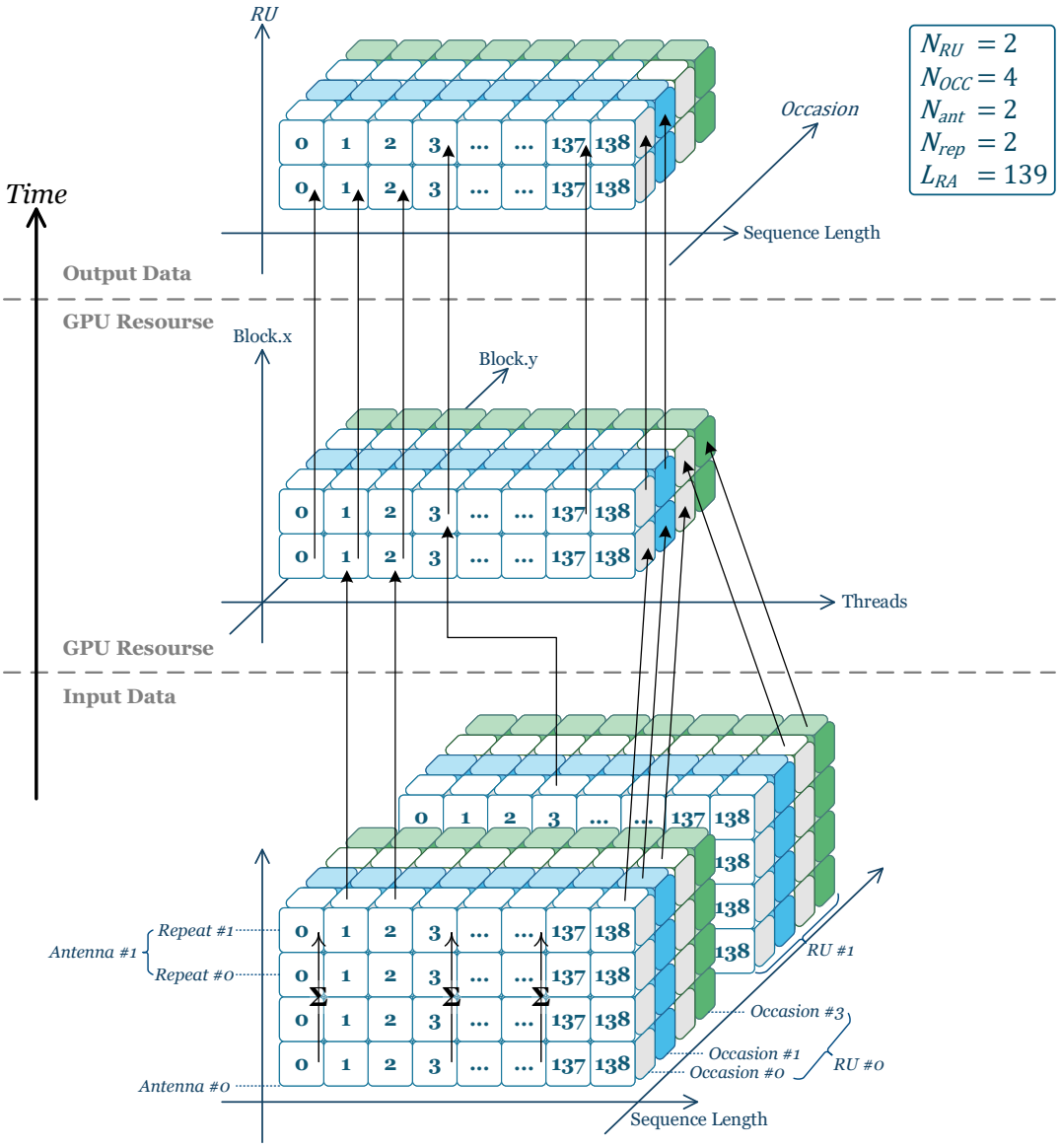



圖 3.3：計算訊號疊合之 GPU 資源分配

運算較為簡單的基準功率（Reference Power）作為後續估測時的 Noise 參考。

### 3.5.1 計算方法

Reference Power 記為 $P_{ref}$。計算方法如式 3.5所示，即計算在上一步驟中疊合後的訊號平均功率。一訊號點之功率計算為該複數點於複數平面上與原點之距離的平方，因此在實際計算上可以省略在計算距離時會有的較耗時間之開根號運算。

$$P_{ref_{R,O}} = \frac{1}{L_{RA}} \sum_{k=0}^{L_{RA}-1} |c_{R,O}(k)|^2$$
$$R = 0, 1, \ldots N_{RU} - 1 \quad O = 0, 1, \ldots N_{OCC} - 1$$
$$c \in \{\mathbb{C}\} \qquad P_{ref} \in \{\mathbb{R}\} \qquad \text{(式 3.5)}$$

### 3.5.2 GPU 資源分配

在計算 Reference Power 時所分配的 GPU 資源呈現於表3.3。式 3.5顯示此運算含有大量的累加運算，故在實作上再次使用章節3.1.2所介紹的 GPU 累加演算法。稍有不同處是，此次運算除一開始的功率計算外，其餘計算皆是純粹的浮點加法運算，無須其他複雜的資料前處理動作，故可以分配部分 Thread 在進入第一輪迴圈前先做一次加法，因此 Thread 的啟動數量 $K$ 與 $L_{RA}$ 的關係修改表示為式 3.6；如此不僅可以少一次的 Thread 同步時間，亦可降低 Thread 的使用數量，不須啟用只填入 0 的 Thread。

$$\begin{aligned} k &= \lfloor \lg(L_{RA}) \rfloor \\ K &= 2^k \end{aligned} \qquad \text{(式 3.6)}$$

| $L_{RA}$ | Block 使用數量 | Thread 使用數量 | Shared Memory (Bytes) |
|---|---|---|---|
| 139 | ($N_{RU}$, $N_{OCC}$, 1) | (128, 1, 1 ) | $4 \times 2 \times 128^2$ |
| 839 | ($N_{RU}$, $N_{OCC}$, 1) | (512, 1, 1 ) | $4 \times 2 \times 512$ |

表 3.3：基準功率計算使用之 GPU 資源

圖3.4即為一計算 Reference Power 時之 GPU 資源規劃示意圖。因為代表相同的訊號資料已於上一步驟中做疊合，故在圖中的下方資料輸入區塊中僅能看到疊合後的資料。另外，圖的上方區塊為 GPU 的資源規劃，顯示出在剛啟動載入資料時，前 11 個 Thread 不僅會自 Global Memory 中提取相對應的資料點計算其功率，亦會提取最後 11 點的資料並計算功率後與之相加，而其餘 Thread 僅將其對應的資料點在計算完功率後，放入 Shared Memeory 中。至此才會啟動在章節3.1.2所提到的 GPU 累加演算法；同理，若為 $L_{RA} = 839$ 的情況，則會安排前 $839 - 512 = 327$ 個 Thread 先計算加法後，才會啟動 GPU 累加演算法。最後在累加計算結束後，僅需每個 Block 中的第一個 Thread 將累加結果除以 $L_{RA}$ 得 Reference Power 後輸出。

按此資源規劃以及資料擺放，一開始 Thread 自 Global Memory 中取得輸入資料時，因一 Block 內的 Thread 所請求之資料位置皆相鄰，所以當一個 Warp 內的 32 個 Core 一起向 DRAM 要求資料時，會一次請求連續的 256 Bytes[3] 資料，按章節2.3.5所述，此連續且對齊的 256 Bytes 資料存取，可以合併成僅二次的資料傳輸。另外，在後續累加演算法中使用 Shared Memory 時，因運算中所產出的數皆使用單精度浮點數儲存，故一數於 Shared Memory 中皆剛好占用一個 bank，且一 Warp 內所有的 Core 所使用的 Shared Memory 皆連續，故在一 Warp 內的一輪計算只需二次的 Shared Memory 提取（被加數與加數）與一次的儲存即可。綜合以上，在記憶體操作方面已配合硬體上的記憶體存取限制將可能的存取次數降至最低，故不會有多餘無效的記憶體存取延遲；而在實際計算量方面，每一 Thread 最多僅需計算二次浮點乘法與十次浮點加法[4]，因此個別 Thread 的運算量亦相當低。

[2] 長度為 128 的複數陣列，一複數由實數與虛數組成，個別需要 4 Bytes。
[3] 長度為 32 的連續複數資料，一複數由實數與虛數組成，個別占用 4 Bytes。
[4] 需計算功率（二次浮點乘法、一次浮點加法）與 $\lfloor \lg 839 \rfloor = 9$ 次累加。

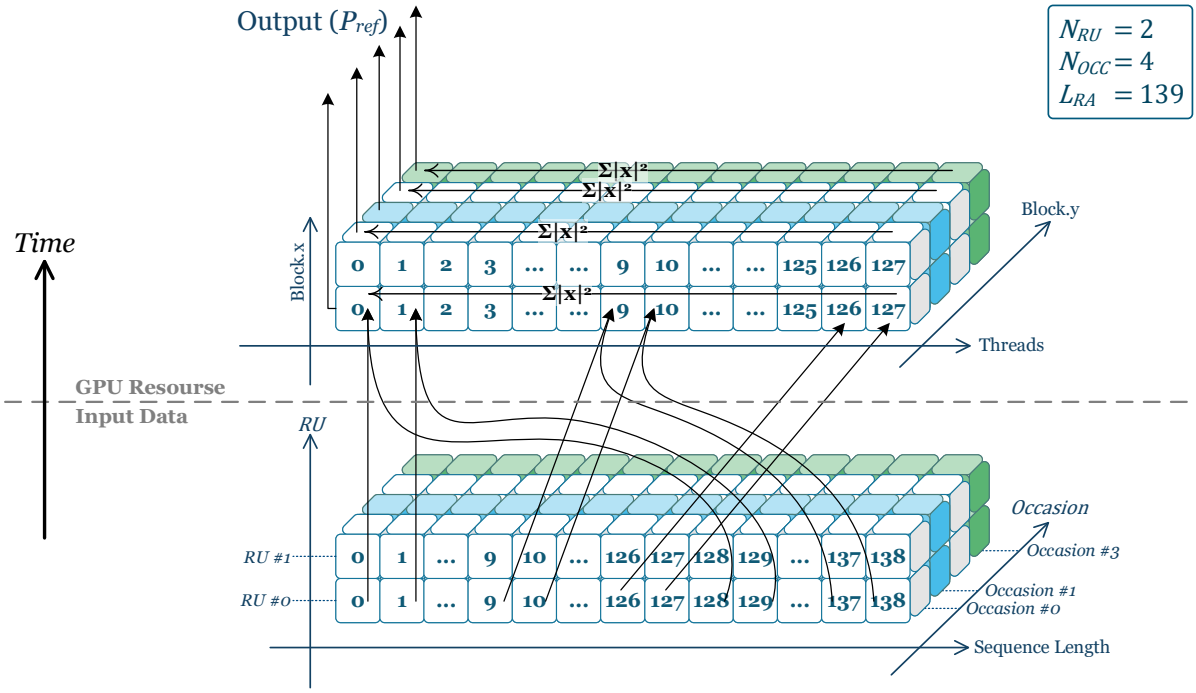



圖 3.4：基準功率計算之 GPU 資源分配

## 3.6 Correlation

在此將取用於章節3.1基地台初始化時輸出的 Root Sequence 與章節3.4所計算之疊合訊號計算。與 Root Sequence 計算 Correlation 用意即在確認此輸入訊號是否為該 Root Sequence 透過 Cyclic Shift 而得之一可能的 Preamble Sequence，故在此需要將疊合後的訊號與所有的 Root Sequence 做 Correlation 比對。

### 3.6.1 計算方法

將所有經疊合之訊號與 Root Sequence 計算 Correlation 的方式表示於式 3.7，其中 $\overline{c_{R,O}(k)}$ 為 $c_{R,O}(k)$ 之共軛複數。一組疊合後的訊號需與所有 Root Sequence 計算 Correlation，故在運算後 1RU 將產出 $N_{OCC} \times N_{root}$ 組結果，後續再由這些序列估測是否有 UE 發送 Preamble。

$$C_{R,O,i}(k) = \overline{c_{R,O}(k)} \times y_i(k)$$
$$R = 0, 1, 2, ..., N_{RU} - 1 \quad O = 0, 1, 2, ..., N_{OCC} - 1$$
$$k = 0, 1, 2, ..., L_{RA} - 1 \quad i = 0, 1, 2, ..., N_{root} - 1 \qquad \text{(式 3.7)}$$
$$C, c, y \in \{\mathbb{C}\}$$

### 3.6.2 GPU 資源分配

在計算 Correlation 所使用之 GPU 資源如表3.4所示。所有接收到的訊號皆須與 Root Sequence 交叉計算，所以每個 RU 啟用需啟用 $N_{OCC} \times N_{root}$ 個 Block 以個別計算一組輸出。而一 Thread 負責一 Correlation 運算。

| $L_{RA}$ | Block 使用數量 | Thread 使用數量 | Shared Memory (Bytes) |
|---|---|---|---|
| 139 | $(N_{RU}, N_{OCC} \times N_{root}, 1)$ | (139, 1, 1 ) | 0 |
| 839 | $(N_{RU}, N_{OCC} \times N_{root}, 1)$ | (839, 1, 1 ) | 0 |

