國立陽明交通大學

多媒體工程研究所

碩士論文

Institute of Multimedia Engineering

National Yang Ming Chiao Tung University

Master Thesis

評估利用固態硬碟加速深度學習應用之可行性

Evaluating the DNN Model Acceleration on Solid State Drive

陽明交大
NYCU

研究生：陳柏丞（CHEN, BO-CHENG）

指導教授：葉宗泰（YEH, TSUNG-TAI）

中華民國一一一年八月

August, 2022

評估利用固態硬碟加速深度學習應用之可行性

Evaluating the DNN Model Acceleration on Solid State Drive

研究生：陳柏丞　　Student：Bo-Cheng Chen

指導教授：葉宗泰 博士　　Advisor：Dr. Tsung-Tai Yeh

國立陽明交通大學

多媒體工程研究所

碩士論文

陽明交大
NYCU

A Thesis

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

August 2022

Taiwan, Republic of China

中華民國一一一年八月

評估利用固態硬碟加速深度學習應用之可行性

研究生：陳柏丞　　指導教授：葉宗泰 教授

國立陽明交通大學多媒體工程研究所

摘要

基於神經網路的個人化推薦系統已經深入人們的生活，包含社群媒體、搜索和電子商務。目前流行的推薦模型都需要大型的嵌入表，其中包含數十億個參數需要儲存在記憶體內，才能夠讓模型運作。然而要容納大量資料的記憶體空間會帶來巨大的硬體成本。本研究提出將大型的嵌入表放入現有的 SSD 內，針對推薦系統內的運算子分析不同的 I/O 方式才能夠減少運行時間。藉由預載入部分的嵌入表搭配合適的 I/O 讀取，加速特定運算子的執行時間，最高不僅有 1.67 倍的加速，同時只使用 1/8 原本記憶體使用量。

陽明交大
NYCU

關鍵字：推薦系統、固態硬碟、神經網路、加速器、鄰近數據處理

Evaluating the DNN Model Acceleration on Solid State Drive

Student：Bo-Cheng Chen Advisor：Dr. Tsung-Tai Yeh

Institute of Multimedia Engineer

National Yang Ming Chiao Tung University

Abstract

Personalized recommendations systems based on neural networks are widely used in people's lives, including social media, search and ecommerce. Large embedding tables containing billions of parameters must be stored in memory for current recommendation models to work. The memory space required to store a large amount of data, on the other hand, comes at a significant hardware cost. In order to reduce running time, our study proposes inserting embedding tables into a commercial SSD and analyzing different I/O ways for the operators of the recommendation systems. The execution time of a specific operator can be sped up to 1.67 times while using only 1/8 of the original memory usage by preloading a portion of the embedding tables with appropriate I/O read.

陽明交大
NYCU

Keyword: Recommendation System, Solid State Drive, Neural Network, Accelerator, Near Data Processing

Contents

陽明交大 NYCU

陽明交大
NYCU

List of Figures

陽明交大
NYCU

陽明交大
NYCU

List of Tables

陽明交大
NYCU

Chapter 1

Introduction

1.1 Problems

推薦系統已經廣泛的應用在人們周遭的生活上面，比如社群媒體、電子商務、搜索引擎等。以前公司會透過銷售經驗及客戶的點擊率來將產品分類，結合客戶的使用經歷，再搭配一些線性規劃的數學模型來協助推薦系統。直到現在神經網路的流行，才出現各式各樣的模型更有效的向客戶提供精確且個人化的推薦[1, 2]。其中包含 Meta 的 DLRM 模型[3]、Google 的 WnD 模型[4]、阿里巴巴的 DIN 模型[5, 6]等，然而這些模型共同特徵就是會耗費數據中心大量的運算資源。以神經網路為基礎的推薦系統都會有大型的 Embedding Tables 用來處理分類的特徵，雖然大型的 Embedding Tables 可以實現更高的個人化推薦，但是為了記錄每一種類型的產品及客戶個人的偏好，將會耗費高達數百 Gb 的儲存空間[7-9]。事實上，一個推薦模型的大小是由服務器上的記憶體大小決定，一次要放入全部的表格是不實際的而且也未必有那麼大的記憶體空間。

目前論文的策略大部分都是將 Embedding Tables 放入 SSD 內[10-13]，雖然 SSD 的性能沒有比 DRAM 好，但 SSD 的儲存空間遠大於 DRAM 且單位儲存成本較低。為了攤提掉將資料從 SSD 讀取 Embedding Tables 的時間，像是 RecSSD[13]在開源的 SSD 內加入各種模組，快取住之前讀過的 Page 及數據，如果 Host 端需要的某些資料已經在 Buffer 內就直接拿來用，不必再去內部 Flash 重新搜尋資料，這樣就可以減少 I/O 次數進而減少整體的搬運成本，研究結果顯示推薦系統推論延遲可以降低兩倍之多。

陽明交大 NYCU

1.2 Motivation

市售的 SSD 並沒有開放給使用者加入客製化模組的功能，加上先前的研究指出 DLRM 內的 SparseLengthsSum（SLS）屬於計算密度低但耗費記憶體空間的運算子，即便用 GPU 加速也沒有很大的成效[14]。想要加速推薦系統，同時 Embedding Tables 又放在 SSD 內，兩者之間的取捨就是本研究的目的。

1.3 Contribution

我們分析了各種常見的 I/O 方式對 SSD 請求時會有的特徵，以此來評估各個 I/O 在讀取 SSD 內部資料的優缺點；找出執行 SLS 時影響延遲的參數，並提供兩種方法搭配不同情境來改善極端參數時的缺陷。使用者可以基於我們的程式，測試不同 I/O 方式搭配不同的 Embedding Tables 設定，找出最合適的組合。根據 SLS 現有的問題，提供完整的分析，在讀取特定比例下的 Embedding Tables 最好可以有 1.67 倍的加速，而且只有使用約原本 1/8 的記憶體空間。

陽明交大
NYCU

1.4 Architecture

本論文第一章節會介紹研究動機，第二章節會介紹對應的背景知識，第三章節會介紹目前推薦系統的特徵及運算瓶頸，第四章節會介紹本研究的實驗設計及實作，第五章節會分析 DLRM 發出的請求特徵，找出造成瓶頸的參數，比較各種方法的優勢，最後一個章節為本研究做個總結及未來展望。

Chapter 2

Background

這個章節會概述深度學習推薦系統的架構、運算子和相關的演算法；以及固態硬碟的基本知識和內部行為；還有 Linux 系統上抓取封包的常見工具。

2.1 Deep Learning Recommendation Model

目前深度學習推薦系統分成三種模式：Embedding-dominated、MLP-dominated 以及 Attention-dominated。各自的代表有 Meta 的 DLRM 模型[3]，主要應用於社群媒體、Google 的 WnD 模型[4]，主要應用於 Play 商店、阿里巴巴的 DIN 模型[6]，主要用於廣告推薦。其中點擊率（Click-through-rates, CTRs）作為最後的輸出，其準確度對於推薦用戶產品時極為重要。

陽明交大
NYCU

DLRM 模型（Embedding-dominated），其中的運算子具有佔大量記憶體空間但低密度運算的特性[15]，因此作為本研究探討的對象。

2.1.1 Model Architecture

DLRM 模型的架構[1, 3]，如 Figure 1 所示，分為三個部分：針對 Dense Inputs 的 Bottom-MLP、針對 Sparse Inputs 的 Embedding Tables 以及將處理過後的特徵值作為預測用的 Top-MLP。

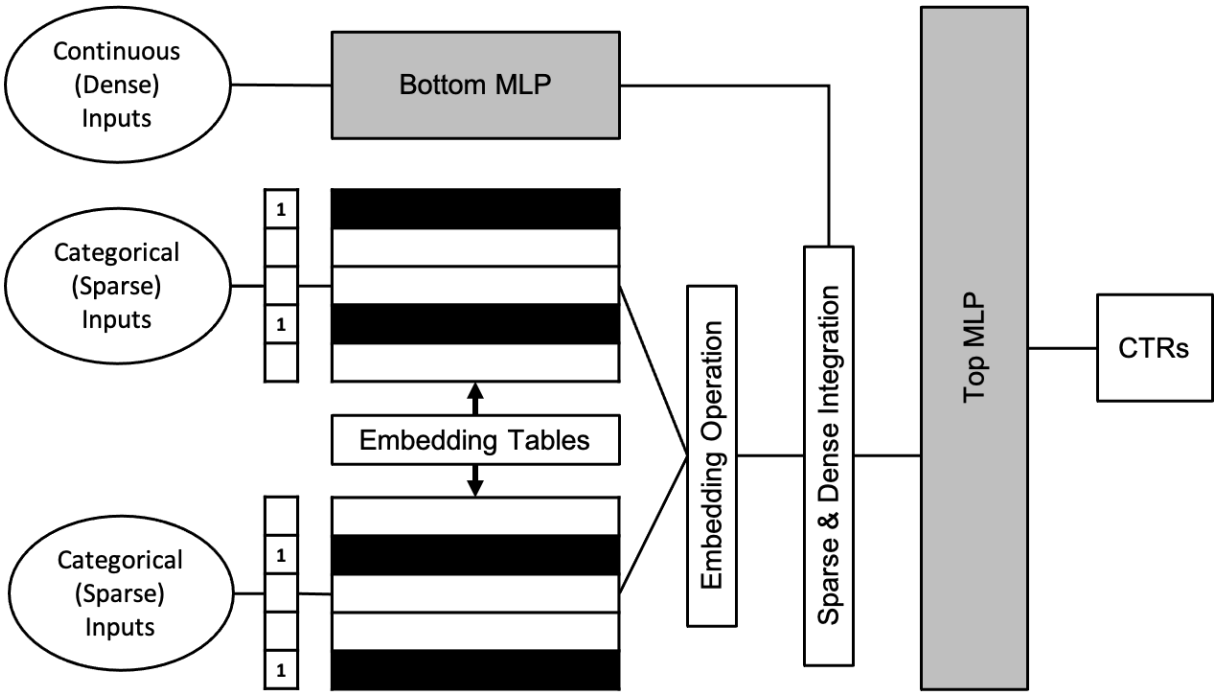


Figure 1. DLRM 的基本架構

陽明交大
NYCU

2.1.2 Dense Inputs and Sparse Inputs

DLRM 模型的輸入分為二個種類：Dense Inputs and Sparse Inputs。

Dense Inputs 又稱為 Continuous Inputs，主要是年齡、性別、使用時間等用戶資料，也就是每個使用者的都應該具備的資訊。

Sparse Inputs 又稱為 Categorical Inputs，主要是用來分類使用者喜好的特徵。比如影音平台內有上千部的電影，觀看的人數可能有上百萬個，但是每個使用者頂多觀看過幾百部，也就是說電影跟使用者的關係是非常 Sparse。根據電影的熱門程度再加上使用者偏好，Sparse Inputs 就可以做為分類的特徵。