表 3.4：Correlation 計算使用之 GPU 資源

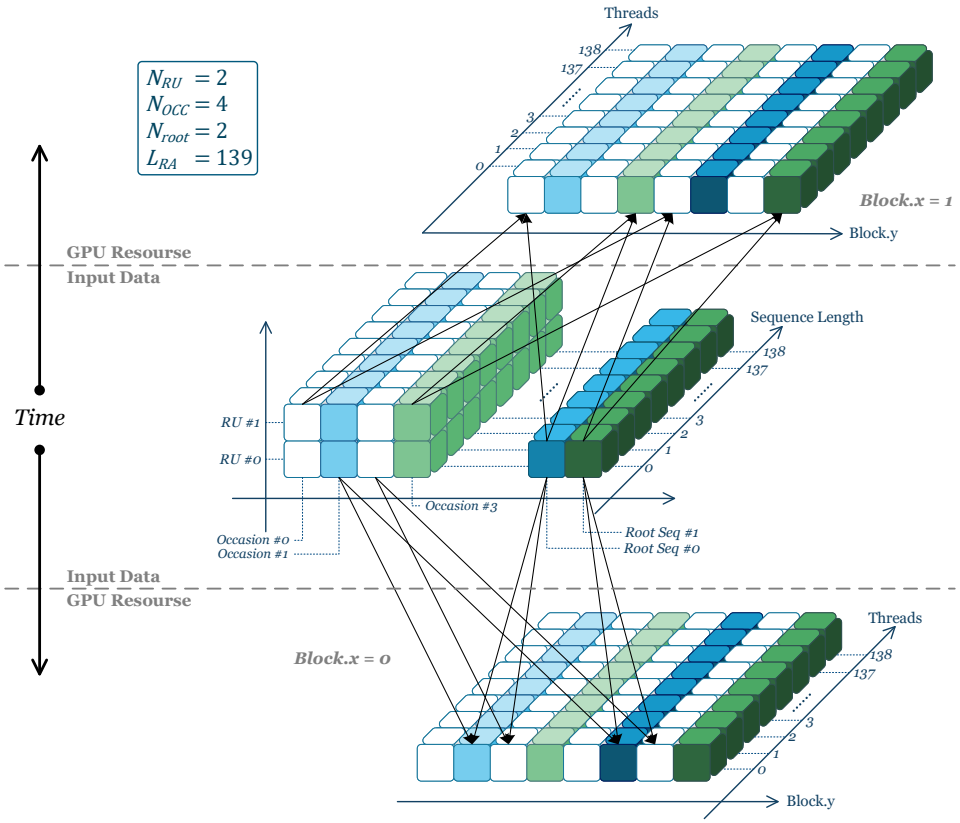



圖 3.5：Correlation 使用之 GPU 資源

圖3.5即為一例 Correlation 於 GPU 運算之資源分配與資料擺放示意圖。GPU 運算資源呈現於圖的上下二區塊中，上下區塊代表不同 RU 所執行運算之資源，一 RU 內的各 Occasion 將使用 $N_{root}$ 個 Block，其內個別含有 $L_{RA}$ 個 Thread，一 Block 會自二 Global Memory 區塊分別取得疊合後的訊號資料與事先計算之 Root Sequence。此二區塊的資料皆於之前運算完成時依序擺放並對齊，故於此運算需要這些資料時不會有非連續的存取情況發生，也因為資料擺放是對齊的，所以在此二大不連續區塊中提取資料時，也可以因為有效地利用 L2 Cache，而不會有過多的 DRAM 存取的狀況發生。在輸出資料上，也會將所有 Correlation 計算結果依序輸出至 Global Memroy 中，所以無論是輸入或輸出的提取或擺放皆是很有效率的。最後，一 Thread 在取得資料後，僅需一次的複數乘法即可完成，運算量是整個流程中最低的。

## 3.7 逆向快速傅立葉轉換

此逆向快速傅立葉轉換（Inverse Fast Fourier Transform, IFFT）依然借用在章節3.3所介紹的 cuFFT 函式庫輔助完成。因 cuFFT 對於輸入長度為 2 的冪次方的序列會有最佳的效能表現 [31]，故將序列的長度透過補 0 的方式將 $L_{RA}$ 為 139 的序列展開到長度為 1024 的序列；$L_{RA}$ 為 839 的序列展開到長度為 2048 的序列，此展開後的長度記為 $N_{ifft}$，故總運算量為 $N_{RU} \times N_{OCC} \times N_{ifft}$ 點的 IFFT 計算。

另外，後續的運算僅需使用 Correlation 結果於 time domain 上的 power 判斷即可，故可將 Correlation 的結果轉換至 time domain 上後，直接計算 cuFFT 輸出結果的 power 值。因此，此步驟的計算可表示如式 3.8。cuFFT 函式庫也有提供方法讓開發者設定使 cuFFT 實際運算前後增加程式碼以針對輸入資料或輸出結果處理，開發者不需再另外啟動其他 GPU 資源，進而有多餘的啟動與資料存取時間。

$$
\begin{gathered}
I_{R,O,i} = iFFT(C_{R,O,i}) \\
P_{R,O,i}(n) = |I_{R,O,i}(n)|^2 \\
R = 0,1,2,...,N_{RU}-1 \quad O = 0,1,2,...,N_{OCC}-1 \\
i = 0,1,2,...,N_{root}-1 \quad n = 0,1,2,...,N_{ifft}-1 \\
C, I \in \{\mathbb{C}\} \qquad\qquad P \in \{\mathbb{R}\}
\end{gathered}
\tag{式 3.8}
$$

## 3.8 前導碼估測

在所有資料前處理後，最後流程即判斷訊號中是否有 UE 傳送 Preamble 及判定該 UE 所選擇之 signature 為何。本論文在 GPU 實作此估測流程如圖3.6所示，先透過將 Correlation 的結果切分不同為 Window，並根據章節2.2.2說明，每一 Window 可對應至不同的 $C_v$，故可在 Window 中尋找符合條件的尖峰值以判定是否有 UE 傳送該 Window 所對應之 $C_v$。而若在相鄰的二 Window 內皆有找到符合條件之尖峰值，且其位置過於相近，則以假設為一延遲過久之訊號為由僅留下最大值，最後得出 signature 與其延遲時間的 $TA$ 值。

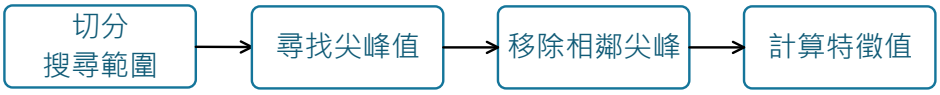



圖 3.6：前導碼估測於 GPU 實作流程圖

### 3.8.1 GPU 資源分配

此估測流程所使用的 GPU 資源呈現於表3.5。因 Long Preamble 與 Short Preamble 的優化架構不同，故在 Long Preamble 的部分甚至使用 Stream 切換大量的工作，另外 Thread 與 Block 的使用會在不同的估測階段而有不同代表的計算角色，所以詳細資源分配設計概念將於後續各小節依序說明。

| $L_{RA}$ | Stream 使用數量 | Block 使用數量 | Thread 使用數量 | Shared Memory (Bytes) |
|---|---|---|---|---|
| 139 | 1 | ($N_{RU}$, $N_{OCC}$, 1) | (max($N_{C_v}(i)$), $N_{root}$, 1 ) | $(4+2+2)\times 64$ |
| 839 | $N_{RU} \times N_{OCC}$ | (64, 1, 1) | 式 3.9 | $(4+2+2)\times 1024$ |

表 3.5：估測 Preamble 使用之 GPU 資源

$$\text{Short Preamble Thread 使用數量} = \left( \left\{ \begin{array}{ll} 1024 & ,\max(L_w) > 1024 \\ 64 & ,\max(L_w) < 64 \\ 2^{\lceil \lg(\max(L_w)) \rceil} & ,\text{otherwise} \end{array} \right\}, 1, 1 \right) \quad \text{(式 3.9)}$$

### 3.8.2 搜尋範圍切分

搜尋範圍切分可再分成二細項步驟。第一步由 CPU 計算於章節2.2.3中介紹的 Window 長度（記為 $L_w$）與章節3.1.1中提到的 $N_{root}$ 與 $N_{C_v}(i)$ 值，這部分可在系統剛啟動初始化時計算，後續只需按照此結果啟動對應 GPU 資源。第二步即在 GPU 內存取該 Thread 對應的 Window 資料段，每個 Thread 分配 Window 資料段的方式則會依 Long Preamble 與 Short Preamble 不同。

**Short Preamble**

一個 Block 將會估測在一次 PRACH Occasion 內有多少 UE 發送 Preamble，以及其對應的 signature 與 $TA$ 值；一個 Thread 負責一個 Window 內的計算，而一次 Occasion 內的資料又須與所有的 Root Sequence 比對，故一個 Block 內最多需要 $\max(N_{C_v}(i)) \times N_{root}$ 個 Thread[5]。

Short Preamble 切分 Window 與 GPU 資源分配方式可表示如圖3.7，圖中的上方為一個 Occasion 內需啟動的 Thread 總數，由於程式編寫上同一維度啟動的 Thread 數量需相同，故實際需啟動 66 個 Thread，但是最後二個 Thread 會在啟動後即停止。一 Thread 即代表一 signature，故實際運行的 Thread 數皆為 64 個，

**Long Preamble**

Long Preamble 切分 Window 與 GPU 資源分配方式可表示如圖3.8，圖中的上方為一個 Stream 內需啟動的 Block 與 Thread 數；在 Long Preamble 的情況下，一個 Grid 將會估測在一次 PRACH Occasion 內有多少 UE 發送 Preamble，以及其對應的 signature 與 $TA$ 值。在 GPU 剛啟動時，一個 Block 代表一個 Window，其所包含的 Thread 互相合作找出該 Window 內是否有符合條件的尖峰值，且一個 Window 即代表一個對應的 signature，故需要 64 個 Block。最後，因為已經將 Grid 分配去對一個 Occasion 內的資料做估測，所以需要借用 Stream 同時計算所有的 RU 與 Occasion 的估測，故需要 $N_{RU} \times N_{OCC}$ 個 Stream。

[5] $N_{C_v}(i)$ 與 $N_{root}$ 的計算可參考章節3.1.1介紹。

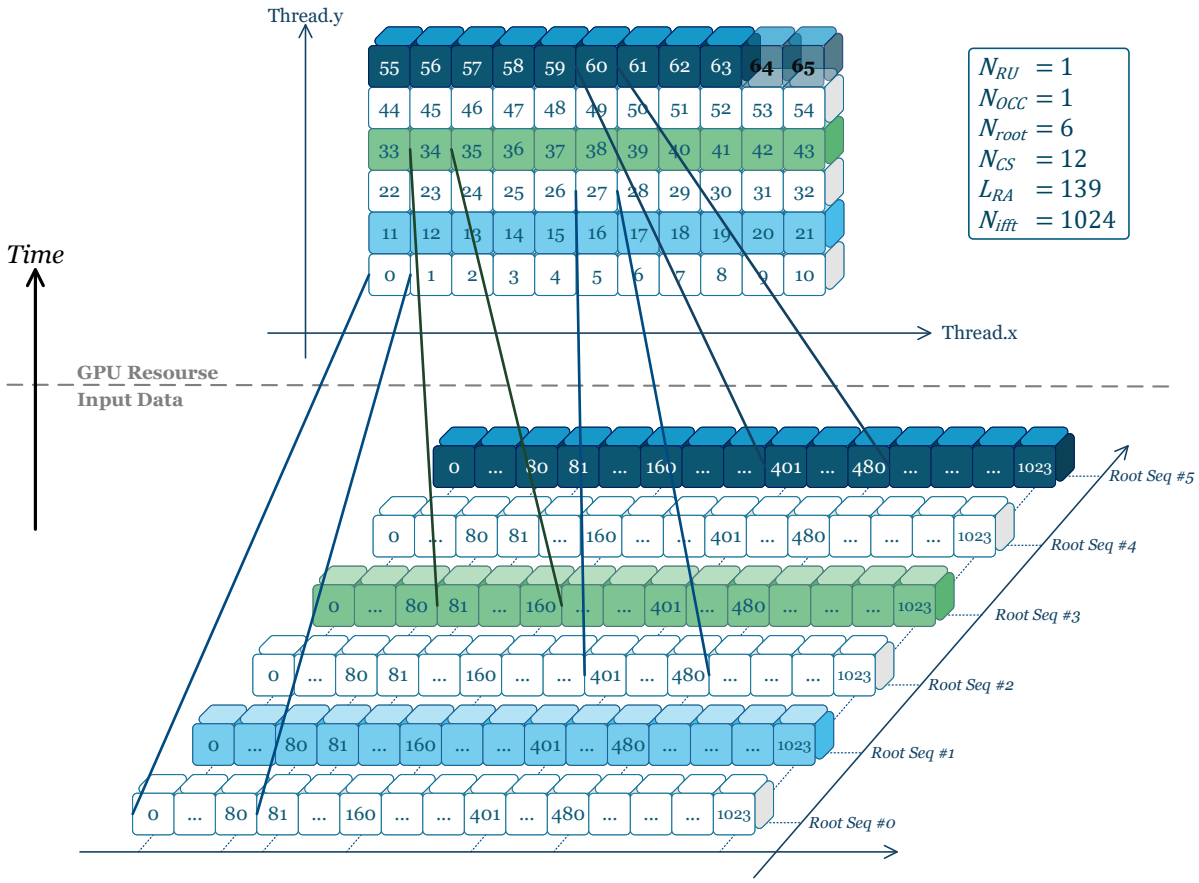



圖 3.7：GPU 存取 Window 範圍資料之使用資源（Short Preamble）

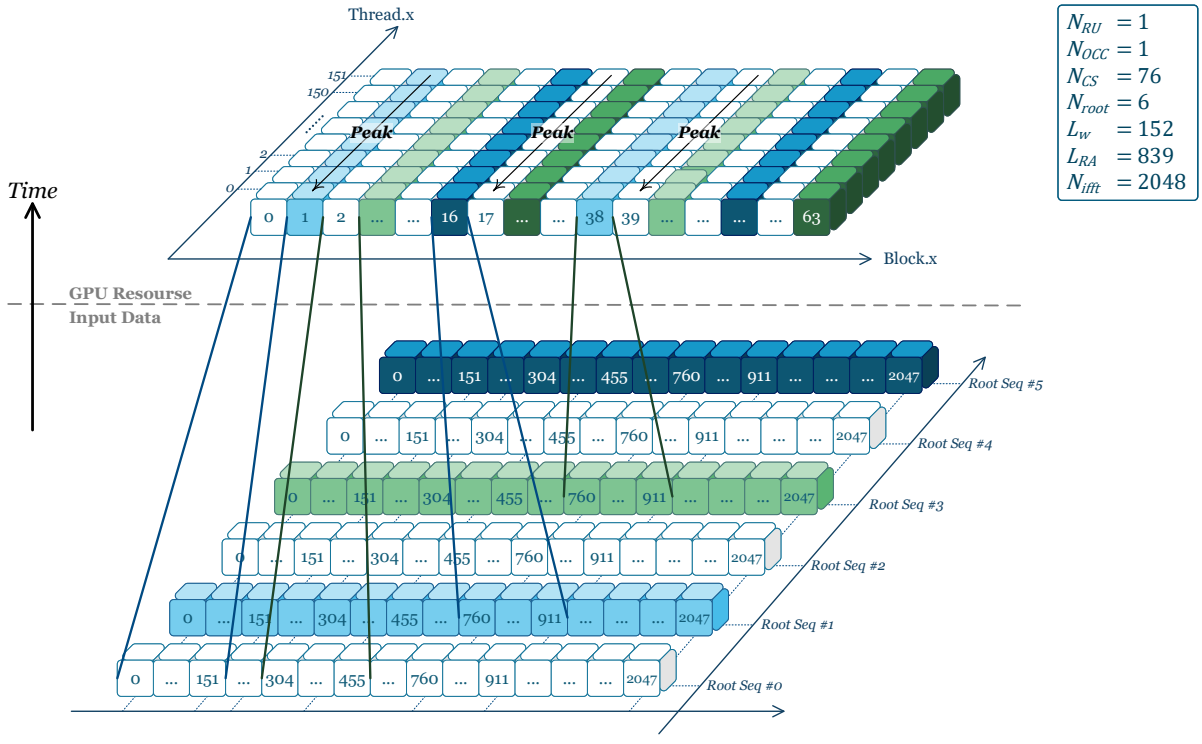



圖 3.8：GPU 存取 Window 範圍資料之使用資源（Long Preamble）

### 3.8.3 尋找尖峰值

在此階段，Long Preamble 與 Short Preamble 的做法依然不同。但是其目的皆是在一 Window 內找出符合條件之尖峰值。成為尖峰值的條件有二：功率值為 Window 內最大值、其值需大於臨界值（Threshold）。臨界值亦稱為噪音基準，即此數值以下之訊號皆視為噪音，不具意義。在本論文中，預設以章節3.5所計算之基準功率的 4 倍設定為臨界值，基地台可視不同運行場域設定為不同倍數。以圖3.9為例，前三個 Window 中各有一符合條件之尖峰值。

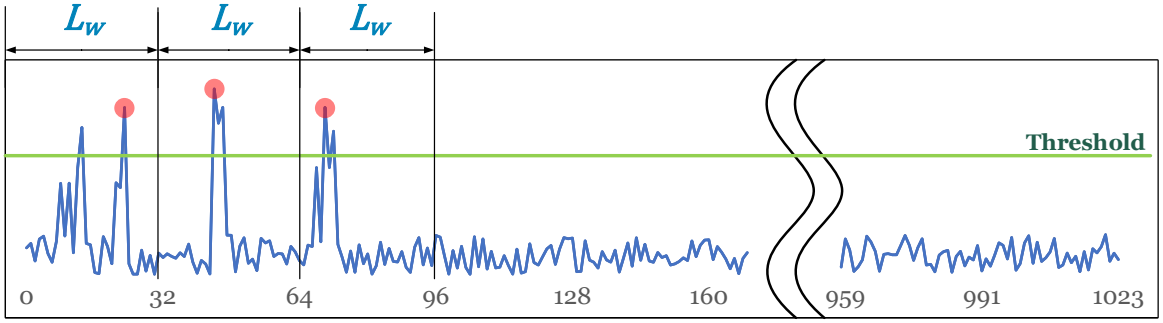



圖 3.9：尋找尖峰值結果

**Short Preamble**

如同前一小節所述，在 Short Preamble 的情境中一個 Thread 將負責找出一 Window 內的尖峰值，由於 Short Preamble 的平均 Window 長度並不長，所以就簡單的使用迴圈完成此一比較。而因後續有移除相鄰尖峰的動作，故 Thread 之間亦須知道彼此 Window 內的尖峰值，所以各 Thread 會將判斷結果存於 Shared Memory，判斷結果的資料包含：功率尖峰值（4 Bytes）、該尖峰所對應之 $C_v$（2 Bytes）、該尖峰延遲 $TA$ 時間（2 Bytes），又最大 $C_v$ 值為 64，故一 Block 內需使用 $8 \times 64 = 512$ Bytes 的 Shared Memory。

**Long Preamble**

如同前一小節所述，在 Long Preamble 的情境中是使用一個 Block 找出一 Window 內的尖峰值，一個 Block 內的 Thread 會合作找出 Window 內符合條件的尖峰值。在這裡所使用的方法與前面章節3.4與章節3.5做序列總和的方式相同；在前面章節計算加總時，每輪迴圈一 Thread 會取二數相加將結果放入 Shared Memory 中，在此則改為每輪迴圈一 Thread 取二數相比，將符合條件且功率較大者之尖峰值放入 Shared Memory 中。由於 Long Preamble 的平均 Window 較長，採此策略將能大幅降低執行時間，且因為目前規格中 Long Preamble 在一時間內在頻域上皆只有一次 PRACH Occasion，故實際使用到的 Stream 數量只會有 $N_{RU}$ 個，並不會有大量的 Stream 工作切換而造成計算延遲。

在部分情境下，Window 長度可能會超過 1024，因此需要特別限制 Thread 數量在 1024 內以符合 CUDA 的使用限制，於此情境的作法會與章節3.5的做法相同，先將超出範圍的資料在剛啟動時比較，並將結果存入 Shared Memory 後再啟動演算法。另外，使

用此方法需要 Thread 彼此合作分享資料，故需要使用 Shared Memory。一 Thread 在完成比較後，需要紀錄的資料包含功率尖峰值（4 Bytes）、該尖峰所對應之 $C_v$（2 Bytes）、該尖峰延遲 $TA$ 時間（2 Bytes），又啟動的最大 Thread 數量為 1024，故一 Block 內最多需使用 $8 \times 1024 = 8192$ Bytes 的 Shared Memory。

### 3.8.4 相鄰尖峰移除

在上一步驟尋找到的結果中，若有相鄰二 Window 內皆有找到符合條件之尖峰值且二尖峰所在位置相差小於 $\lceil \frac{N_{ifft}}{L_{RA}} \rceil$ 則視為一延遲過久之訊號而將值較小者移除。圖3.10為一延遲訊號之示意圖，最上方所接收之訊號，為一時間點完美吻合之訊號；第二組訊號則有稍微延遲；第三組訊號則有嚴重之延遲，故其有可能於相鄰之 Window 內產生尖峰值，因此僅留下功率較大者。以圖3.9為例，因第一 Window 與第二 Window 內的最大值位置過於相近，故視之為同一組 Preamble，因此在移除相鄰尖峰後，僅會剩下第二與第三 Window 的結果，如圖3.11所示。

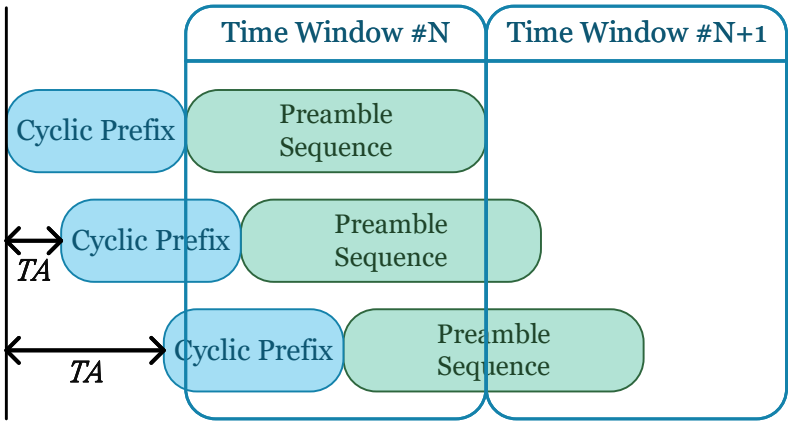



圖 3.10：延遲訊號判斷

藉由以上說明，此步驟需要 Window 間的資料互相比較，然而在 Long Preamble 的設計上，不同 Window 間的資料是放在不同 Block 中的，若彼此分享資料會需要使用到 Global Memory，而這樣的架構相當的沒有效率，會有大量的時間花費在記憶體存取上，故 Long Premable 在此步驟開始運行前，會先將各 Block 中的尖峰值資料匯集到一個 Block 內的 Shared Memory 中再往後處理。在實作上，由於一共會有 64 個 Window 的資料，所以一個 Blcok 內最少需要啟動 64 個 Thread 來代表不同的 Window 操作，所以前面小節在描述 Thread 使用數量時才會有此一限制。自此，Long Premable 與 Short Preamble 的操作架構又會相同。

於 CPU 中啟動此計算需自第一個 Window 遍歷至最後一個 Window 以判斷並移除相鄰 Window 位置過於接近的尖峰值，其時間複雜度為 $O(n)$，若於 GPU 中使用相同之方法，將無法善用 GPU 多核同步執行之優勢。本篇論文透過使用演算法3.2所描述的方式，先由偶數編號的 Thread 所代表的區間往後與之比對，並消除過於相近的尖峰值，再由奇數編號的 Thread 所代表之區間往後比對與消除。如此僅需 2 次的比對即可完成，時間複雜度降至僅 $O(1)$。

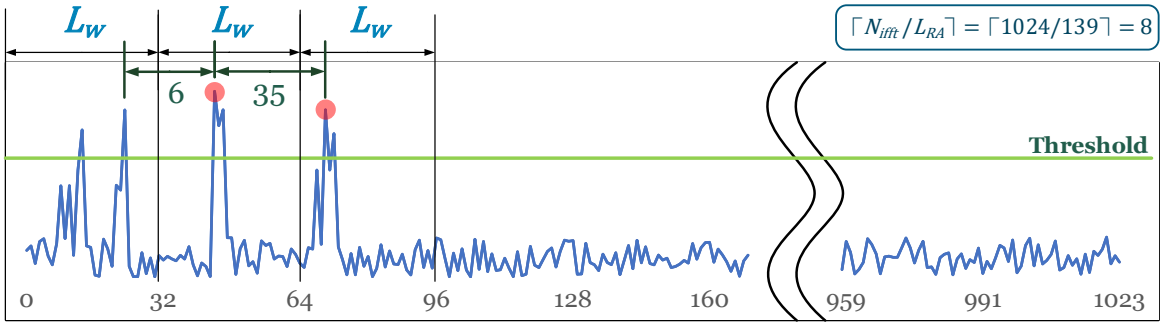



圖 3.11：移除相鄰尖峰結果

**演算法 3.2:** 移除相鄰尖峰值方法

**輸入：** 長度為 $C_v$ 的 Shared Memory $S$。
$S$ 內將含有各 Windows 內的尖峰值 $p$ 與其所在位置 $d$。

**輸出：** 已移除相鄰尖峰值的 Shared Memory $S$

$T_i \leftarrow$ Thread Index
$k \leftarrow N_{ifft}/L_{RA}$

**if** $T_i$ is even **then**
  **if** (
    $S[T_i].p > 0$ ***and*** $S[T_i+1].p > S[T_i].p$ ***and***
    $S[T_i+1].p > 0$ ***and*** $S[T_i+1].d - S[T_i].d < k$
  ) **then**
    $S[T_i].p \leftarrow 0$
  **end**
**end**

Threads Synchronization

**if** $T_i$ is odd **then**
  **if** (
    $S[T_i].p > 0$ ***and*** $S[T_i+1].p > S[T_i].p$ ***and***
    $S[T_i+1].p > 0$ ***and*** $S[T_i+1].d - S[T_i].d < k$
  ) **then**
    $S[T_i].p \leftarrow 0$
  **end**
**end**

Threads Synchronization

**return** $S$

### 3.8.5 特徵值與 $TA$ 計算

在經過前面諸多篩選後，Shared Memory 中將存有各 Window 中符合條件的尖峰值且已將視為同一 Premable 的部分剔除，故剩下的尖峰值皆視為有 UE 欲連上網或同步的行為，而各 Window 皆有其所代表之 Root Sequence 的一 $C_v$ 值，故可求出其所對應之 signature，計算方法如式 3.10所示。另外如於章節2.2.3所提到的，因 PRACH 亦有訊號傳輸時間同步之功能，故亦需計算 $TA$ 值，供後續 UE 調整傳輸時間之參考。$TA$ 值求得方法即如圖3.10所示，計算 Peak 出現位置與其 Window 開始位置之差 [32]。

$$\text{signature} = (N_{C_v}(i-1) \times i) + C_v \qquad \text{(式 3.10)}$$

至此計算完成後即代表 RACH 流程使用 GPU 優化改善架構的運算已完成。在 GPU 將估測結果回傳至 CPU 後，GPU 即進入等待狀態，待下一輪 CPU 呼叫 GPU 協助下一輪 RACH 估測運算。

# 第四章　實驗結果與討論分析

本章節將承接前章呈現 RACH 流程在經過前章的優化方法後的效率與其模擬結果分析。本章將依序說明實驗環境、實驗方法、實驗結果和分析。

## 4.1　實驗環境

### 4.1.1　硬體環境

| 硬體環境 | 規格 | |
|---|---|---|
| CPU | 處理器名稱 | Intel® Core™ i9-9960X X 系列處理器 |
| | 產品系列 | Intel® Core™ X 系列處理器 |
| | 核心/執行緒數量 | 16 / 32 |
| | 擴充指令集 | Intel® SSE4.1、Intel® SSE4.2、Intel® AVX2、Intel® AVX-512 |
| | RAM | 32 GB |
| | AVX-512 單元數量 | 2 |
| GPU-1 | 產品名稱 | NVIDIA® Quadro® P6000 |
| | 處理器名稱 | GP102 |
| | CUDA® Core 數量 | 3840 |
| | 一 SM 內 CUDA® Core 數量 | 128 |
| | SM 數量 | 30 |
| | Shared Memory | 96 KB |
| | Global Memory | 24 GB GDDR5X |
| | 計算能力版本 | 6.1 |
| GPU-2 | 產品名稱 | NVIDIA® Tesla® T4 |
| | 處理器名稱 | TU104 |
| | CUDA® Core 數量 | 2560 |
| | 一 SM 內 CUDA® Core 數量 | 64 |
| | SM 數量 | 40 |
| | Shared Memory | 64 KB |
| | Global Memory | 16 GB GDDR6 |
| | 計算能力版本 | 7.5 |