2.1.3 Multilayer Perceptron

多層感知器（Multilayer Perceptron, MLP）是深度學習中常用的方法，至少包含三層結構：輸入層、隱藏層、輸出層，層與層之間可以再放入不同量的隱藏層，其中層

之間的節點都是相互連結（Fully Connected）。在 DLRM 中，MLP 會用於提取 Dense Inputs 的特徵和作為預測器來分類最後的特徵值。由於 MLP 層間的節點是相互連結，在訓練或推論（Inference）時有大量的矩陣運算，因此具有高密度計算的特性[15]。

2.1.4 Embedding Tables

為了處理和提取 Sparse Inputs 有效的特徵，並將其映射在 Dense Inputs，以便之後的多層感知器做預測，因此需要 Embedding Tables 的協助。其中 Sparse Inputs 會被加密成 Multi-hot 向量，向量中的值 1 代表有效、0 代表無效，也就是用戶有沒有看過該類型的電影。

鑑於潛在數據達數百萬筆，但只含有少量筆的有效數據（Sparse Inputs）。首先利用 Embedding Tables 將 Multi-hot 向量轉換成有意義的 Dense 向量，之後再與 Dense Inputs 做連接，其中稀疏特徵會使模型難以訓練再加上 Embedding Tables 需要有大量的空間及低密度計算[15]，因此有效的資料搬移將是一大挑戰。

陽明交大
NYCU

2.1.5 Sparse Lengths Sum

如何利用 Embedding Tables 將 Sparse Inputs 作轉換？需兩個步驟，分別是 Look-up 以及 Sparse Lengths Sum。前者利用經轉換過的 Multi-hot 向量在 Embedding Tables 內找到需要做運算的向量（Embedding Vectors）；後者則是將蒐集到的向量做對應項的加總，如 Figure 2 所示，結果與 Dense Inputs 做連結，再交由最後的預測層。

每一張 Embedding Tables 會根據模型的設定不同，大小從幾十 Mb 到幾十 Gb 不等，然而真的做處理的向量可能只有數十到數百個，佔全部表格不到 1%。因此要運行單一個 DLRM 模型可能要消耗幾十 Gb 的記憶體或者儲存空間但實際使用率又很少。

Algorithm 1 SparseLengthsSum

$Emb \leftarrow vector : R(EmbeddingSize) * C(FeatureSize)$
$Lengths \leftarrow vector : 1 * K(BatchSize)$
$Indices \leftarrow vector : 1 * (K * L(LookupSize))$
$Result \leftarrow vector : K * C$
$curID \leftarrow 0$
$outID \leftarrow 0$
for Length in Lengths **do**
  for Index in Indices[curID:curID+Length] **do**
    $EmbVec \leftarrow Emb[Index]$
    for i in range(C) **do**
      $Result[outID][i] \leftarrow Result[outID][i] + EmbVec[i]$
    end for
    $curID \leftarrow curID + Length$
    $outID \leftarrow outID + 1$
  end for
end for

N Y L U

Figure 2. Sparse Lengths Sum 演算法[1]

2.1.6 Challenges in DLRM

綜觀整個 DLRM 模型，主要用到的運算有 FC (Fully Connected)以及 SLS（Sparse Lengths Sum），各自的每秒浮點數運算（FLOPs）分別為 18（FLOPs/Byte）以及 0.25（FLOPs/Byte）[1]。考慮到因應不同 DLRM 的設定會影響 Embedding Tables 的大小，隨著表格的增大，對於 Embedding Vectors 的存取也就更加離散，同時記憶體的需求也就更大。如果是 Embedding-dominated 的模型，即便使用圖形處理器（GPU）做加速，整體的效果很有限[14]。

總之，想要加速 DLRM 的運行速度而且減少記憶體的用量，勢必要從 SLS 這個運算子下手。而 SLS 具備高記憶體需求低密度運算的性質，因此如何有效地存取特定的 Embedding 向量為本研究的動機。

2.2 Solid State Driver/Disk

固態硬碟（Solid State Driver, SSD），是一種以快閃記憶體（NAND Flash）為介質的非揮發性儲存裝置，性能低於 DRAM 高於機械式硬碟[16]。相較於機械式硬碟有速度快、功耗低、防震、聲音小、體積小等優點。

現今市售 SSD 循序讀寫的速度分別可以達到 3500 Mb/s 以及 3200 Mb/s 左右；隨機讀寫的速度可達 480000 IOPS 以及 550000 IOPS 左右，IOPS 全名為 Input/Output Operations Per Second，是一種用來測量儲存裝置的標準，一般隨機存取由於存取的資料較少，一次大概為 4Kb。不過 SSD 有受限於 NAND Flash 的關係，隨著長期使用，有硬體損壞時資料無法挽救、寫入次數的限制、讀取干擾及掉速的問題。

陽明交大
NYCU

2.2.1 SSD in Computer

應用程式如果想要在 SSD 寫入資料，首先會透過檔案管理系統（File System, FS），FS 會將資料分段成不同大小，再交由 Host Driver。Driver 會將資料透過多個 LBA（Logical Block Address）告訴 SSD 內部的控制器（Controller）該寫入 NAND Flash 的哪個區塊。Host 跟 Device 之間會透過協定（Protocol）像是 NVMe 等定義好的指令集和功能，其中溝通橋樑可以是 PCIe 接口，讓 Host 可以對 Device 做特定行為的操作。Controller 收到指令後會交由內部的 Flash 轉換層（Flash Translation Layer）將

LBA 轉換成 PBA（Physical Black Address），通常轉換的表格會記載內部的 DRAM，之後透過正確的位址就可以寫入 NADN Flash，如 Figure 3 所示。

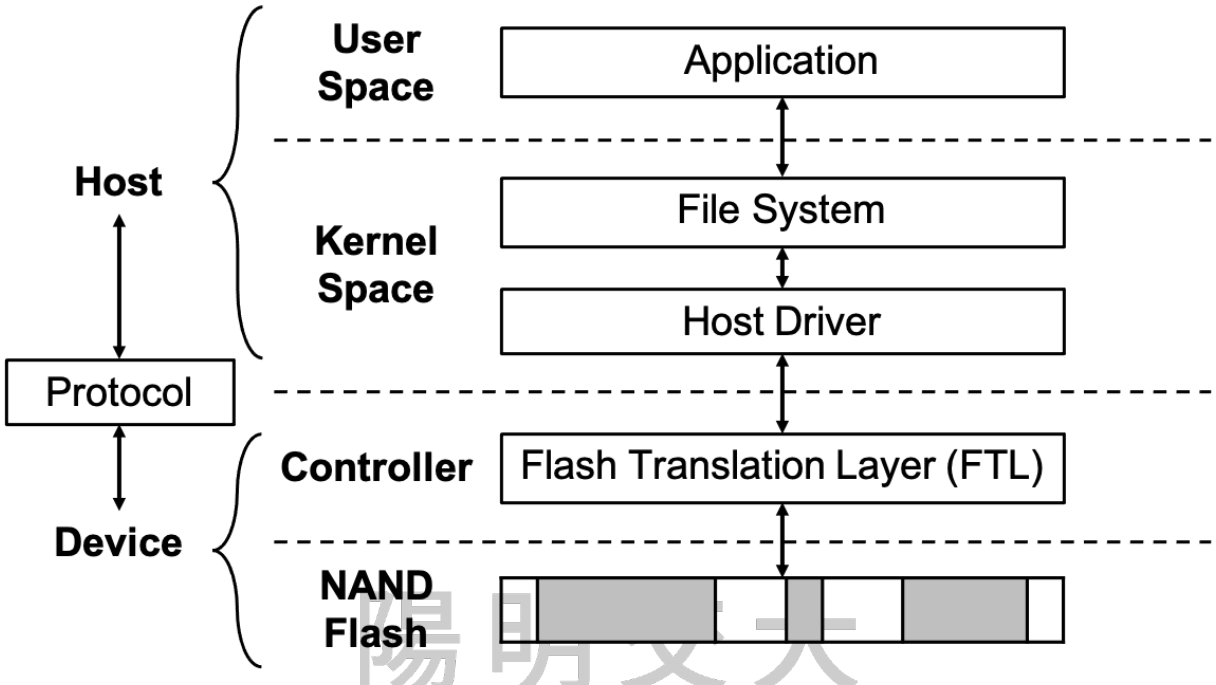


Figure 3. 從 Host 傳輸資料到 Device 的流程圖

2.2.2 NAND Flash

SSD 內部主要有 CPU、RAM、NAND Flash 等組成，其中 NAND Flash 的存放以位元（bit）為單位組成，根據不同的硬體設計可以分為 SLC、MLC、TLC、QLC，主要的差別是在每個單元（cell）可以儲存的位元數，單位面積下可以儲存的空間也會不同，雖然增加可以儲存的空間，但是錯誤率也會增加，彼此的關係如 Table 1 所示。

Type	SLC	MLC	TLC	QLC
Bits	1	2	3	4
Counts of R/W	SLC > MLC > TLC > QLC			
Speed of R/W	SLC > MLC > TLC > QLC			
Cost & Endurance	SLC > MLC > TLC > QLC			

Table 1. NAND Flash 儲存單元比較圖

NAND Flash 的結構由上至下分別由、Die、Plane、Block、Page 組成，以一個 8Gb 的 NAND Flash 為例，會有 1 個 Die，由兩個 Planes 組成，共 2048 個 Blocks 組成；一個 Block 由 256 個 Pages 組成，一個 Page 則是由 32 個 Sectors 組成，一個 Sectors 的大小為 512b。由於硬體設計的關係，讀寫得最小單位是 Page，擦除的最小單位則是 Block。然而每次寫入的大小有時候不會剛好等於一個 Page，因此未滿一個 Page 的資料會先放到 RAM 內等搜集到特定的大小後，再交由 Controller 去做處理。

陽明交大
NYCU

2.2.3 Flash Translation Layer

快閃記憶體轉換層（Flash Translation Layer, FTL），主要完成來自 Host 的 LBA 轉換成 PBA，也就是映射（Mapping）到正確的 Flash 上的空間位置，得到正確的位址後才去對 Flash 做寫入。由於 Flash 的物理特性，Block 必須先擦除才能夠寫入，不能夠直接做資料更新，同時也不能在同一個 Block 一直做擦除或寫入，不然會導致 Flash 的使用壽命減短。FTL 會維持一張 LBA 到 PBA 的映射表，除了紀錄映射關係之外，也會根據表格往新的 Block 去做寫入。有時候舊的數據已經不需要了，FTL 也可以根據

映射表來做垃圾回收（Garbage Collection, GC）；為了讓 Flash 的使用壽命更長，FTL 也會做磨損平衡（Wear Leveling）期望所有的 Block 都可以來攤提寫入成本。

除了寫入有限制以外，Flash 的讀取次數也是有限制的，讀取的次數太多會導致上面的數據發生錯誤，形成干擾（Read Disturb），因此 FTL 會設定一個閾值（Threshold），如果達到這個數值 FTL 就應該把這些數據移到較新的 Block，以保護資料不會流失。此外，隨著時間的流逝，Block 在自然或擦寫讀失敗時會損壞，對於壞掉的 Block 管理，也包含在 FTL 的範疇內。

2.2.4 Modern SSD Architecture

在大型數據中心內設計人員嘗試開發出一種適用於高容量 Host 的儲存設備，像是微軟和 Meta 的 OpenCompute（OCP），該架構為一個 Host 會連結 64 個 SSD，以 Figure 4 為例。現代 SSD 通常會具備 16 個或以上的閃存通道（Flash Channels），用來同時執行閃存陣列（Flash Array）的 I/O 指令，假設每個通道有 512 MBps 的帶寬（Bandwidth），具有 16 個通道的 SSD 內部總計有 8 GBps 的帶寬。由於跟 Host 端溝通的軟體介面加上硬體架構的關係，實際上 Host 外部沒辦法使用那麼高的帶寬，反而會下降到 1 GBps，也就是原本所有 64 個 SSD 內部 16 個通道總計應該要有 512 GBps 的內部帶寬，但實際外部能夠使用的帶寬只能達到 64 GBps。然而為了可以跟 Host 做溝通，再經過 PCIe 後，最後只能得到 32 GBps 的頻寬。

陽明交大
NYCU

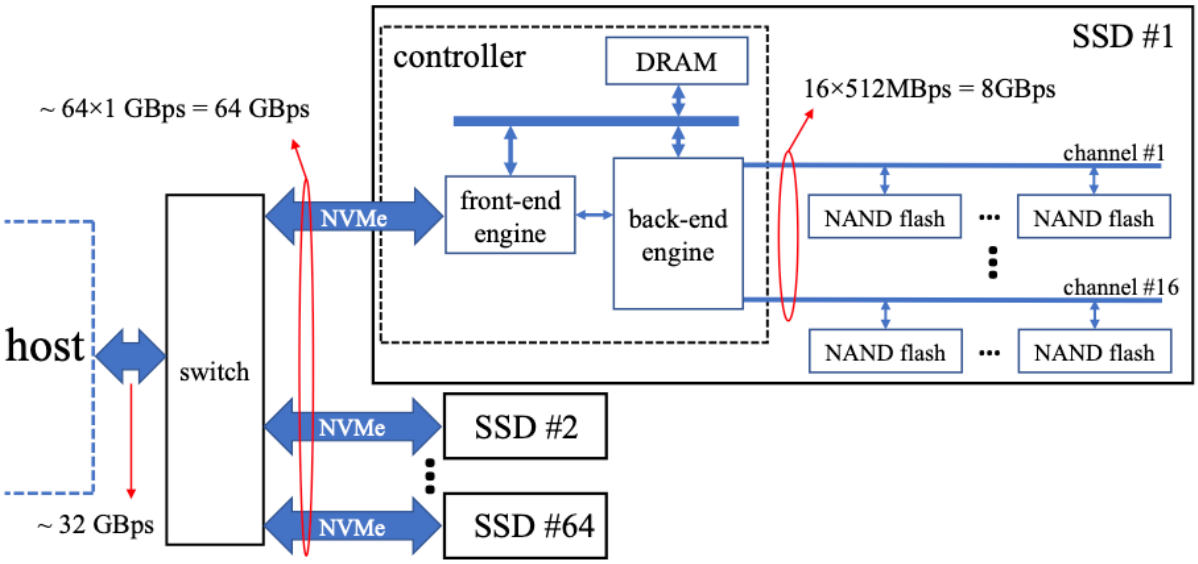


Figure 4. Modern SSD Architecture[17]

SSD 內部帶寬跟 Host 實際可以用的帶寬之前有 16 倍的差距，也就是主機讀取 32 Tb 的數據需要 16 分鐘，但 SSD 內部其實在 1 分鐘內就讀取完畢。因此引用 In-storage Processing (ISP)來解決過多的資料搬移，ISP 的概念就是接收到指令後，利用儲存裝置內部的處理器來運作使用者的程式，也就是不離開裝置在裝置內部就完成應用程式，藉此來減少無意義的資料移動。而 SSD 相較於傳統固態硬碟，不僅同時處理多個指令，一次能夠讀取的資料量也很大，如果需要 SSD 內部資料的運算如果部分能移到內部去做計算，不要拿到外部在做運算，就能大幅度的加速程式執行時間。

2.3 Extended Berkeley Packet Filter

Extended Berkeley Packet Filter (eBPF)[18]源自於 BPF，而 BPF 最初是用來過濾及監控封包，但之後被擴充成 eBPF，之後就變成 Linux 核心內建的內部行為分析工具。eBPF 可以在核心內部的沙盒（Sandbox）隔離的虛擬環境去運行程式，也就是在具有

陽明交大 NYCU

特權（Privileged）的作業系統內核下，以不改變源代碼或載入核心模組的前提下，安全且有效地擴增內核功能，其中包含動態追蹤、靜態追蹤和事件分析。

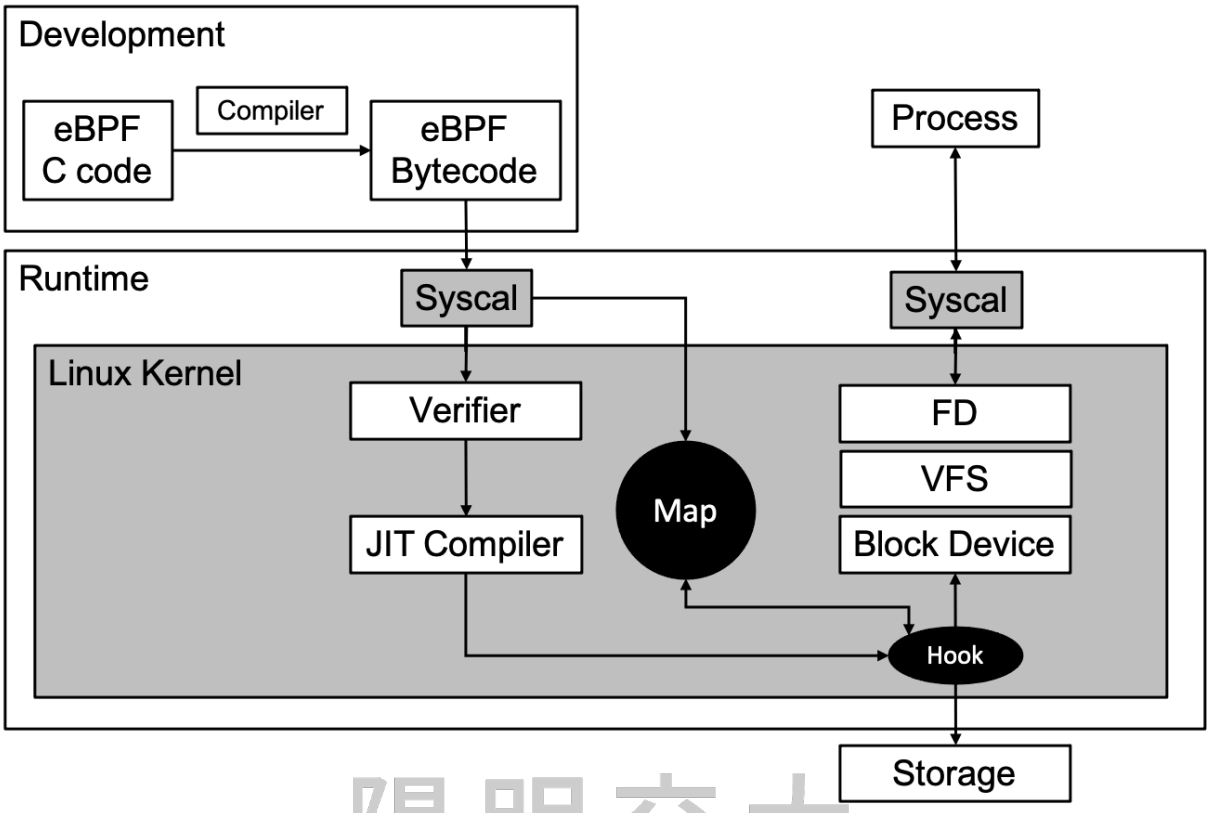
本研究需要分析應用程式將 SSD 的資料載入到記憶體時的行為，因此需要 eBPF 的協助，接下來會介紹其工作流程及合適的工具包。

2.3.1 eBPF Workflow

eBPF Program 都是事件觸發（Event-driven），當核心或應用程式通過特定的鉤點（Hook Point，以下簡稱 Hook），程式才抓得到資訊。有些已經定義好的 Hooks 像是 System call, Function entry/exit, Kernel tracepoints 等。如果想要的資訊無法透過已經定義好的 Hooks 觸發，是能夠建立一些探針（Probe）比如核心探針或使用者探針，來協助 eBPF 做監控。

陽明交大
NYCU

當目標 Hooks 被觸發時，就可以利用 BPF System call 將 eBPF Program 載入到 Linux 核心，但在將 Program 載入到 Linux 核心內前，必須經過兩個步驟：Verification 和 JIT Compilation，如 Figure 5 所示。第一步驗證（Verification），為了確保 eBPF Program 載入到核心可以安全運行的。首先，Process 載入 eBPF Program 時必須擁有特權指令；再來是 Process 不可以 crash 或以其他方式損害到系統；最後必須保證 Process 可以完成運行，也就是不能陷入無窮迴圈內。驗證成功後代表對系統不會有損害，再交給下一步即時處理（JIT Compilation），這步驟會將 eBPF Program 內部的 Generic Bytecode 轉成機器的特別指令集，以此來優化並加速運行。此舉可以幫助 eBPF Program 像本地編譯的核心代碼或加載核心模組一樣高效運行。



陽明交大
NYCU

Figure 5. eBPF Workflow

eBPF Program 還有一個重要的功能就是能夠共享搜集的資訊和儲存狀態的能力，因此在核心內部會有一個 eBPF Maps，可以儲存及檢索數據。如果要調用 eBPF Maps 內部的資訊可以透過 eBPF Program 來取得，也就是 User Space 應用程式內的 System call，不然一般程式是無法取得的。

2.3.2 BPF Compiler Collection

BPF Compiler Collection (BCC)[19]是一個框架，使用戶能夠編寫帶有 eBPF Program 的 Python Program。這個框架主要針對涉及利用 eBPF 來對應用程式或系統進行分析或追蹤的時候，其中 eBPF Program 可以用來搜集資料及定義相關事件，同時在

User Space 拿到的數據，BCC 也能夠將其轉成人類可讀的統計資訊。當我們運行帶有 eBPF Program 的 Python Program 時，他會生成 eBPF Bytecode 並將其載入到核心中。

在 BCC 公開的 github 中有提供許多的工具，其中我們想要了解從 SSD 讀取資料時，循序讀取跟隨機讀取對應的請求數及數據大小，因此基於已提供的工具做修改。