表 4.1：實驗硬體環境

表4.1呈現本論文實驗的硬體設備之重要規格。於 CPU 環境中列出的 AVX-512 欄位全名為「Intel® 進階向量延伸指令集 512」(Intel® Advanced Vector Extensions 512, Intel® AVX-512)；簡言之，AVX-512 為一特殊指令集（Instruction Set)，提供 512 Bits 向量運算的功能，例如可以在一次指令內完成 8 個 64 Bits 或 16 個 32 Bits 的整數運算，不需使用迴多次運算，進而加速在 CPU 的運算效率 [33]。

另外，因本論文將比較優化後的流程與演算法在不同 GPU 上是否有不同的運算效率表現，故表4.1亦列出二張後續比較用的 GPU 硬體規格。可以看出二張 GPU 的規格有相當的差異，如雖然 GPU-1 的 SM 數量較少，可是一 SM 內所包含的 CUDA® Core 數量卻較多。

### 4.1.2 軟體環境

| 軟體執行環境 | 規格 | |
|---|---|---|
| CPU | 作業系統 | CentOS Linux Release 7.8.2003 |
| | Linux 核心版本 | 3.10.0-1127.10.1.el7 |
| | 編譯器 | Intel® C++ Compiler |
| | 編譯器版本 | 19.0.5.281 |
| GPU | CUDA 編譯器版本 | 11.0.167 |
| | CUDA 分析工具 | nvprof |
| | nvprof 版本 | 11.0.167 |

表 4.2：實驗軟體環境

nvprof 為 CUDA toolkit 內提供的一項工具，此工具可以讓 GPU 程式開發者分析 GPU 程式的執行效能、執行指令、記憶體使用效能等 [19]。

## 4.2 實驗內容

本節將會說明本論文的實驗內容與方法。第三章為本論文最終的設計架構流程說明，然本論文在將整體流程架構透過 GPU 優化改進時，有多次的修改與調整，以下將說明歷次修改版本與其內容，以利後續的分析比較。在第二章時已說明 PRACH 可以根據不同場域環境設置不同參數，而不同的參數將導致後續 RACH 的運算量有相當大的差異，故也會在本節說明本論文的模擬環境參數。另外，也會說明本論文欲比較的項目與實驗方法。

### 4.2.1 程式優化版本

因 High-PHY 的 RACH 流程較為複雜，故在優化程式編寫過程中有過多次修改與調整，圖4.1將說明各版本的改善方向重點。

修改歷史

**原始CPU版本**

- 原始CPU處理流程內含有大量的迴圈在進行大量的重複運算。
- 原始處理流程如圖4.2所示。

**GPU版本 v1**

- 依據圖4.2的流程套用至GPU改寫。
- 流程中間的Correlation、逆向傅立葉轉換、訊號疊合、噪音估測、Preamble估測改使用GPU完成，外部迴圈控制仍由CPU完成。

**GPU版本 v2**

- 使用更多的GPU資源，以同步計算v1版本在CPU重複執行的所有迴圈控制。

**GPU版本 v3**

- v1、v2版本的GPU程式在設計上使用CUDA Unified Memory。v3改採用Device Memory。

**GPU版本 v4**

- 修正演算法，使整體運算流程如第三章所述。
- 將訊號疊合的計算往前移、使用基準功率作為噪音判斷之基準。

圖 4.1：RACH 中 High-PHY 使用 GPU 優化的修改歷史

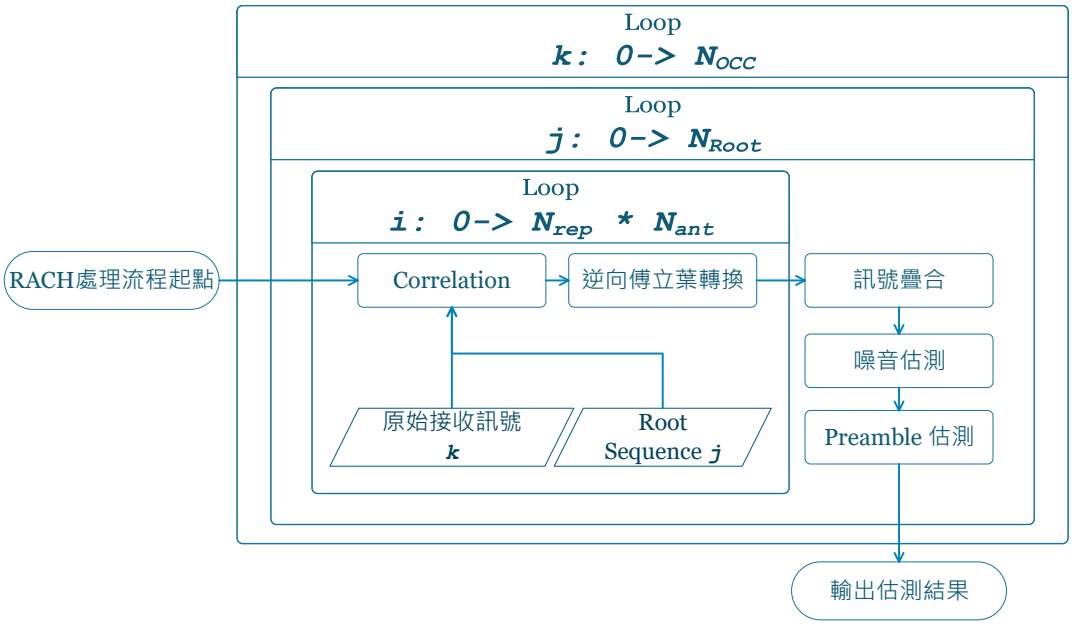



圖 4.2：原始透過 CPU 處理 RACH 中的 High-PHY 運算流程

### 4.2.2 測試參數

如同本節開頭說明，由於 PRACH 可根據不同場域有不同的設定，使得在處理訊號的 RACH 流程上有不同的運算資料量。表4.3列出後續將比較的模擬測試環境參數。

| 編號 | $L_{RA}$ | $N_{fft}$ | $N_{rep}$ |
|---|---|---|---|
| A-1 | 839 | 49152 | 1 |
| A-2 | 139 | 2048 | 12 |

(a) Low-PHY 測試環境參數

| 編號 | $L_{RA}$ | $N_{OCC}$ | $N_{root}$ | $N_{ant}$ | $N_{rep}$ | $N_{ifft}$ |
|---|---|---|---|---|---|---|
| B-1 | 839 | 1 | 2 | 4 | 1 | 2048 |
| B-2 | 139 | 7 | 4 | 2 | 2 | 1024 |
| B-3 | 139 | 1 | 4 | 2 | 12 | 1024 |

(b) High-PHY 測試環境參數

表 4.3：實驗模擬測試環境參數

### 4.2.3 實驗方法

因本論文是透過 GPU 優化整體 RACH 運算流程，因此以下各節將會依序比較 Low-PHY 與 High-PHY 中各步驟與總流程在優化前後執行時間的差別，也因為本論文將比較優化後的流程與演算法在不同 GPU 上的運算效率表現，因此後續在呈現優化後的執行時間，皆會有 GPU-1 與 GPU-2 的二種時間標示。

在行動基地台中，PRACH 介面的功能為提供一資源與方法使 UE 可開始後續一系列連上網路註冊流程與提供 UE 在 Uplink 傳輸上的同步資訊，因此若判斷或計算錯誤將導致後續 UE 可能無法連上網路，或是後續於 Uplink 上的傳送時機錯誤。雖然本論文旨在優化 RACH 的運算流程，其估測與判斷結果仍需正確，這樣的優化才有意義。因此後續各種實驗的執行背後皆是建立在 signature 與 $TA$ 值估測結果與原平台相同的條件之上，甚至有嚴格模擬測試，增加為兩倍 UE 數量訊號，並要求其值皆須正確。

## 4.3 Low-PHY 優化實驗結果與分析

因 Low-PHY 的流程與運算相對簡單，故 Low-PHY 部分的運算流程於優化前後並無太大差異，僅將 FFT 的運算改透過 GPU 計算，表4.4與表4.5分別呈現 Low-PHY 在透過 GPU 優化前後的執行時間；圖4.3則呈現執行時間的比較長條圖。

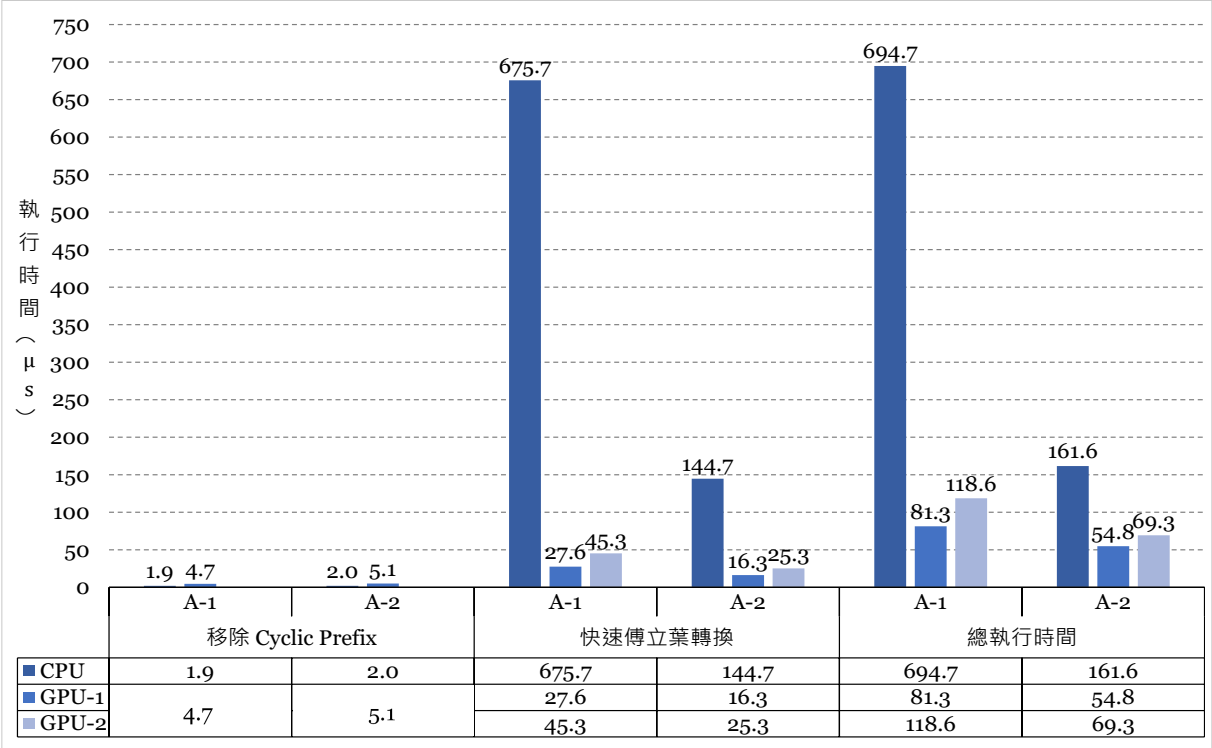
| 測試編號 / 執行流程 | A-1 | | A-2 | |
|---|---|---|---|---|
| | 運算量 | 執行時間（$\mu$s） | 運算量 | 執行時間（$\mu$s） |
| 移除 Cyclic Prefix | - | 1.94 | - | 2.03 |
| 快速傅立葉轉換[1] | 49152 | 675.70 | 24576 | 144.72 |
| 總執行時間（$\mu$s） | 694.73 | | 161.64 | |

表 4.4：Low-PHY 執行時間 - 原始版本

[1]運算量單位為：點。A-1 測資之運算量即為 49152 點的 FFT 運算。

| 執行流程 \ 測試環境 | 執行時間（$\mu$s） | | | | | |
|---|---|---|---|---|---|---|
| | A-1 | | | A-2 | | |
| | 運算量 | GPU-1 | GPU-2 | 運算量 | GPU-1 | GPU-2 |
| 移除 Cyclic Prefix | - | 4.67 | | - | 5.08 | |
| 快速傅立葉轉換 | 49152 | 27.56 | 45.29 | 24576 | 16.26 | 25.25 |
| 總執行時間（$\mu$s） | - | 81.29 | 118.60 | - | 54.78 | 69.28 |

表 4.5：Low-PHY 執行時間 - 使用 GPU 優化後版本




| | 移除 Cyclic Prefix | | 快速傅立葉轉換 | | 總執行時間 | |
|---|---|---|---|---|---|---|
| | A-1 | A-2 | A-1 | A-2 | A-1 | A-2 |
| CPU | 1.9 | 2.0 | 675.7 | 144.7 | 694.7 | 161.6 |
| GPU-1 | 4.7 | 5.1 | 27.6 | 16.3 | 81.3 | 54.8 |
| GPU-2 | | | 45.3 | 25.3 | 118.6 | 69.3 |

圖 4.3：Low-PHY 優化前後執行時間比較

### 4.3.1 移除 Cyclic Prefix

因為此步驟在優化前後皆在 CPU 執行的，故在優化後執行時間的列表上僅呈現一數值。原始架構僅需計算輸入資料中 Preamble Sequence 的位移量，而優化後的架構則須另外計算 GPU 記憶體對齊位址，故花費時間較久。

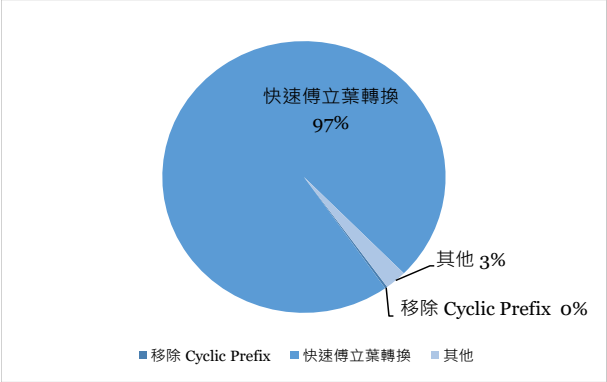
### 4.3.2 快速傅立葉轉換

如在章節3.3提到的，此計算借助 CUDA toolkit 當中的 cuFFT 函式庫協助完成，因而可將運算時間複雜度降至 $O(n \lg n)$，故當欲計算的 FFT 點數越多時，借助 GPU 運算的優點就越顯著，此表現亦可由圖4.3當中的快速傅立葉轉換欄位觀察到；由於 A-1 測資的 FFT 運算點量為 49152，故在透過 GPU 優化後的執行時間下降幅度遠大於運算點