陽明交大
NYCU

Chapter 3

Characterization in DLRM

這個章節將會利用 DeepRecSys[15][20]分析 DLRM 模型[21][22]，找出目前 DLRM 有哪些瓶頸（Bottleneck）和特徵。針對各個運算子分別執行所需的時間佔比和記憶體使用量進行分析，最後會介紹最新研究都利用什麼方式去解決。

3.1 Breakdown Time on DLRM's Operators

分別對 RMC1~3 在 Batch Size=1 的前提下在 CPU 執行以及利用 GPU 加速後，比較各個運算子時間佔比的變化，如 Figure 6 及 Table 2 所示。

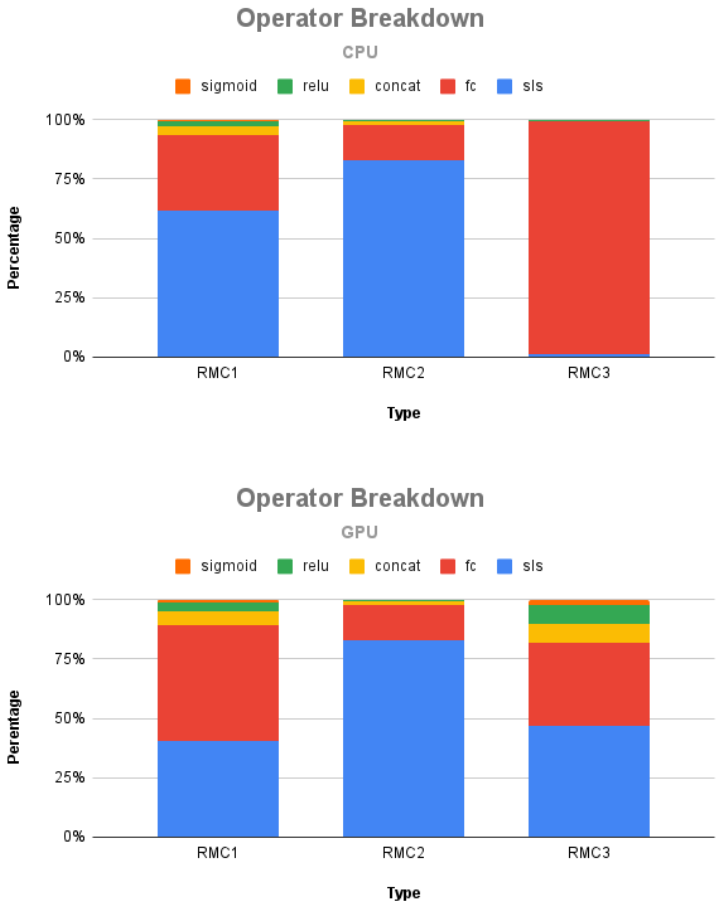


Figure 6. Operator Breakdown in CPU and GPU

	CPU			GPU		
	RMC1	RMC2	RMC3	RMC1	RMC2	RMC3
SLS	0.021983	0.15941	0.00679	0.009245	0.154389	0.169044
FC	0.011205	0.028887	0.519231	0.011178	0.028178	0.12555
Concat	0.001351	0.003176	0.001967	0.001287	0.003083	0.0288
Relu	0.000663	0.00078	0.001552	0.000855	0.000694	0.029338
Sigmoid	0.000312	0.00038	0.001115	0.000312	0.000364	0.007668

Table 2. Operator Breakdown in Time (ms)

不論是在 CPU 或在 GPU 上執行，明顯發現 SLS（藍色區塊）及 FC（紅色區塊）佔主要的時間比，首先觀察 RMC3 在 CPU 和 GPU 前後的變化，針對 SLS 明顯佔比提升（1.3% → 46.9%），而 FC 佔比明顯降低（97.8% → 34.8%），代表 FC 透過 GPU 高度平行運算的特性可以明顯被加速到，但是 SLS 時間佔比不降反升，代表 SLS 並不適用於 GPU 的平行運算，反而會為了平行而讓真實時間上升，如 Table X 所示；再來觀察 RMC1&2 的行為，經過 GPU 加速後 SLS 佔比（RMC1: 61.9% → 40.4%, RMC2: 82.8% → 82.7%）仍佔很大一部分的比例，而 FC 也沒有明顯減少（RMC1: 31.6% → 48.9%, RMC2: 15.0% → 15.1%）。

其中造成 RMC1~3 彼此 FC 佔比不同主要是因為各自在提取 Dense Inputs 時會有不同的 MLP 再加上預測用的 MLP 也有所不同，同理 SLS 的時間佔比也會因初始設定的不同而影響。綜合上述，可以推斷 SLS 就是 DLRM 模型在運行時的瓶頸，不僅利用 GPU 加速的成效有限，有可能為了要符合高度平行化，導致花費更高的成本。

陽明交大 NYCU

3.2 Breakdown Memory Size on DLRM's Operators

各個運算子的記憶體使用量也是我們在意的，如 Table 3 所示。

	RMC1	RMC2	RMC3
SLS	4096	4096	2560
FC	0.403332	1.27975	12.8216
Concat	0	0	0
Relu	0	0	0

Table 3. Memory usage of operators

由於 SLS 需要去 Embedding Tables 搜集指定的 Embedding 向量，模型在需要在記憶體內開一個很大的空間去暫存表格資料，但實際上會用到的 Embedding 向量又很少，造成記憶體的浪費，再加上前一個實驗，可以結論出 SLS 的計算量並不高，因為 GPU 的加速有限，因此 SLS 是屬於 Memory-bound 的運算子，而 FC 則是屬於 Computed-bound 的運算子。

陽明交大 NYCU

3.3 Current Solutions

目前最新的論文會將 SLS 搜尋 Embedding 向量的部分移到開放資源的 SSD 平台（OpenSSD），藉由在 OpenSSD 內部的控制器新增一些模組，來減少 I/O 存取的次數。像是 RecSSD[13]主要將 Embedding Tables 放在 SSD 內，搜尋的部分也是在 SSD 內部完成。為了加快內部的搜尋時間，RecSSD 在 OpenSSD 內部的 DRAM 新增一些模組，像是快取住近期有哪些請求，而這些請求又去哪些 Page 做 Flash 的讀取，同時也

會將已經讀取過的 Flash 內資料放入 Buffer 內，如果 Host 需要在相同的 Page 內讀取資料，就可以直接從 Buffer 拿，不需要重新到 Flash 內部搜尋，以此來減少 I/O 次數。

除了 RecSSD 的軟體解法，另一篇論文提出軟硬體解法－FlashEmbedding[12]，該論文提出 Embedding Vector SSD (EV-SSD)的架構，有別於傳統的 SSD 他在控制器內部新增了一個介面讓 Host 可以直接請求特定的 Embedding 向量而且透過 DMA 來傳輸資料。此外為了解決讀取放大的問題，需要經過篩選來減少不必要的讀取。比如說一個請求進來，EV-SSD 會先解析請求的 LBA，經反覆的比較，來決定 Embedding 向量的起始 LBA，之後才會去 Flash 讀取資料。然而搜尋到的 Embedding 向量不會馬上傳給 Host，會先放入到一個 Buffer 內，等確定所有要被 Lookup 的 Embedding 向量搜集完成，才會透過 DMA 傳輸給 Host，透過軟體的輔助來減少無意義數據的傳輸。相對於單純將 Embedding Tables 放入 SSD，這兩種方法 end-to-end 的延遲都能降低 2 倍多。

陽明交大
NYCU

綜觀上述的方法，都是想要解決 SLS 內搜集 Embedding 向量的問題，共同點是將 Embedding Tables 放入 SSD 內部，利用軟體或硬體的方式在 SSD 內部就完成搜尋的步驟，並在 SSD 內部 DRAM 加入一些快取來記錄重複的部分，進而減少不必要的 I/O 以及資料的傳輸。然而上述的實作都是在 FPGA 板上，可修改性很高，相對於市售的 SSD 上參考價值較低，因此本研究目的在 Embedding Tables 放在市售 SSD 的前提下，針對 SLS 的瓶頸，找出合適的 I/O 方式讀取所需的 Embedding 向量，同時又不影響執行時間，以提供大型工作站在深度學習硬體成本考量及運算速度的取捨。

Chapter 4

Methodology

這個章節將會介紹我們的實驗流程及實作。已知 Sparse Lengths Sum (SLS)目前是 DLRM 最大的瓶頸，不僅用 GPU 加速沒有明顯成效又耗費大量記憶體空間，而且實際需要的運算又低。我們打算將全部 Embedding Tables 放入 SSD，但是這麼做一定會有搬運資料的高成本，因此設計一系列的實驗來找出合適的搬運方法。

4.1 Problem Statement

當 DLRM 內的 Embedding Tables 存放在 SSD 時，如果要執行 SLS 運算，必須從 SSD 搜集特定的 Embedding Vectors，而目前常見的 Linux I/O 方式有 Buffered I/O、Unbuffered I/O 及 mmap，比較用這些 I/O 方式來實作的 SLS 有哪些益處。

在 SSD 中循序讀取跟隨機讀取差了幾十倍，因此利用 BCC Library 來分析監控到的封包，以此來辨別循序讀取跟隨機讀取各自指令數量及資料大小，綜合這些特徵篩選出合適的 I/O 方式。

Embedding Tables 的大小會隨著設定不同而改變，但彼此之間有共同特點，我們逐一去做調整，最後找出整體需要看的 Embedding 向量數才是關鍵，跟 Embedding Table 的大小無關。最後組合不同 I/O 方式，讓 SLS 達到最少記憶體用量及花費時間。

陽明交大
NYCU

4.2 Experiment setup

實驗環境：

- 硬體：
  - CPU: Intel ® Core ™ i7-8700 CPU @ 3.2 GHz
  - Main memory: 62.7 GB
  - SSD: SAMSUNG 970 EVO Plus M.2 SSD @ 500 GB
    - PCIe Gen 3.0 * 4, ~4 GB/s

Type	Sequential	Random
Read	~3500 MB/s	~19000 IOPs, 4KB, QD=1
Write	~3200 MB/s	~60000 IOPs, 4KB, QD=1

Table 4. SSD 最高讀寫速度

陽明交大
NYCU

- 軟體：
  - OS: Ubuntu 20.04.1 LTS
  - Docker: 1.10.1-pytorch + 1.11.0-cuda11.3-cudnn8-devel
  - Others: llvm-7, v0.24.0-bcc

目前 DLRM 常見的模型有 RMC1~3，其中 Embedding Tables 的設定如 Table 5：

Type	RMC1	RMC2	RMC3
Embedding size	4000000	500000	2000000
Feature size	32	64	32
Table count	8	32	10
Total size	7.7Gb	7.7Gb	8.6Gb
Indices per lookup	80	120	20

Table 5. DLRM 的 Embedding Tables 設定

本研究中用到的 I/O 存取方式如 Table 6 所示：

Cmd	fseek/fread	lseek/read	mmap	read	(1)	(2)
Alias	io_buf	io_unbuf	mmap	ram	ratio	opt

Table 6. 不同的 I/O 存取方式

io_buf 是利用 C 標準函式庫的 fseek/fread 來存取 Embedding 向量；io_unbuf 則是利用 Linux 的 lseek/read 來存取 Embedding 向量；mmap 則是用 Linux 的 mmap 來存取 Embedding 向量；ram 則是將全部的 Embedding 表格都放入記憶體來模擬真實情況，同時作為本研究的 Baseline。

ratio 會先從 SSD 拿好一定比例的 Embedding Tables 放到記憶體內，之後再從 SSD 存取剩下需要的 Embedding 向量，Table 6 (1)；opt 會先分析 Embedding 向量在 Embedding Tables 的分佈，然後先拿頻率最常出現區間的 Embedding Tables 放到記憶體內，之後再從 SSD 存取剩下需要 Embedding 向量，Table 6 (2)。

陽明交大 NYCU

本研究只會跑 RMC1 跟 RMC2，因為這兩個模型受 SLS 的影響最大，同時每個數據都是跑 5 次後再取平均，而且每次都會執行[23]

```
echo 3 | sudo tee /proc/sys/vm/drop_caches > /dev/null
```

來清掉 Cache 來保證實驗不因快取住而有不正確的結果，而且如果不清除快取，由於沒有新的事件觸發，會導致 eBPF 沒辦法抓取對應的封包。

4.3 Experiment tools and implementation

由於目前最新的 BCC 版本在某些情況下會有問題，本研究採用分支 v0.24.0 搭配 llvm-7 去編譯，並新增 biopattern.py 的功能。其中 biopattern.py 的功能可以定時去抓取系統讀取 SSD 的請求及數據大小[24]，如 Figure 7 所示。

```
struct counter {
    u64 last_sector;
    u64 seq_bytes;
    u32 sequential;
    u64 rdm_bytes;
    u32 random;
    char name[TASK_COMM_LEN];
    u32 pid;
};
```

```
if (counterp->last_sector) {
    if (counterp->last_sector == sector) {
        __sync_fetch_and_add(&counterp->sequential, 1);
        __sync_fetch_and_add(&counterp->seq_bytes, nr_sector * 512);
    } else {
        __sync_fetch_and_add(&counterp->random, 1);
        __sync_fetch_and_add(&counterp->rdm_bytes, nr_sector * 512);
    }
}
```

```
nctu@[lab136] in ~/bcc/tools on (v0.24.0)xxx
(/°Д°)/ sudo python3 biopattern.py 1 500
PID    CMD                TIME      DISK      RND   SEQ    COUNT  RND_BYTES  SEQ_BYTES
1704680 bench              13:12:07  nvme0n1   206   784      990    5898240 1023148032
1704680 bench              13:12:08  nvme0n1     2   781      783    2359296 1021640704
1704680 bench              13:12:09  nvme0n1     1   731      732     262144  957874176
1704680 bench              13:12:10  nvme0n1     1   548      549    1310720  715980800
1704680 bench              13:12:11  nvme0n1     3   546      549    3145728  712835072
1704680 bench              13:12:12  nvme0n1     7   538      545    2019328  702357504
```

Figure 7. biopattern.py 的資料結構、部分程式碼及運行結果

接下來會介紹不同 I/O 方式的 SLS，首先定義一個資料結構，用來設定 Embedding Tables 的大小、實驗執行的 Batch Size 以及 Lookup 的 ID 產生器，如 Figure 8 所示。

```
struct sls_config {
        std::string table;              // filename
        u32 emb_row;                    // R, embedding-size
        u32 emb_col;                    // C, feature-size
        u32 lengths;                    // K, lengths-count, batch-size
        u32 lengths_size;               // L, num-indices-per-lookup
        std::vector<u32> ids;    // ids-list
```

Figure 8. SLS 初始參數的資料結構

根據 Table 6 的設定，io_buf, io_unbuf, mmap 都是屬於拿取特定的 Embedding 向量，由於 Linux 內部實作細節會有不同的結果，之後會在 Evaluation 做比較；ram 則是將整個 Embedding Tables 搬到 DRAM 內，主要是模擬目前 SLS 在 DRAM 的行為，同時也作為實驗的基準線；剩下的 ratio 及 opt 則是根據實驗設計，先拿部分的 Embedding Tables 到 DRAM，如果先拿的部分有命中，那直接從記憶體拿資料，如果沒有命中，再利用 Unbuffered I/O 拿特定的 Embedding 向量。

本研究實做的 SLS 都有去跟 caffe2[25]內建的 SLS 的結果去做比對，確保正確性。此外，還有提供 Benchmark 可以測試自定義的參數，不受限於 RMC1 跟 RMC2；同時提供測試各種 I/O 方式 SLS，根據不同情境找到適合的 SLS，如 Figure 9 所示。

```
./each --dir="~/dlrm-file/dlrm/table_rm1" \
--embedding-size="4000000" \
--feature-size="32" \
--num-indices-per-lookup="80" \
--type="ram"
```

Figure 9. 特定 SLS 執行指令的範例

陽明交大 NYCU

Chapter 5

Evaluation

為了評估各種 I/O 方式在 SLS 的表現，會先考慮循序隨機讀取時的請求數量及各自的資料量，由於 DLRM 的初始設定會影響 Embedding Tables 的大小，接下來會找出主要影響的參數，最後根據實驗的資訊，整理出最適合 DLRM 模型的 I/O 方式。

5.1 Requests profiling from different I/O in DLRM

首先我們去觀察 io_buf, io_unbuf, mmap, ram 對 SSD 讀取數據時的指令分配（Figure 10、11）。ram 是將全部的 Embedding Tables 都讀取到記憶體內，當初將表格寫入到 SSD 時是採用循序寫入的方式，因此 ram 大部分都是循序讀取；io_buf 是用 fseek/fread 去存取 Embedding 向量，由於 C 標準函式庫在使用 FILE*時都會在記憶體建立一個 Buffer，而 fread 會讀取一段資料到 Buffer 內，因此 io_buf 有較 ram 以外高的循序讀取；io_unbuf 屬於 Direct I/O，由於沒有 Buffer 的設計，因此大部分都是屬於隨機讀取；mmap 可以直接對 Page Cache 進行操作，直接透過 pointer 讀寫 Page Cache，減少系統呼叫與資料的複製，大部分屬於隨機讀取。

在 RMC1 跟 RMC2 的條件下，這 4 種方式請求的分佈都差不多，但觀察實際請求數（Table 7、8），以隨機讀取為主的方式會有較多的請求數，主要來自於隨機讀取一次的資料量會小於循序讀取的資料量，為了讀取相同資料量的 Embedding 向量，因此就會花費大量的隨機讀取請求，而優點就是讀取的資料都是確定需要的。

陽明交大 NYCU

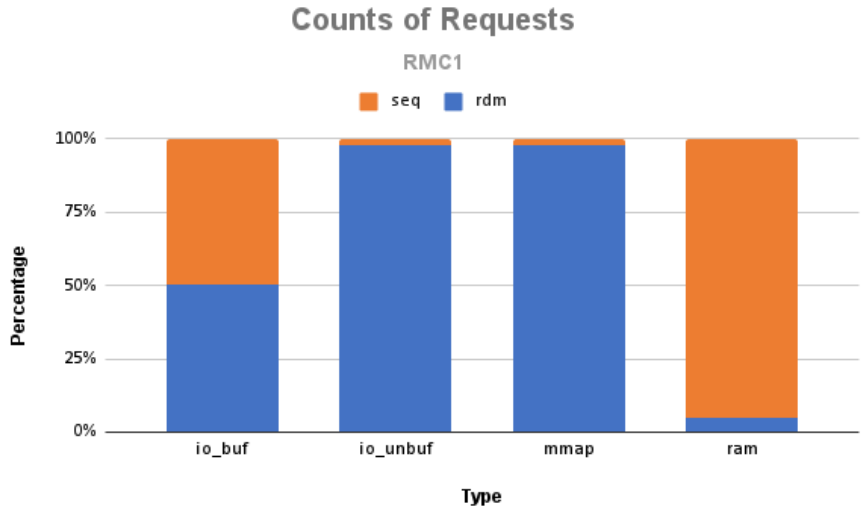


Figure 10. RMC1 請求分佈

	io_buf	io_unbuf	mmap	ram
rdm	1015.40	783.40	782.60	326.20
seq	1002.00	17.20	16.80	6257.00

陽明交大

Table 7. RMC1 實際請求數

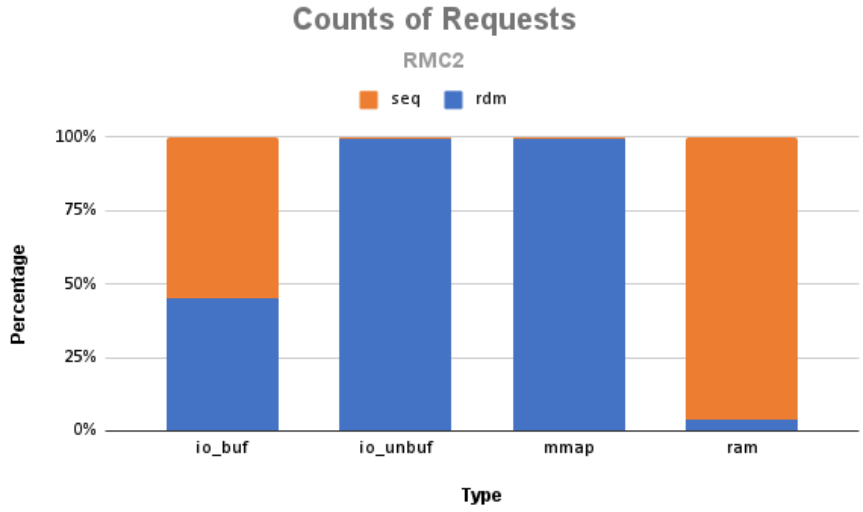


Figure 11. RMC2 請求分佈

	io_buf	io_unbuf	mmap	ram
rdm	4763.20	3962.40	3883.80	255.00
seq	5830.00	16.20	21.80	6249.60

Table 8. RMC2 實際請求數

接下來，我們去觀察 4 種 I/O 方式在 RMC1 和 RMC2 的循序/隨機讀取個別對應的資料量（Table 9、10）。由於 ram 會讀取全部的 Embedding Tables 相較於其他方式過於顯著，這樣會無法看出其他三種的情況，因此不放入圖中（Figure 11、12）；io_buf 會 Buffer 一些資料的關係因此相較於 io_unbuf 有更多的讀取量；mmap 在隨機讀取上多出很多資料量，主要是因為 mmap 是先建立 Process 上 Virtual Address 與 Physical Address 的對應關係，當要載入資料時根據對應關係去 Page Cache 拿，如果找不到，會發生 Page Fault 再從 SSD 讀取資料到 Page Cache 及建立新的對應關係，然而需要 Lookup 的 Embedding 向量非常分散，會一直發生 Page Fault 其中可能會多拿一些不必要的資料到 Page Cache，因此才會在隨機讀取中多拿那麼多資料。