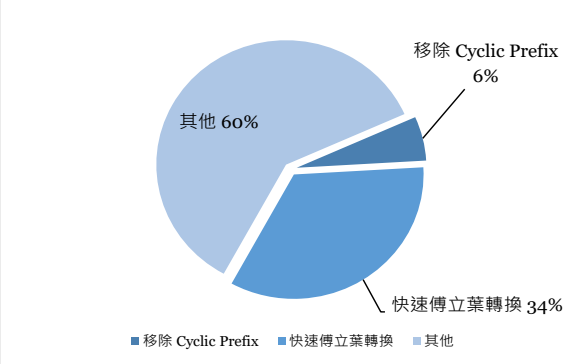
量為 24576 的 A-2 測資。

### 4.3.3 Low-PHY 流程執行時間

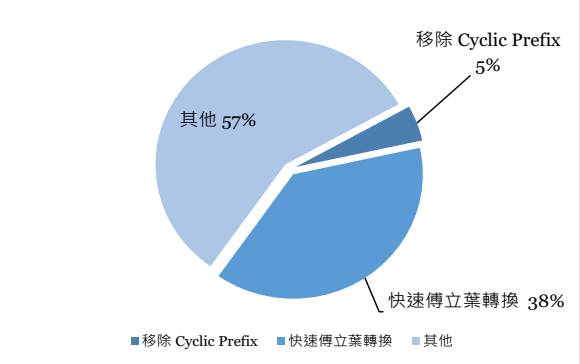
在 Low-PHY 流程上，採用優化架構前後的各步驟執行時間占比呈現於圖4.4。原先佔比相當大的 FFT 運算在優化後大幅地降低，取而代之佔執行時間最久的是標示為「其他」的項目，此項目即為使用 GPU 優化的成本，其中包含：複製運算資料至 GPU 記憶體、啟動 GPU 時間、GPU 與 CPU 工作切換的時間等，雖然此部分佔比相當大，但是因為總體執行時間下降的非常低，故使用 GPU 優化改善運算仍是相當划算的。




(a) 原始架構




(b) 優化架構—使用 GPU-1 版本




(c) 優化架構—使用 GPU-2 版本

圖 4.4：Low-PHY 各步驟執行時間佔比

## 4.4 High-PHY 初始化階段優化實驗結果與分析

如在第二、三章所說明，在程式剛啟動初始化時會先計算後續比較用的 Root Sequence，而此段初始化在原始架構與使用 GPU 優化後的執行時間分別列在表4.6與表4.7。

[2]運算量欄位代表執行一次該運算所產出的序列總長度。

| 測試編號 / 執行流程 | B-1 | | B-2 | | B-3 | |
|---|---|---|---|---|---|---|
| | 運算量 | 執行時間（$\mu s$） | 運算量 | 執行時間（$\mu s$） | 運算量 | 執行時間（$\mu s$） |
| 生成 Root 序列[2] | 839 × 1 | 49273 | 139 × 1 | 2153.7 | 139 × 1 | 1893 |

表 4.6：High-PHY 初始化階段執行時間 - 原始版本

| 測試環境 / 執行流程 | 執行時間（$\mu s$） | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | B-1 | | | B-2 | | | B-3 | | |
| | 運算量 | GPU-1 | GPU-2 | 運算量 | GPU-1 | GPU-2 | 運算量 | GPU-1 | GPU-2 |
| 生成 Root 序列 | 839 × 1 | 266.6 | 1079.8 | 139 × 1 | 23.73 | 79.9 | 139 × 1 | 23.61 | 80.54 |

表 4.7：High-PHY 初始化階段執行時間 - 使用 GPU 優化後版本

### 4.4.1 生成 Root Sequence

在生成 Root Sequence 的時間比較圖呈現於圖4.5。由於執行時間落差非常大，故在圖表上的時間標示軸上刻度採用對數間距，避免部分數值過大造成比較圖的判讀困難。

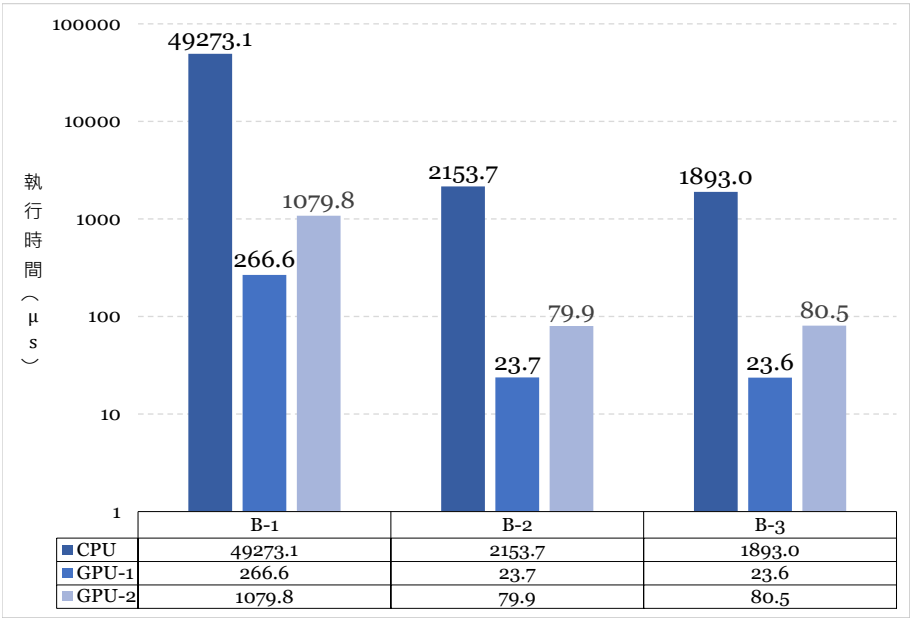



圖 4.5：生成 Root Sequence 單位執行時間比較

可以藉由表4.7觀察到在 GPU-2 上在各階段的執行時間約為 GPU-1 的 3～4 倍左右，而 GPU 在此步驟可能較花時間的點為：累加計算、啟動大量的 GPU 核心，但又因後續章節4.5的 High-PHY 當中之計算基準功率步驟也是大量累加計算，其運算時間在 GPU-1 與 GPU-2 的僅差約 50%，故此步驟在二張顯卡上的執行時間差異甚大的原因應為啟動大量的 GPU 核心不斷切換執行，因此可說明是 GPU-1 在切換 Core 執行不同工作上較有效率。

另外，亦可由圖4.5看到 GPU-1 與 GPU-2 的執行時間幾乎降到原始版本的 0.5%～

4% 左右。雖然此計算執行於基地台剛啟動時的初始化階段，對於後續實際處理訊號的運算效率並無太大幫助，但是亦可藉此優化降低基地台啟動時間並減少後續 GPU 的 Warm Up 延遲。

## 4.5 High-PHY 優化實驗結果與分析

High-PHY 流程在優化前後的執行時間分別呈現於表4.8與表4.9。由於優化後架構的流程與原始方法不同，故表中的順序按各架構實際執行的順序排列，並標示出個別步驟的運算量，往後各步驟實驗討論小節則按原始架構流程安排。也因相同原因，部分流程運算量結構差異甚大，難以比較改善前後差異，所以後續各實驗討論小節並不包含所有中間執行步驟。

| 測試編號 / 執行流程 | B-1 | | | B-2 | | | B-3 | | |
|---|---|---|---|---|---|---|---|---|---|
| | 運算量 | 執行時間（$\mu$s） | 須執行次數 | 運算量 | 執行時間（$\mu$s） | 須執行次數 | 運算量 | 執行時間（$\mu$s） | 須執行次數 |
| Correlation [3] | 839 × 1 | 5.17 | 8 | 139×1 | 2.89 | 112 | 139×1 | 1.34 | 96 |
| IFFT [4] | 2048 × 1 | 9.05 | 8 | 1024×1 | 3.8 | 112 | 1024×1 | 1.70 | 96 |
| 訊號疊合 [5] | 2048 × 1 | 6.24 | 2 | 1024×1 | 2.92 | 28 | 1024×1 | 7.35 | 4 |
| 噪音估測 [6] | 2048 × 1 | 3.23 | 2 | 1024×1 | 3.74 | 28 | 1024×1 | 1.94 | 4 |
| Preamble 估測 [7] | 2048/55 | 3.27 | 2 | 1024/17 | 2.66 | 28 | 1024/17 | 2.36 | 4 |
| 總執行時間（$\mu$s） | 294.18 | | | 1378.42 | | | 572.55 | | |

表 4.8：High-PHY 各階段執行時間 - 原始版本

| 測試環境 / 執行流程 | 執行時間（$\mu$s） | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | B-1 | | | B-2 | | | B-3 | | |
| | 運算量 | GPU-1 | GPU-2 | 運算量 | GPU-1 | GPU-2 | 運算量 | GPU-1 | GPU-2 |
| 訊號疊合 | 839×4 | 14.2 | 18.69 | 139×28 | 13.16 | 18.28 | 139×24 | 15.0 | 18.44 |
| 計算基準功率 | 839×1 | 10.51 | 16.70 | 139×7 | 11.03 | 16.32 | 139×1 | 10.0 | 15.54 |
| Correlation | 839×2 | 9.17 | 12.83 | 139×28 | 9.4 | 11 | 139×4 | 9.4 | 10.65 |
| IFFT | 2048×2 | 26.22 | 34.56 | 1024×28 | 27.5 | 33.99 | 1024×4 | 27.7 | 33.19 |
| Preamble 估測 | 4096/64 | 64.54 | 74.8 | 28672/448 | 80.01 | 96.87 | 4096/64 | 81.06 | 99.55 |
| 總執行時間 | - | 127.1 | 158.0 | - | 135.5 | 154.5 | - | 141.82 | 161.9 |

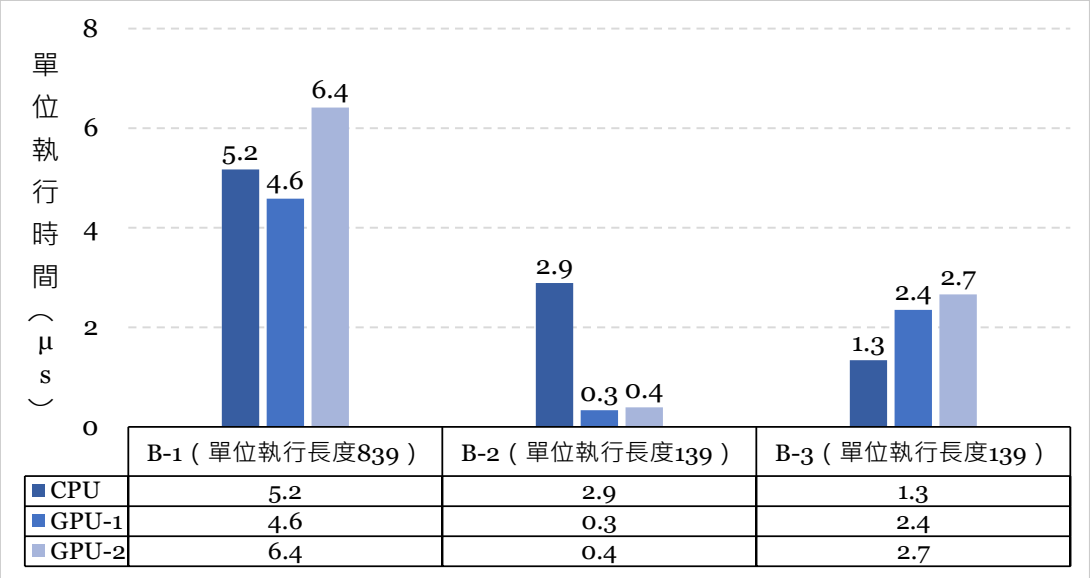
表 4.9：High-PHY 執行時間 - 使用 GPU 優化後版本

[3]運算量欄位為執行一次該運算所產出的 Corrleation 結果長度。
[4]運算量欄位為執行一次該運算所需計算的 IFFT 長度。
[5]運算量欄位為欲疊合的輸入序列總長度。
[6]運算量欄位為輸入序列的總長度。
[7]運算量欄位為：序列長度/Window 數量（因每一序列可取 Window 數量不一定相同，故在此取最大 Window 數量）。

### 4.5.1 Correlation

圖4.6呈現在優化前後計算出一組 Correlation 序列所需的平均時間。由於已經在前面的步驟將重複訊號疊合，故雖然在計算相同長度之下的 B-1 與 B-3 測試條件下執行時間較久，但僅需計算 $N_{OCC}$ 組序列，故此流程的總執行時間仍較原始架構低。

另外亦可觀察 B-2 與 B-3 的測試條件執行效率，雖然 B-2 測試條件的計算量為 B-3 條件的 7 倍，但此流程的執行時間卻是差不多的，由此可看出使用 GPU 計算之特性：若個別比較 CPU 與 GPU 單一核心的運算效率，CPU 將占優勢（B-1、B-3 測試條件的總執行時間），但是在面對大量的運算資料時，GPU 需要的時間卻相差無幾，因此在攤分時間後，單位運算量所需的計算時間即可降低（B-2、B-3 測試條件相同單位的執行時間）。