陽明交大
NYCU

除了 ram 以外，剩下三種的方式都有讀放大的問題存在，像是 RMC1 總共需要 8*80*32*8byte 的資料，大約 0.15Mb 左右，但根據 Table X，讀取到的大小遠大於需要的大小，主要因為讀的最小單位就是 Page，才導致讀過多不必要的資料。

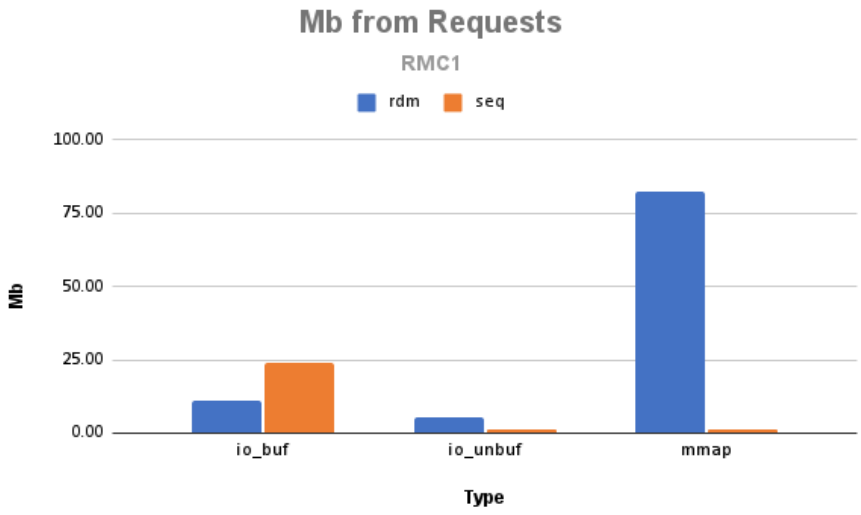
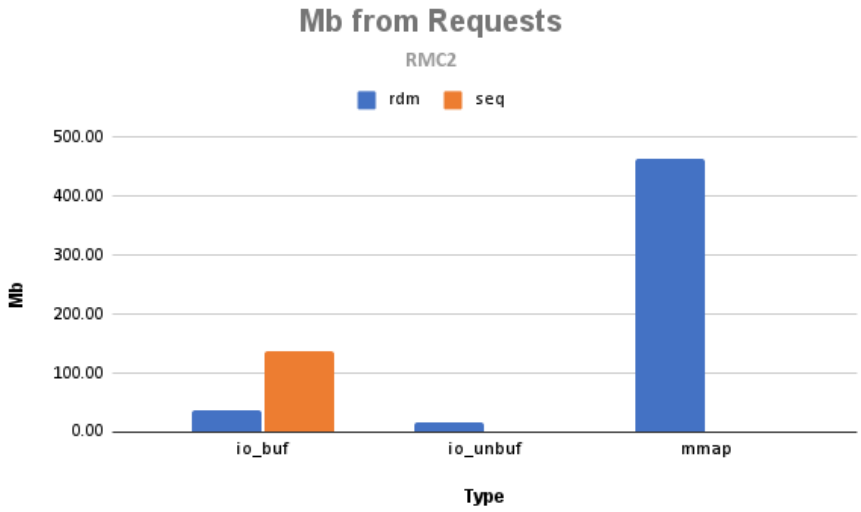


Figure 12. RMC1 請求對應資料大小

	io_buf	io_unbuf	mmap	ram
rdm	11.03	5.34	82.41	37.98
seq	24.09	1.25	1.15	7778.10

Table 9. RMC1 請求對應資料大小數據（Mb）



陽明交大

Figure 13. RMC2 請求對應資料大小

	io_buf	io_unbuf	mmap	ram
rdm	37.64	17.68	464.55	50.88
seq	136.55	1.11	1.59	7756.02

NYCU

Table 10. RMC2 請求對應資料大小數據（Mb）

5.2 Latency profiling from different I/O in DLRM

接下來我們測試 4 種 I/O 方式和不同 Batch Size 下實際在 RMC1 和 RMC2 的表現，如 Figure 14 所示。由於 ram 將全部的 Embedding Tables 放到記憶體內，運算也都是在記憶體內完成，因此花費的時間都在搬運資料跟 Batch Size 大小無關；io_buf 跟 mmap 都會多拿一些資料，然而多拿的部分並不是我們的目標，多耗費了一些時間在搬運不需要的資料。RMC1 跟 RMC2 的趨勢基本上雷同，ram > io_buf > mmap > io_unbuf 然而 RMC2 的 ram 在 Batch Size=32 之後花費的時間最少，代表與其慢慢去 SSD 拿特

定的 Embedding 向量，還不如講全部的表格搬到記憶體內，花費的時間還比較少。之後的實驗就是要找出到底是哪些參數在影響搜尋 Embedding 向量的速度。

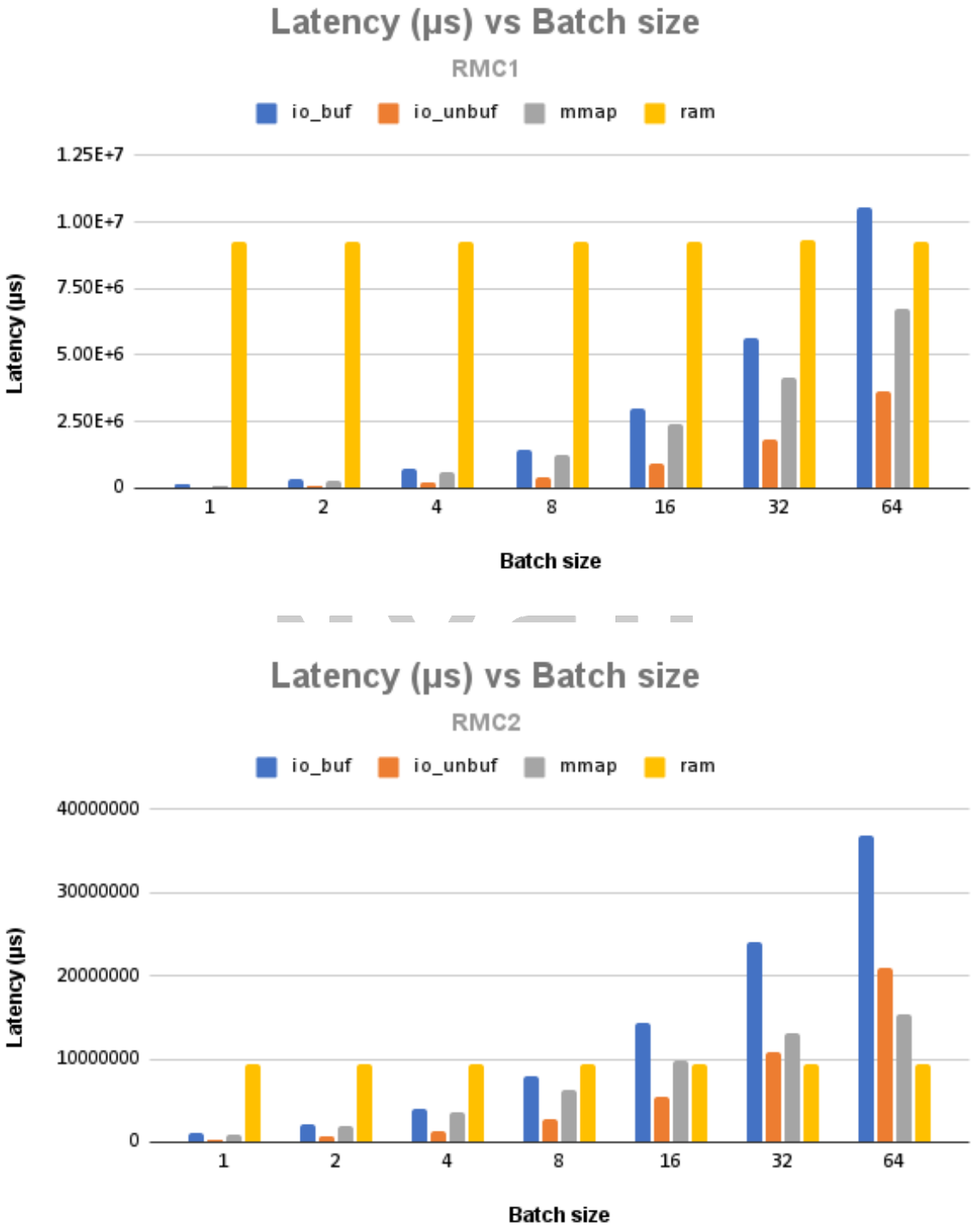


Figure 14. RMC1 跟 RMC2 在不同 Batch Size 下不同 I/O 方式花費的時間

5.3 Overhead profiling in DLRM by arguments tuning

一開始認為是 Embedding Tables 的大小在影響，基於 RMC2 單個表格等比例放大，其他條件不變，發現 io_buf, io_unbuf, mmap 的時間不因表格變大而增加，如 Figure 15。接下來分別去增加 Batch Size 大小及 Lookup Size 大小，其他條件維持原本 RMC2 的設定，觀察到改變這兩個因素趨勢基本一致，Latency ∝ Batch Size*Lookup Size，也就是跟單張表格內 Lookup 的總量有關，如 Figure 16。

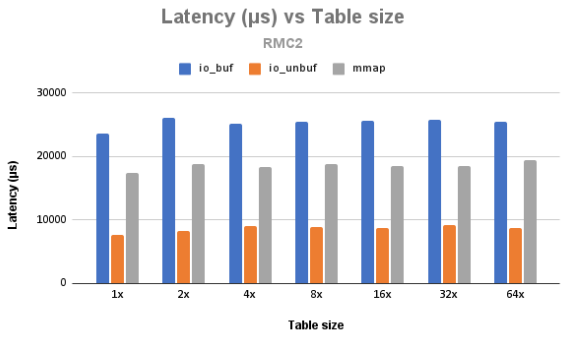


Figure 15. Latency 與表格等比放大關係圖

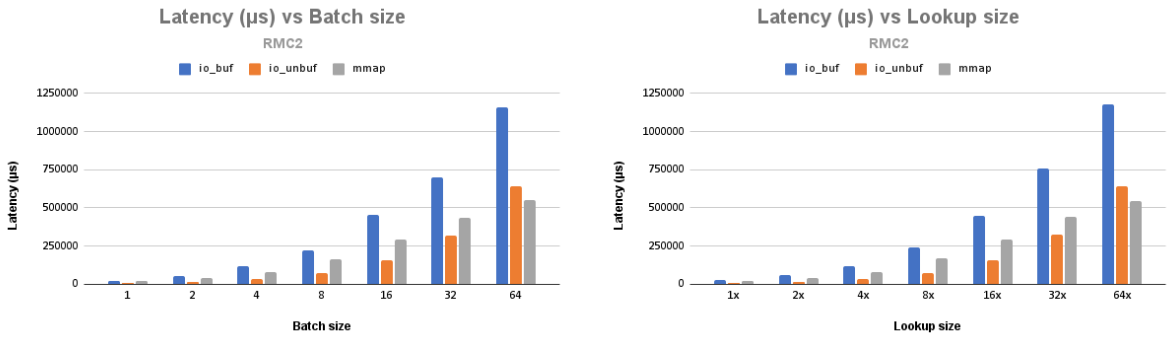


Figure 16. Latency 與 Batch Size、Lookup Size 關係圖

5.4 Tradeoff between memory and I/O in DLRM

既然知道影響 Latency 的主因是單張 Lookup 的所需的 Embedding Vectors 總量，基於這個結果，為了解決當 Batch Size 到一定值後，除了 ram 以外的方法，其他三種