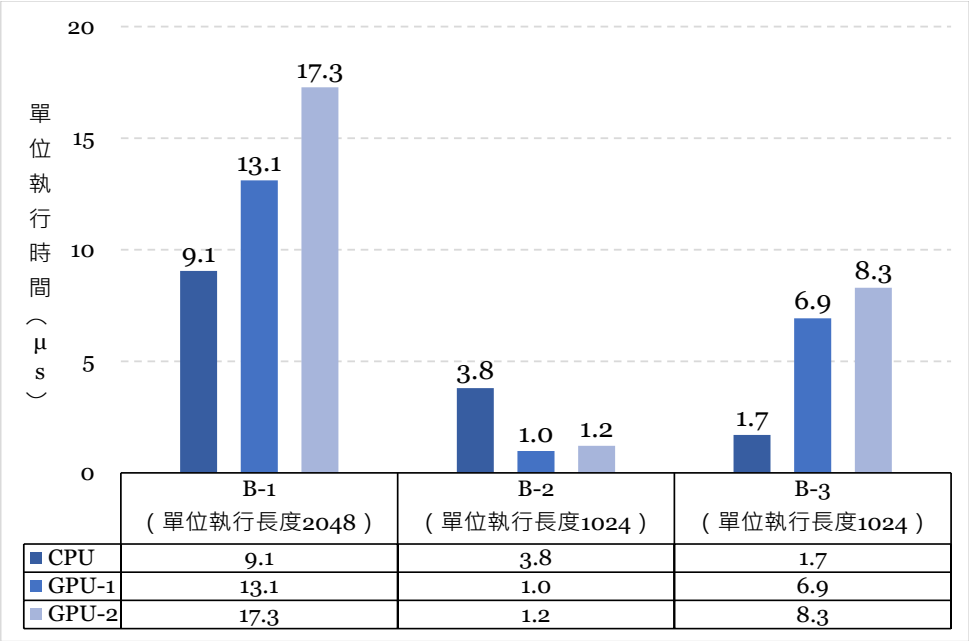



| | B-1（單位執行長度839） | B-2（單位執行長度139） | B-3（單位執行長度139） |
|---|---|---|---|
| CPU | 5.2 | 2.9 | 1.3 |
| GPU-1 | 4.6 | 0.3 | 2.4 |
| GPU-2 | 6.4 | 0.4 | 2.7 |

圖 4.6：Correlation 優化前後執行時間比較

### 4.5.2 IFFT

圖4.7呈現在優化前後計算出一序列所需的平均時間。由於在此轉換的計算量不如 Low-PHY 大，故優化前後的時間差異並不如章節4.3.2的 FFT 執行時間差異大。但仍可藉由 B-2 與 B-3 測試條件的結果看出：即使在大量資料量的情況下，IFFT 總體執行時間仍沒有太大差異，GPU 可以藉由切換閒置核心隱藏資料存取、單一核心運算效率低的缺點。

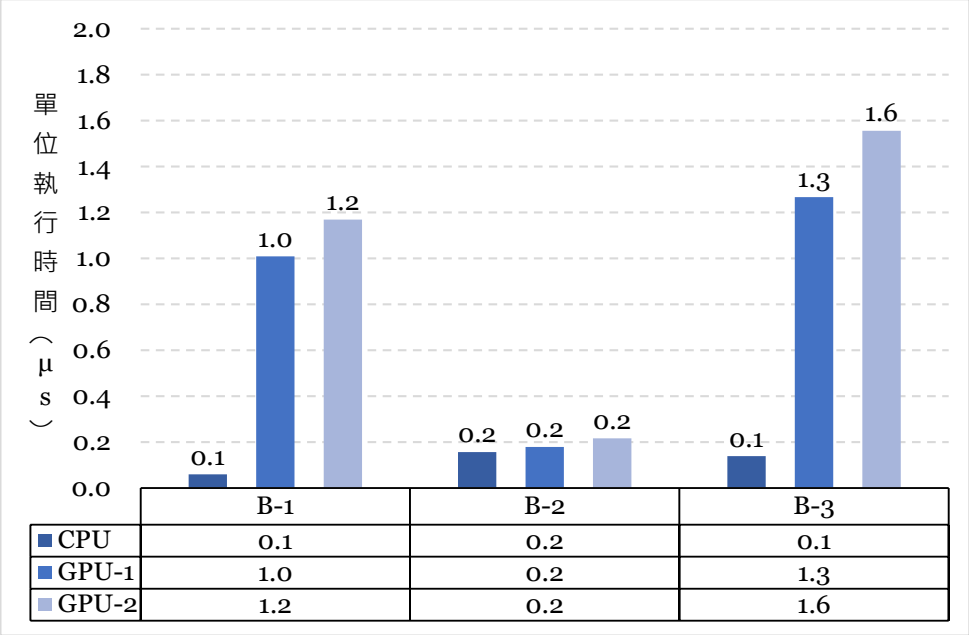



| | B-1（單位執行長度2048） | B-2（單位執行長度1024） | B-3（單位執行長度1024） |
|---|---|---|---|
| CPU | 9.1 | 3.8 | 1.7 |
| GPU-1 | 13.1 | 1.0 | 6.9 |
| GPU-2 | 17.3 | 1.2 | 8.3 |

圖 4.7：IFFT 優化前後執行時間比較

### 4.5.3 Preamble 估測

圖4.8呈現在優化前後找出一 Window 內的尖峰值所需所需的平均時間。在此部分執行時間並沒有預期的下降，誠如前面多個小節步驟的實驗結果所得：GPU 非常適合大量且重複相同的運算；而 Preamble 估測的流程並不符合這種特性，故如果沒有 B-2 測試條件大量的計算支撐，GPU 在單位長度的平均執行效率上其實是差於 CPU 的。



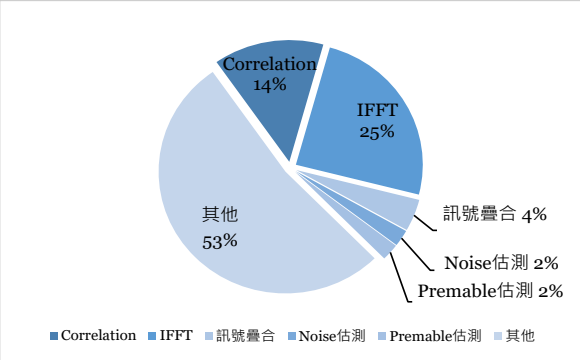

| | B-1 | B-2 | B-3 |
|---|---|---|---|
| CPU | 0.1 | 0.2 | 0.1 |
| GPU-1 | 1.0 | 0.2 | 1.3 |
| GPU-2 | 1.2 | 0.2 | 1.6 |

圖 4.8：Preamble 估測優化前後執行時間比較

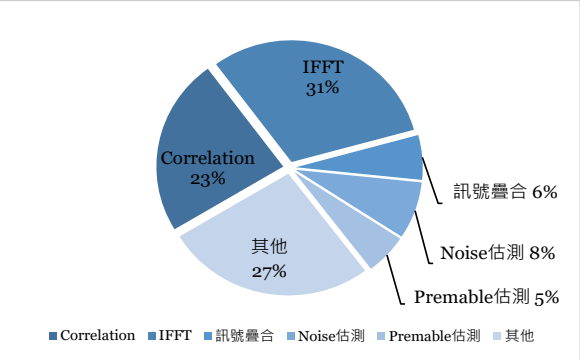
### 4.5.4 總執行時間

圖4.9與圖4.10呈現原始與優化後各執行流程的運行時間佔比。圖4.9中佔比最大的為「其他」項目，在原始架構中的其他項目主要是因為 OS 的排程切換而造成的閒置、切換工作的時間，而這部分甚至在 B-1 測試佔比超過一半，因此若能將此部分透過系統優化，B-1 條件的執行時間應該可以再降低。另外 CPU 除在切換工作以外，最花時間的項目即為 IFFT 與 Correlation，這二項即為複雜的數學運算，而其他較為簡單的累加計算與邏輯判斷則為 CPU 的運算強項。

由於 GPU-1 與 GPU-2 在 High-PHY 的執行時間差不多，故在圖4.10僅呈現 GPU-1 的流程執行時間佔比。優化前後佔比差異最大為 Preamnble 估測的部分，Preamble 估測甚至在 B-1 條件下佔 51%，此即為與 CPU 最大的差異，GPU 在執行複雜的 IFFT、Correlation 運算較 CPU 有效率，但是在做簡單的邏輯判斷時，GPU 反而較無優勢。在優化架構上 High-PHY 標示為「其他」項目的時間佔比相較於 Low-PHY 顯得下降許多，主要原因是這段時間被多項運算隱藏；因為 GPU 可以同時一邊執行運算，一邊準備下一 GPU 啟動所需的資源，因此相鄰運算流程的時間是有重疊的，故此段時間佔比因而下降許多。

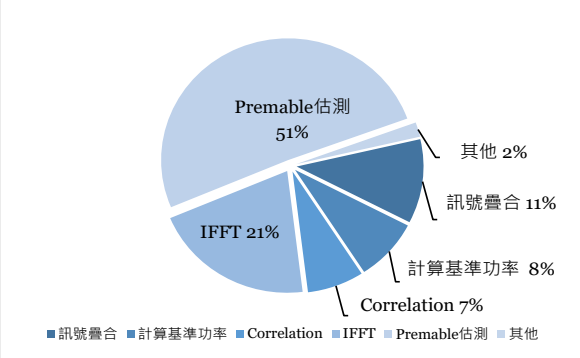



(a) B-1 測試條件

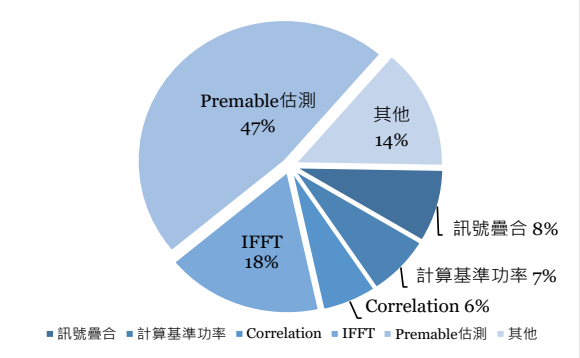



(b) B-2 測試條件

圖 4.9：原始架構的 High-PHY 各步驟執行時間佔比




(a) B-1 測試條件

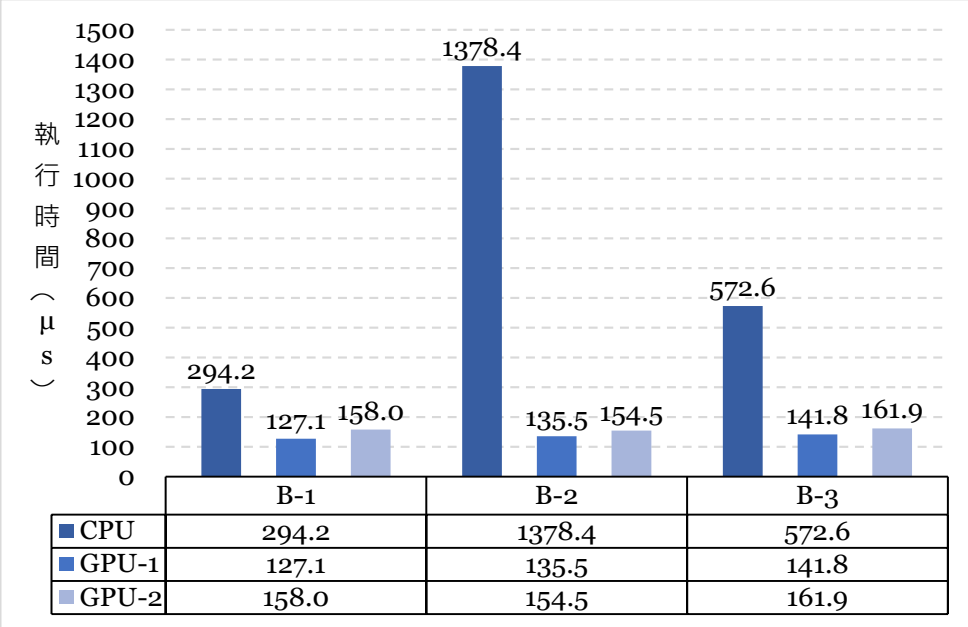



(b) B-2 測試條件

圖 4.10：優化後版本在 High-PHY 各步驟執行時間佔比

整體 High-PHY 在優化前後的平均執行時間呈現於圖4.11，在各種條件下，優化後架構的執行時間皆是下降的。B-2 與 B-3 測資因為相關的運算條件相當接近，僅是資料

量的倍數不同，而原始架構會因此差異使執行時間差約 150%，但優化後的架構卻不會因此有太大的差異。因此在 B-2 測試條件下執行時間僅為原本的 10%、B-3 測試條件則約為原本的 25%。同時在 B-1 測資上，因透過 GPU 同時計算的資料量並不如原始架構多太多，且將訊號疊合步驟往前移對於後續步驟需要執行的運算量並無顯著差異，故執行時間的僅約下降至原本的 50%。




| | B-1 | B-2 | B-3 |
|---|---|---|---|
| CPU | 294.2 | 1378.4 | 572.6 |
| GPU-1 | 127.1 | 135.5 | 141.8 |
| GPU-2 | 158.0 | 154.5 | 161.9 |