的 Latency 都會急劇上升的狀況，因此提出了 ratio 及 opt 的方法。這兩種方法主要是混合 ram 及 io_unbuf 的優勢，細節在 Methodology 有做說明。前面的實驗加上這次的 ID 產生器都是均勻分佈（Uniform），從 Figure 17 觀察到 ratio 跟 opt 的結果會介於 ram 跟 io_unbuf 之間，RMC1 在 Batch Size=128 後、RMC2 在 Batch Size=32 後這兩種方法在特定情況下還是沒有比 ram 好，但有幅度的減緩 io_unbuf 劇增的情形。

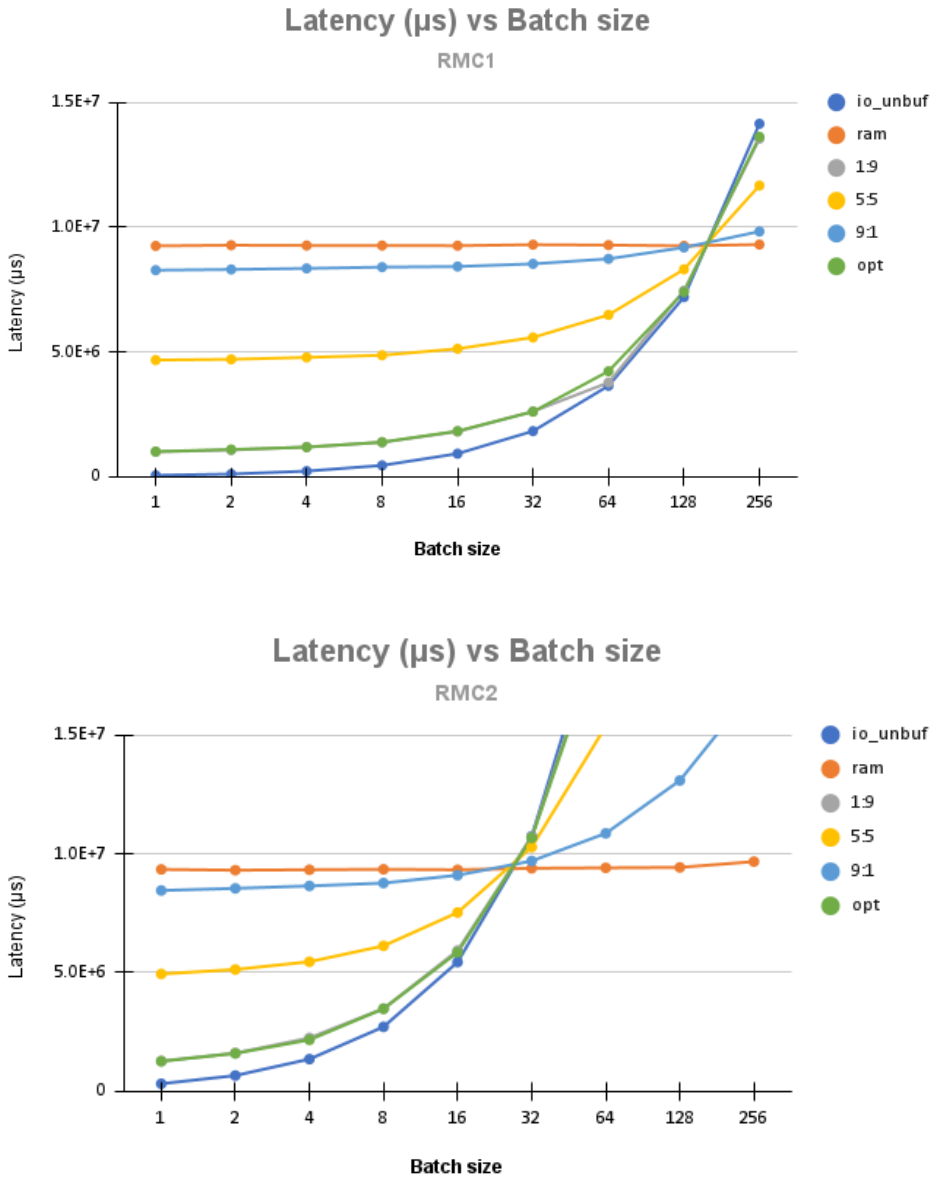


Figure 17. ratio 和 opt 在 RMC1 跟 RMC2 的表現（ID 分佈為 Uniform）

從上述的實驗可以得知，先將一部分的 Embedding Tables 放入記憶體內，可以有效減少 io_unbuf 在 Batch Size 大時的缺點。從 ratio 9:1 跟 ratio 1:9 可以看出，ID 的分佈也會影響，因為在 1:9 的時候放到記憶體的部分 Embedding Tables 沒辦法命中到很多個，但在 9:1 已經有大量的 Embedding Tables 被放入記憶體，幾乎都會命中，在 RMC1 可以突破 Batch Size=128 的瓶頸。接下來，我們把 ID 改成雙峰的 Binomial 分佈，以模擬每個人在特定種類的偏好。從實驗結果（Figure 18、19）發現，opt 在 RMC1 當 Batch Size=128 時有大幅度的改善，也能應付 Batch Size=256 的狀況；opt 在 RMC2 當 Batch Size=32 相較於前面的實驗有所突破，這兩個案例證實 ID 的分佈以及預載入到記憶體的部分 Embedding Tables 的命中率會影響 ratio 及 opt 的執行時間。

隱留必究

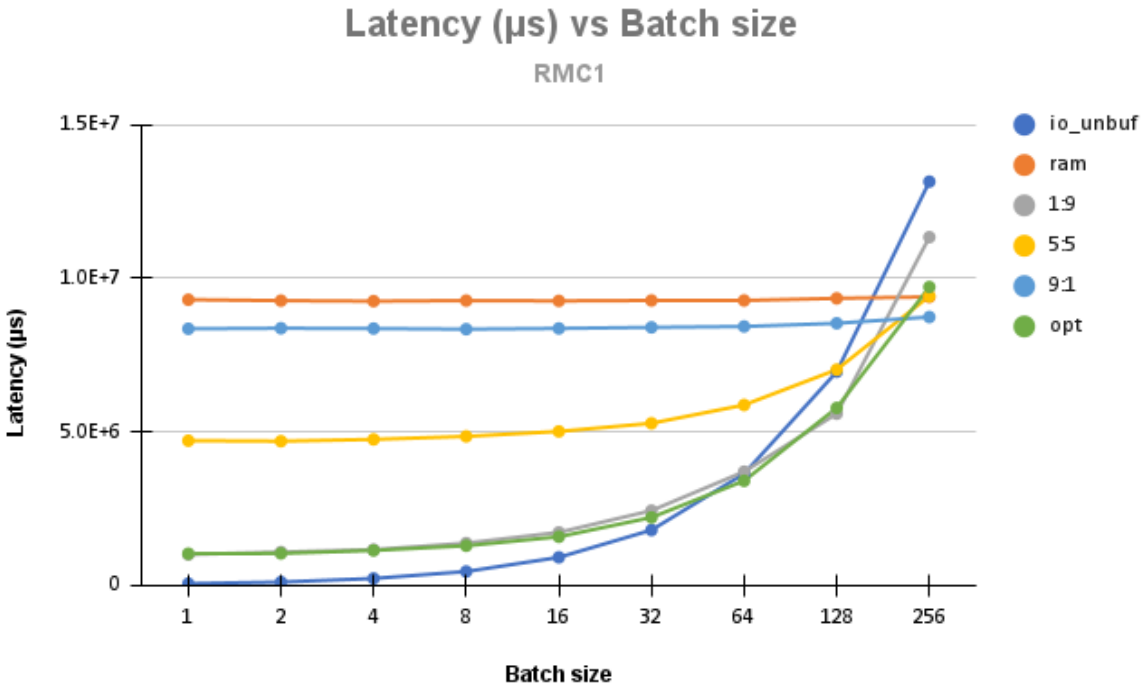


Figure 18. ratio 和 opt 在 RMC1 的表現（ID 分佈為 Binomial）

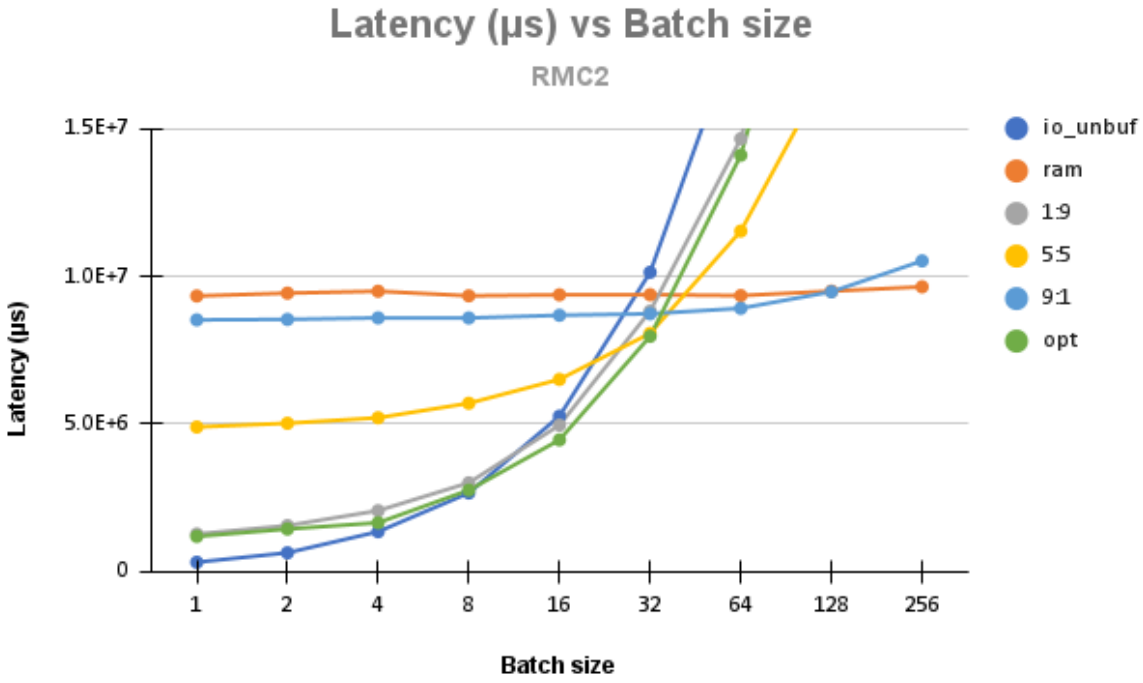


Figure 19. ratio 和 opt 在 RMC2 的表現（ID 分佈為 Binomial）

陽明交大
NYCU

同樣的，我們去分析 ratio 及 opt 各自對應的請求分佈跟數據量，其中 RMC1 的 Batch Size=128，RMC2 的 Batch Size=32。雖然這兩種方法會預載入一部分的 Embedding Tables 但大部分還是以隨機讀取為主，如 Figure 20 所示。