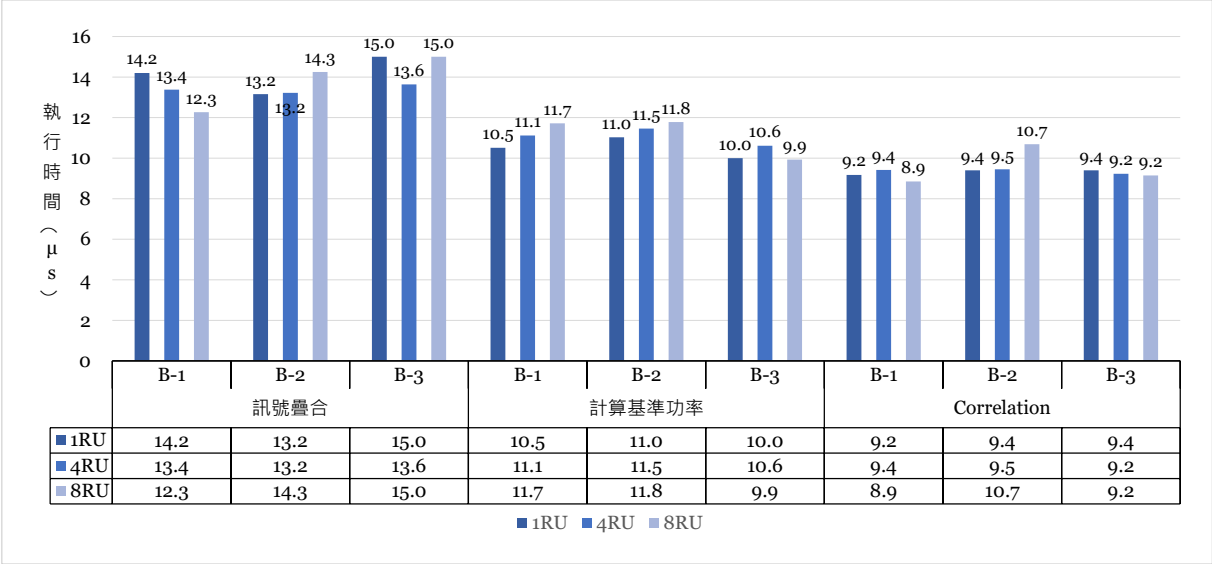
圖 4.11：High-PHY 優化前後的執行時間比較

## 4.6 多 RU 執行環境模擬測試

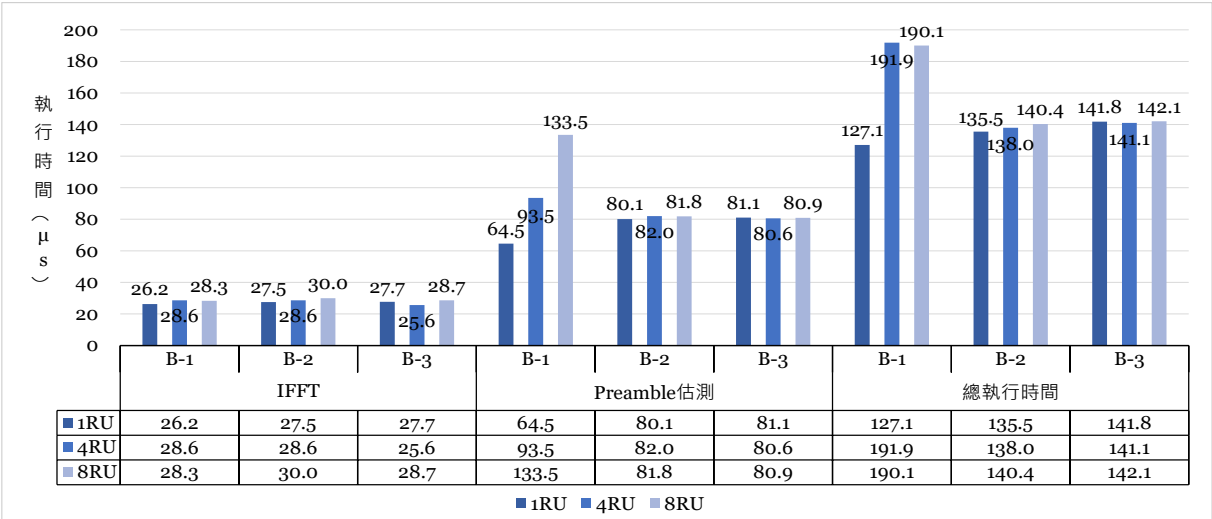
由於在設計新架構時，有加入支援多 RU 的運算環境，表4.10呈現加入多 RU 的資料後 High-PHY 運算執行時間，圖4.12則是加入前節 1RU 的執行時間一起比較，由於 GPU-1 與 GPU-2 的執行並無太大差異，故圖中僅列出於 GPU-1 的執行時間。

隨著 RU 數增加，資料量加倍的情況下，個別步驟執行時間普遍無明顯上漲趨勢，唯一明顯差異發生在 B-1 測資於 Preamble 估測階段，該階段的優化架構在分攤多 RU 資料的部分使用 CUDA 中的 Stream 功能完成，因此可以說明 CUDA 的 Stream 平行度是較 Block 差的。透過 CUDA 提供的 nvprof 工具觀察在 8 RU 的測試環境下 Stream 如何啟動平行計算，其執行概況約可用圖4.13表示。圖4.13呈現了 GPU 對於 Stream 的執行為依序啟動多個工作並行（concurrent）運算，因為這些運算並非同時啟動與同時停止，故平行度會較 Thread 與 Block 低，但是個別執行時間仍然是重疊的，因此當使用的 Stream 數量變多，也不會使總執行時間呈線性成長。但是也因為 Stream 之間的資料與資源是完全不共用的，所以 Stream 之間的工作切換代價較高，故當 Stream 使用數量增加時，因切換工作而造成的延遲將會增加，進而使個別 Stream 的執行時間增加。

圖4.12說明即使是在 8 RU，資料量提高到 8 倍的情況下，總執行時間與 1RU 情境無太大的差異，因此可以說明程式在透過 GPU 優化後，在 GPU-1 與 GPU-2 實際使用到的 GPU 資源都在 1/8 以下。




| | 訊號疊合 | | | 計算基準功率 | | | Correlation | | |
|---|---|---|---|---|---|---|---|---|---|
| | B-1 | B-2 | B-3 | B-1 | B-2 | B-3 | B-1 | B-2 | B-3 |
| 1RU | 14.2 | 13.2 | 15.0 | 10.5 | 11.0 | 10.0 | 9.2 | 9.4 | 9.4 |
| 4RU | 13.4 | 13.2 | 13.6 | 11.1 | 11.5 | 10.6 | 9.4 | 9.5 | 9.2 |
| 8RU | 12.3 | 14.3 | 15.0 | 11.7 | 11.8 | 9.9 | 8.9 | 10.7 | 9.2 |




| | IFFT | | | Preamble估測 | | | 總執行時間 | | |
|---|---|---|---|---|---|---|---|---|---|
| | B-1 | B-2 | B-3 | B-1 | B-2 | B-3 | B-1 | B-2 | B-3 |
| 1RU | 26.2 | 27.5 | 27.7 | 64.5 | 80.1 | 81.1 | 127.1 | 135.5 | 141.8 |
| 4RU | 28.6 | 28.6 | 25.6 | 93.5 | 82.0 | 80.6 | 191.9 | 138.0 | 141.1 |
| 8RU | 28.3 | 30.0 | 28.7 | 133.5 | 81.8 | 80.9 | 190.1 | 140.4 | 142.1 |

圖 4.12：多 RU 執行環境模擬測試運算時間比較

| 測試環境 / 執行流程 | RU 數量 | 執行時間（$\mu s$） | | | | | |
|---|---|---|---|---|---|---|---|
| | | B-1 | | B-2 | | B-3 | |
| | | GPU-1 | GPU-2 | GPU-1 | GPU-2 | GPU-1 | GPU-2 |
| 訊號疊合 | 4 | 13.38 | 18.00 | 13.22 | 17.83 | 13.64 | 18.13 |
| | 8 | 12.27 | 21.53 | 14.25 | 19.34 | 15.00 | 17.80 |
| 計算基準功率 | 4 | 11.12 | 16.65 | 11.46 | 15.79 | 10.61 | 14.75 |
| | 8 | 11.72 | 16.76 | 11.78 | 15.90 | 9.93 | 14.45 |
| Correlation | 4 | 9.42 | 12.62 | 9.45 | 11.77 | 9.23 | 12.06 |
| | 8 | 8.85 | 12.79 | 10.69 | 13.42 | 9.15 | 12.08 |
| IFFT | 4 | 28.62 | 20.42 | 28.59 | 21.44 | 25.59 | 20.16 |
| | 8 | 28.33 | 20.06 | 29.98 | 25.41 | 28.65 | 20.32 |
| Preamble 估測 | 4 | 93.50 | 60.67 | 81.98 | 98.63 | 80.60 | 98.01 |
| | 8 | 133.5 | 57.26 | 81.84 | 102.00 | 80.92 | 97.09 |
| 總執行時間 | 4 | 191.9 | 154.0 | 137.96 | 157.19 | 141.07 | 158.00 |
| | 8 | 190.1 | 191.7 | 140.37 | 163.02 | 142.10 | 157.93 |

表 4.10：多 RU 執行環境模擬測試運算時間

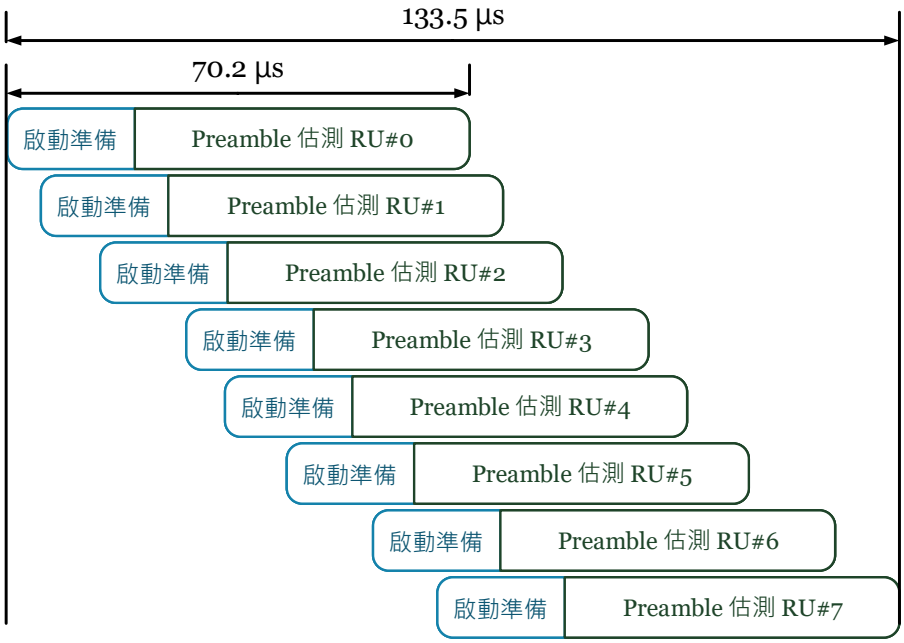



圖 4.13：Preamble 估測使用 Stream 排程示意圖

## 4.7 程式優化版本比較

於章節4.2.1有提到，程式在透過 GPU 重新設計時，中間有多次的修改版本，以下將比較各版本 High-PHY 於 GPU-1 環境的執行總時間。

| 測試環境 / 優化版本 | 執行時間（$\mu$s） | | |
|---|---|---|---|
| | B-1 | B-2 | B-3 |
| V1 | 4259.6 | 16895.6 | 16623.8 |
| V2 | 800.9 | 539.7 | 378.5 |
| V3 | 653.7 | 351.4 | 301.2 |
| V4 | 127.1 | 135.5 | 141.82 |

表 4.11：High-PHY 執行時間 - 各優化後版本

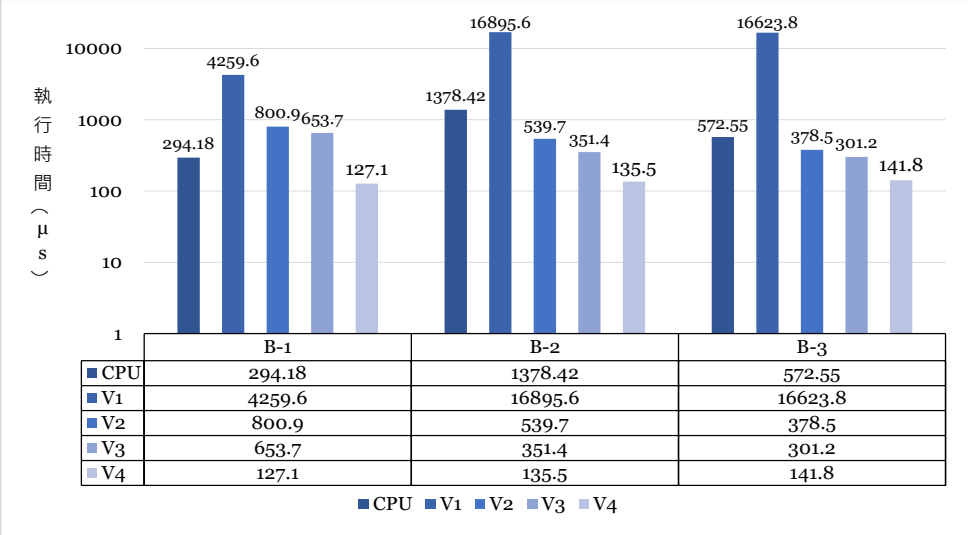



圖 4.14：High-PHY 優化版本的執行時間比較

圖4.14則是將各版本的執行時間與原始版本比較，因為數值落差相當大，故將垂直軸的數值採用對數間距以利判讀比較。在原始 V1 版本因未將個別流程的外部迴圈攤平而不斷地進出 GPU 執行小量運算，所以總執行時間反而比原先版本高出許多；由此結果可得知一程式在未依 GPU 特性優化時直接使用 GPU 運算，不僅不會有任何好處，甚至會使執行時間提升至許多倍。而在 V2 版本中，因將外部迴圈一起移至 GPU 執行同步大量的運算，已開始善用 GPU 的多核同步執行優勢，所以執行時間立刻降到與原始版本相同等級。V3 版本在修正記憶體的同步問題後也下降約 100～200$\mu$s 左右；雖然在章節2.3.5有提到諸多使用 Unified Memory 的優點，但是在 RACH 計算流程中的各項計算結果是不需要傳回至 Host Memory 的，故使用 Unified Memory 反而會有隱藏的時間延遲。最後在修改整個 RACH 於 GPU 運行的演算法與架構後，執行時間終降到可以接受之範圍內。