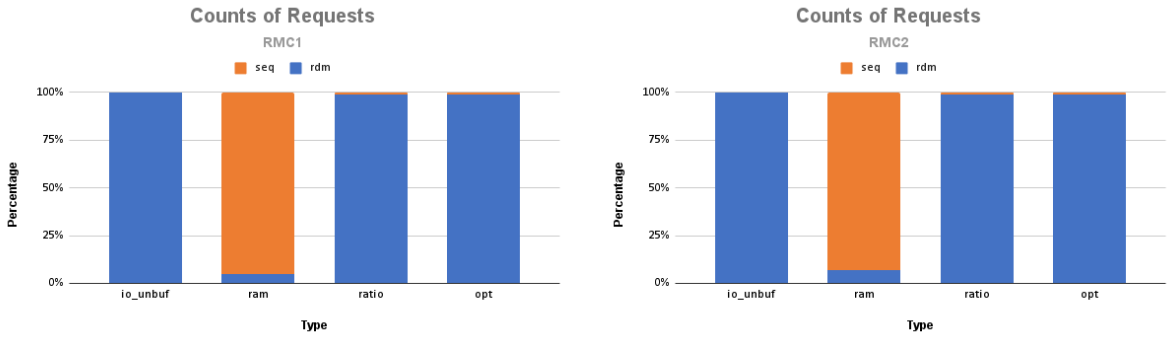


Figure 20. RMC1 跟 RMC2 請求分佈

RMC1				
	io_unbuf	ram	ratio	opt
rdm	76886.8	316.6	63090.8	52413.8
seq	17.8	6252.6	737	720.2

RMC2				
	io_unbuf	ram	ratio	opt
rdm	112523.0	464.6	84129.6	73619.6
seq	22.2	6222.8	845.8	842.0

Table 11. RMC1 跟 RMC2 請求實際數量

從實際的請求數量去分析，io_unbuf 在 Batch Size 大的時候，會大量使用隨機讀取，可以解釋 io_unbuf 表現不好的原因；而 ratio 跟 opt 都表明預載入 Embedding Tables 的優點，在 RMC1 中，相較於 io_unbuf 增加約 700 的循序讀取，卻可以減少最多 25000 個隨機讀取；在 RMC2 中，相較於 io_unbuf 增加約 800 的循序讀取，卻可以減少最多 40000 個隨機讀取，如 Table 11 所示。

接下來我們觀察請求對應的資料大小，ratio 跟 opt 相較於 ram 只需要約 1/8 的 ram 的記憶體使用量，就可以達到比 ram 快一點的效果，不過還是有讀放大的問題，如 Figure 21 跟 Table 12 所示。

陽明交大
NYCU

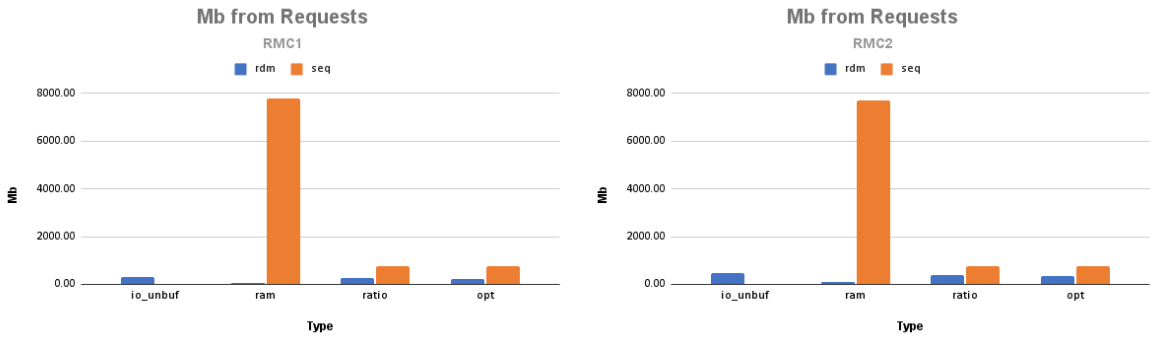


Figure 21. RMC1 跟 RMC2 請求對應資料大小

RMC1				
	io_unbuf	ram	ratio	opt
rdm	309.86	45.37	271.85	229.05
seq	0.89	7769.92	775.07	774.35

RMC2				
	io_unbuf	ram	ratio	opt
rdm	459.74	101.57	386.60	343.53
seq	1.04	7706.19	776.77	773.50

Table 12. RMC1 跟 RMC2 請求對應資料大小數據（Mb）

綜合上述，將 Embedding Tables 放在 SSD 之後用不同的 I/O 方式去讀取，會根據 DLRM 的設定不同而有不同的瓶頸，在 RMC1 如果 Lookup 總量超過 81920（大約 0.256%），在 RMC2 如果 Lookup 總量超過 122280（大約 0.768%），從 SSD 讀取資料的效益就會大幅降低，但是可以透過預載入一部分的 Embedding Tables 到記憶體內，剩下的再用 I/O 去讀取，如果命中數夠高，就能有效地改善，也就是同時有 io_unbuf 及 ram 的優勢。

陽明交大
NYCU

Chapter 6

Conclusions and Future Work

本研究分析推薦系統運算子常見的 I/O 行為，包含循序或隨機讀取的指令數以及數據量，找出造成瓶頸的參數，並提供程式讓使用者可以測試不同 I/O 對於自定義的 DLRM 模型之 SLS 的分析。之後提出預載入部分的嵌入表，來改善嵌入表放入 SSD 延遲的結果，其中最高有 1.67 倍的加速，且只使用 1/8 原本記憶體使用量。在推薦系統的應用中，我們評估了嵌入表放在 SSD 的可行性，提供資料中心對於硬體成本的考量，以及在特定條件下的加速方式。

在軟體端我們完整分析 SLS 不同 I/O 的情形，也提出不同條件下的解決方法，然而極端狀況下，還是效益有限。目前最大的瓶頸還是從 SSD 搬運數據到外部記憶體，由於 SLS 的運算密度很低，如果可以在 SSD 內部就完成 SLS 的運算，並快取住附近的嵌入表，最後只傳出特徵結果，應該能大幅度的加速推薦系統的時間。

陽明交大
NYCU

References

1. Gupta, U., et al. *The architectural implications of facebook's dnn-based personalized recommendation*. in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2020. IEEE.
2. Zhao, Z., et al. *Recommending what video to watch next: a multitask ranking system*. in *Proceedings of the 13th ACM Conference on Recommender Systems*. 2019.
3. Naumov, M., et al., *Deep learning recommendation model for personalization and recommendation systems.* arXiv preprint arXiv:1906.00091, 2019.
4. Cheng, H.-T., et al. *Wide & deep learning for recommender systems*. in *Proceedings of the 1st workshop on deep learning for recommender systems*. 2016.
5. Zhou, G., et al. *Deep interest evolution network for click-through rate prediction*. in *Proceedings of the AAAI conference on artificial intelligence*. 2019.
6. Zhou, G., et al. *Deep interest network for click-through rate prediction*. in *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*. 2018.
7. Park, J., et al., *Deep learning inference in facebook data centers: Characterization, performance optimizations and hardware implications.* arXiv preprint arXiv:1811.09886, 2018.
8. Hazelwood, K., et al. *Applied machine learning at facebook: A datacenter infrastructure perspective*. in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2018. IEEE.
9. Naumov, M., et al., *Deep learning training in facebook data centers: Design of scale-up and scale-out systems.* arXiv preprint arXiv:2003.09518, 2020.
10. Eisenman, A., et al., *Bandana: Using non-volatile memory for storing deep learning models.* Proceedings of Machine Learning and Systems, 2019. **1**: p. 40-52.
11. Sun, X., et al. *RM-SSD: In-Storage Computing for Large-Scale Recommendation Inference*. in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 2022. IEEE.
12. Wan, H., et al. *FlashEmbedding: storing embedding tables in SSD for large-scale recommender systems*. in *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*. 2021.
13. Wilkening, M., et al. *RecSSD: near data processing for solid state drive based recommendation inference*. in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 2021.
14. Hsia, S., et al. *Cross-stack workload characterization of deep recommendation systems*. in *2020 IEEE International Symposium on Workload Characterization (IISWC)*. 2020. IEEE.

陽明交大 NYCU

15. Gupta, U., et al. *Deeprecsys: A system for optimizing end-to-end at-scale neural recommendation inference*. in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 2020. IEEE.
16. Andersen, D.G. and S. Swanson, *Rethinking flash in the data center.* IEEE micro, 2010. **30**(04): p. 52-54.
17. Torabzadehkashi, M., et al., *Computational storage: an efficient and scalable platform for big data and hpc applications.* Journal of Big Data, 2019. **6**(1): p. 1-29.
18. ebpf.io. (2022, Jul 12). eBPF - Introduction, Tutorials & Community Resources [Online]. Available: https://ebpf.io
19. iovisor. (2022, Jul 12). BPF Compiler Collection (BCC) [Online]. Avaiable: https://github.com/iovisor/bcc
20. harvard-acc. (2022, Jul 12). DeepRecSys: A System for Optimizing End-To-End At-scale Neural Recommendation Inference [Online]. Available: https://github.com/harvard-acc/DeepRecSys
21. facebookresearch. (2022, Jul 12). Deep Learning Recommendation Model for Personalization and Recommendation Systems [Online]. Available: https://github.com/facebookresearch/dlrm
22. seemethere. (2022, Jul 12). pytorch/pytorch:1.11.0-cuda11.3-cudnn8-devel [Online]. Available: https://hub.docker.com/layers/pytorch/pytorch/pytorch/1.11.0-cuda11.3-cudnn8-devel/images/sha256-9bfcfa72b6b244c1fbfa24864eec97fb29cfafc065999e9a9ba913fa1e690a02?context=explore
23. cclin0816. (2022, Jul 12). Benchmark [Online]. Available: https://github.com/cclin0816/benchmark
24. iovisor. (2022, Jul 12). bcc Reference Guide [Online]. Available: https://github.com/iovisor/bcc/blob/master/docs/reference_guide.md#1-get_table
25. caffe2. (2022, Jul 12). Operators Catalog | Caffe2 [Online]. Available: https://caffe2.ai/docs/operators-catalogue.html#sparselengthssum

陽明交大 NYCU