# 第五章　結論與未來展望

## 5.1　研究成果與建議

本論文旨在透過 GPU 提出一運算架構流程以改善傳統 RACH 運算流程之效率，據第三章設計之實作架構與第四章實驗而言，本文可歸納出以下研究成果貢獻：

- 本論文所提出之架構可將 5G NR 中的 RACH 流程運算時間依不同場域環境設定，調降至原本的 10%～50%。
- 本論文在不改變 RACH 輸出結果正確性的條件下，大幅地修改傳統 RACH 流程，使多數的計算之運算量大幅降低，進而使整體的運算執行時間降低許多。
- 因本文於實作上考慮未來擴充之彈性，故本文於第三章所提出的流程架構亦支援於 High-PHY 端同時接收多組 RU 之資料，且因已透過 GPU 平下化所有運算，所以即使有多組 RU 輸入資料，運算時間亦不會有太大差異。
- 本論文之架構並無針對特定 GPU 規格設計，因此使用相近硬體世代之 GPU 皆可有相近之效能提升，這項設計將更易使此架構受實際通訊產業所採納並應用於真實行動基地台環境。
- 本論文於實作過程不僅改善提升 RACH 之運作效率，亦對於 GPU 在運算上的特性提出各種演算法應用情境，包含將適合 GPU 的累加演算法應用於累加與相關 Reduction 運算、提出一新算法使在比對一序列相鄰元素時能夠僅有 $O(1)$ 之時間複雜度。
- 本論文使用之 GPU 生產自 NVIDIA® 公司。而 NVIDIA® 亦正在針對 5G vRAN（Virtual Radio Access Network）開發相關應用框架 [34]，本論文之設計架構對於未來 5G 商業行動網路發展應有其實際幫助。

但目前的研究尚未考慮到 CPU 與 GPU 傳輸資料成本代價，由於目前的 CPU 平台為一整體平台，目前的研究僅能將 RACH 流程抽離使用 GPU 優化設計，因此無法對整體搬移複製資料的成本做評估與驗證。

為未來持續優化改善基地台軟體化的相關研究，本論文可根據目前的設計與實驗結果提供若干建議：

- 將原始流程中短時間大量且重複的運算使用 GPU 運算將可有大幅度的運算效能提升，以前章的實驗結果顯示，可提升到原本的 7 倍之執行效率，但若僅是大量卻無重複的運算，則可能會因為 GPU 部分先天缺點而造成效率的低落。

- 在使用 GPU 運算時，應盡量減少啟動進出 GPU 的次數，每一次啟動都會需要一定時間的啟動成本，以前章的實驗結果顯示，若僅是將個別小功能函式改用 GPU 實作，則會因頻繁的啟動、僅做少量的運算，而無法隱藏啟動成本，進而造成大量的等待延遲時間。

- 在 CUDA 程式上，有多種程式抽象化的記憶體管理方式，應仔細觀察存取這些抽象化的記憶體時，實際硬體的記憶體存取是否有間接多餘且不必要的時間成本。以前章的優化版本觀察結果而言，即使是相同的記憶體操作，卻會因為使用不同的抽象化記憶體而有不同的執行效率差別。

- 應根據 GPU 特性，將原始程式的流程與演算法改寫，而非將原始 CPU 演算法直接使用 GPU 實現。實作應考慮原始演算法所需解決的問題，改採適合 GPU 運算特性的演算法。如本論文的累加演算法、前後比較演算法等，皆無法於 CPU 實現，卻能於 GPU 實現且在結果一致的條件下有極佳的效率提升。

## 5.2 未來研究方向

本論文在透過 GPU 設計架構時並無針對個別 GPU 硬體特性調整，而是視 GPU 為一加速運算單位設計。此設計策略的優點即為在相近的世代的硬體上，本論文的設計皆可有相近的效率提升，並不會因為硬體的不同而有太大的效能差異，如前一章節中的 GPU-1 與 GPU-2 執行效率近乎相同；然而缺點為此架構設計不一定最適合該硬體特性且發揮其最大效能。以本論文實驗用的二張顯卡為例，分別為 NVIDIA® Quadro® P6000 與 NVIDIA® Tesla® T4，依據官方文件說明，這二張顯示卡對於不同精準度的浮點計算有不同的運算效率，詳細數據呈現於表5.1。依表中數據，NVIDIA® Tesla® T4 在半精度運算是優於 NVIDIA® Tesla® P6000 之單精度運算的，而本論文在實作時皆是採用較為傳統且普遍支援的單精度運算。RACH 流程的目的僅為偵測是否有使用者欲皆入行動網路，對於運算的精準度並無太大要求，故若於 NVIDIA® Tesla® T4 使用半精度運算，應可在結果一致的條件下再一次提升運算效率，並降低運算所需的記憶體空間。於 GPU 硬體上特別設計半精度加速運算乃為近年之趨勢，故未來在針對基地台軟體化於 GPU 上優化的設計時，使用半精度運算亦為一研究方向之參考。

| 執行環境 | 理論效能 | | |
|---|---|---|---|
| | FP16（半精度） | FP32（單精度） | FP64（倍精度） |
| NVIDIA® Quadro® P6000 | 197.4 G | 12.63 T | 394.8 G |
| NVIDIA® Tesla® T4 | 65.13 T | 8.141 T | 254.4 G |

（單位：每秒浮點運算次數，FLOPS，Floating-point operations per second）

表 5.1：GPU 浮點數運算理論效能

本論文在設計新架構流程時，考慮到傳統流程上的 Coorealtion、IFFT 與訊號疊合之計算皆為線性運算，因此將訊號疊合之運算移至 High-PHY 流程的第一步，透過此改動可使計算 Correlation 與 IFFT 的資料大幅降低。因本論文旨在透過 GPU 優化運算，故未將此流程使用 CPU 實作比較，然在前章的實驗結果顯示，這些降低運算量的計算步

驟於原始架構中執行時間佔比相當大，故未來若將此一架構改使用 CPU 實作，應該也會有不錯的效能提升。

最後，由前一章實驗結果可得目前的系統設計未使 GPU 滿載，仍有餘力可協助其他軟體化基地台的其他運算，且 RACH 流程僅為整體 SDR 系統中的一部分程序，未來若在硬體負荷條件下將其他上下行訊號處理流程皆使用 GPU 運算，則 CPU 可降低訊號處理負載，轉而負責整體工作排程與資源分配，亦可維持整體系統之效能與穩定。整體軟體基地台仍有許多待優化的部分，未來若能逐一優化與設計，未來在面對更大量的無線網路訊號與更嚴謹之時間限制時，才能有足夠的效能完成整個基地台的服務。

# 參 考 文 獻


[1] Opensignal. “5G 用戶體驗報告.” (Jun. 2021), [Online]. Available: https://www.opensignal.com/zh-hant/reports/2021/06/taiwan/mobile-network-experience-5g (visited on 07/02/2021).

[2] 3GPP, “Study on scenarios and requirements for next generation access technologies,” 3rd Generation Partnership Project (3GPP), Technical Report (TR) 38.913, Jun. 2017, Version 14.3.0. [Online]. Available: https://www.3gpp.org/DynaReport/38913.htm.

[3] OpenAirInterface. “Cloud RAN (C-RAN).” (2020), [Online]. Available: https://openairinterface.org/use-cases/cloud-ran-c-ran/ (visited on 07/26/2021).

[4] 3GPP, “NR; Physical layer; General description,” 3rd Generation Partnership Project (3GPP), Technical Specification (TS) 38.201, Dec. 2017, Version 15.0.0. [Online]. Available: https://www.3gpp.org/DynaReport/38201.htm.

[5] 3GPP, “NR; Physical channels and modulation,” 3rd Generation Partnership Project (3GPP), Technical Specification (TS) 38.211, Jun. 2018, Version 15.2.0. [Online]. Available: https://www.3gpp.org/DynaReport/38211.htm.

[6] 3GPP, “NR; NR and NG-RAN Overall description; Stage-2,” 3rd Generation Partnership Project (3GPP), Technical Specification (TS) 38.300, Jun. 2018, Version 15.2.0. [Online]. Available: https://www.3gpp.org/DynaReport/38300.htm.

[7] 3GPP, “NR; Services provided by the physical layer,” 3rd Generation Partnership Project (3GPP), Technical Specification (TS) 38.202, Jun. 2018, Version 15.2.0. [Online]. Available: https://www.3gpp.org/DynaReport/38202.htm.

[8] 3GPP, “Study on new radio access technology: Radio access architecture and interfaces,” 3rd Generation Partnership Project (3GPP), Technical Report (TR) 38.801, Mar. 2017, Version 14.0.0. [Online]. Available: https://www.3gpp.org/DynaReport/38801.htm.

[9] E. Jordan. “Open RAN 101‑RU, DU, CU: Why, what, how, when? (Reader Forum).” (Jul. 8, 2020), [Online]. Available: https://www.rcrwireless.com/20200708/open_ran/open-ran-101-ru-du-cu-reader-forum (visited on 06/09/2021).

[10] 3GPP, “Evolved Universal Terrestrial Radio Access (E-UTRA); Physical channels and modulation,” 3rd Generation Partnership Project (3GPP), Technical Specification (TS) 36.211, Dec. 2009, Version 8.9.0. [Online]. Available: https://www.3gpp.org/DynaReport/36211.htm.

[11] R. Frank, S. Zadoff, and R. Heimiller, “Phase shift pulse codes with good periodic correlation properties (Corresp.),” *IRE Transactions on Information Theory*, vol. 8, no. 6, pp. 381–382, 1962. DOI: 10.1109/TIT.1962.1057786.

[12] D. Chu, “Polyphase codes with good periodic correlation properties (Corresp.),” *IEEE Transactions on Information Theory*, vol. 18, no. 4, pp. 531–532, 1972. DOI: 10.1109/TIT.1972.1054840.

[13] R. Heimiller, “Phase shift pulse codes with good periodic correlation properties,” *IRE Transactions on Information Theory*, vol. 7, no. 4, pp. 254–257, 1961. DOI: 10.1109/TIT.1961.1057655.

[14] S. Sesia, I. Toufik, and M. Baker, *LTE –The UMTS Long Term Evolution: From Theory to Practice, Second Edition*. John Wiley & Sons, Ltd, Jul. 2011, ISBN: 9780470978504. DOI: https://doi.org/10.1002/9780470978504. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9780470978504. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470978504.

[15] Y. Wen, W. Huang, and Z. Zhang, “CAZAC sequence and its application in LTE random access,” in *2006 IEEE Information Theory Workshop - ITW ’06 Chengdu*, 2006, pp. 544–547. DOI: 10.1109/ITW2.2006.323692.

[16] Y. Hu, J. Han, S. Tang, H. Gao, Y. Su, and J. Shi, “A method of PRACH detection threshold setting in LTE TDD femtocell system,” in *7th International Conference on Communications and Networking in China*, 2012, pp. 408–413. DOI: 10.1109/ChinaCom.2012.6417517.

[17] 3GPP, “RACH Desigh for EUTRA,” 3rd Generation Partnership Project (3GPP), Discussion R1-060387, Feb. 2006. [Online]. Available: https://www.3gpp.org/ftp/tsg_ran/WG1_RL1/TSGR1_44/Docs/R1-060387.zip (visited on 07/12/2021).

[18] 3GPP, “NR; Physical layer procedures for control,” 3rd Generation Partnership Project (3GPP), Technical Specification (TS) 38.213, Jun. 2018, Version 15.2.0. [Online]. Available: https://www.3gpp.org/DynaReport/38213.htm.

[19] NVIDIA, *CUDA C++ Programming Guide*, version 11.0, Aug. 6, 2020. [Online]. Available: https://docs.nvidia.com/cuda/archive/11.0/pdf/CUDA_C_Programming_Guide.pdf (visited on 06/14/2021).

[20] Khronos Group. “OpenCL Guide.” (Apr. 19, 2021), [Online]. Available: https://github.com/KhronosGroup/OpenCL-Guide (visited on 06/14/2021).

[21] S. Rennich. “CUDA C/C++, Streams and Concurrency.” (Jan. 2012), [Online]. Available: https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf (visited on 06/15/2021).

[22] A. Rege. “An Introduction to Modern GPU Architecture.” (Dec. 2008), [Online]. Available: http://download.nvidia.com/developer/cuda/seminar/TDCI_Arch.pdf (visited on 06/15/2021).

[23] NVIDIA, *CUDA C++ Best Practices Guide*, version 11.0, Aug. 6, 2020. [Online]. Available: https://docs.nvidia.com/cuda/archive/11.0/pdf/CUDA_C_Best_Practices_Guide.pdf (visited on 06/17/2021).

[24] V. Volkov, “Understanding Latency Hiding on GPUs,” Ph.D. dissertation, EECS Department, University of California, Berkeley, Aug. 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-143.html.

[25] M. Harris. “How to Access Global Memory Efficiently in CUDA C/C++ Kernels.” (Jan. 28, 2013), [Online]. Available: https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels (visited on 06/17/2021).

[26] M. Harris. “Using Shared Memory in CUDA C/C++.” (Jan. 7, 2013), [Online]. Available: https://developer.nvidia.com/blog/using-shared-memory-cuda-cc (visited on 06/17/2021).

[27] M. Harris. “Unified Memory for CUDA Beginners.” (Jun. 19, 2017), [Online]. Available: https://developer.nvidia.com/blog/unified-memory-cuda-beginners (visited on 07/25/2021).

[28] NVIDIA, “Precision and Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs,” NVIDIA Corporation, White Paper, Aug. 6, 2020, Version 11.0. [Online]. Available: https://docs.nvidia.com/cuda/archive/11.0/pdf/Floating_Point_on_NVIDIA_GPU.pdf (visited on 07/02/2021).

[29] J. Cheng, M. Grossman, and T. McKercher, *Professional CUDA® C Programming*. John Wiley & Sons, Ltd, Sep. 2014, ISBN: 9781118739310.

[30] M. Harris. “Optimizing Parallel Reduction in CUDA.” (Mar. 18, 2010), [Online]. Available: https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf (visited on 07/26/2021).

[31] NVIDIA, *CUFFT LIBRARY USER’S GUIDE*, version 11.0, Aug. 6, 2020. [Online]. Available: https://docs.nvidia.com/cuda/archive/11.0/pdf/CUFFT_Library.pdf (visited on 06/25/2021).

[32] S. Kim, K. Joo, and Y. Lim, “A delay-robust random access preamble detection algorithm for LTE system,” in *2012 IEEE Radio and Wireless Symposium*, 2012, pp. 75–78. DOI: 10.1109/RWS.2012.6175341.

[33] Intel Corporation, *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, Jun. 2016. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf (visited on 07/23/2021).

[34] S. VELAYUTHAM. “NVIDIA CEO Introduces Aerial, Software to Accelerate 5G on NVIDIA GPUs.” (Oct. 21, 2019), [Online]. Available: https://blogs.nvidia.com/blog/2019/10/21/aerial-application-framework-5g-networks/ (visited on 07/26/2021).