國立清華大學資訊工程系114學年度上學期專題報告

專題名稱	FPGA-Based Acceleration of TinyLLaVA Model Inference via High-Level Synthesis on Alveo U280				
參加競賽或計畫	☐ 參加對外競賽	☐ 參與其他計畫	■無參加對外競賽或任何計畫		
學號	111060013	111060016	111060019		
姓名	劉祐廷	林芷韻	黃子恩		

摘要

本專題旨在探討如何將多模態大型語言模型 TinyLLaVA-Phi-2-SigLIP-3.1B 的推論流程，透過 High-Level Synthesis (HLS) 技術部署至 AMD Alveo U280 FPGA 平台，以實現模型推論加速。由於此模型運算涉及大量矩陣乘法與權重存取，本研究特別著重於資料路徑設計與記憶體存取策略的最佳化，以降低 I/O 瓶頸並提升資源利用率。在實作方面，我們完成了 SigLIP、Connector 與 Phi-2 三個模組的硬體合成，並驗證其可分別於 U280 上正確運行。針對記憶體部分，透過將權重合理分散至 HBM 不同 bank，搭配 AXI Burst 傳輸 與 tiling 計算策略，成功達成資料平行化與吞吐量提升。在 SigLIP 模型的 Attention 模組中，為避免除法與取餘數運算造成的邏輯資源浪費，我們改以 counter-based index 計算法 取代原始除餘數運算，大幅降低資源使用並提升效率。在 Phi-2 模組中，我們進一步利用權重矩陣旋轉技巧，降低 URAM 的緩衝區需求，並將原本的 float16 運算改為定點數 (fixed<40,16>)，使整體運算速度明顯提升。綜合結果顯示，本研究提出的設計方法能有效縮短推論時間並提升 FPGA 資源使用效率，展現出以 HLS 為基礎進行多模態大型語言模型硬體加速的可行性與潛力。

中華民國 114 年 11 月

研究摘要與成果

本專題旨在探討如何將多模態大型語言模型 **TinyLLaVA-Phi-2-SigLIP-3.1B** 的推理透過 **High-Level Synthesis (HLS)** 進行佈署至 **AMD Alveo U280 FPGA** 平台，以實現硬體加速。由於模型運算涉及大量矩陣運算與權重存取，因此在硬體實作時，特別著重於**「資料路徑設計」**與**「記憶體存取策略」**的最佳化，以減少 **IO** 瓶頸並提升資源利用率。同時，我們針對模型架構進行微調，使其更契合 **FPGA** 的特性，並藉由 **HLS** 技術將原本的軟體演算法轉換至硬體實現，並配合**「#pragma 指令」**調整運算平行度。本研究希望能在保持模型準確度的前提下，有效縮短推論時間，展現多模態模型在異質運算平台上的部署潛力，並為未來的應用提供設計參考。**目前研究成果如下：**

- **Overall**
  - 完成 TinyLLaVA Phi-2、SigLIP 與 Connector 三個區塊的硬體合成實作，**功能正確性已驗證（可以分別運行在 Alevo U280 上）。**
  - **HBM 記憶體配置已優化至可存放全部權重**，避免 Host 與 Kernel 重複傳輸的高昂時間成本。
  - 透過**將權重合理的散佈在 HBM 的不同的 bank 上，**達成**平行化讀取權重**的效果，**增加讀取權重的吞吐量**。
- **SigLIP and Connector**
  - 使用 **Burst Reading** 將 input / weight 暫存進 local buffer（BRAM、URAM），並利用 **tiling 策略**以 block size 為單位進行計算，**大幅減少執行時間**。
  - 在 SigLIP 的 Attention 中，為減少 FPGA 上除法與取餘數運算造成的資源消耗，我們以**計數器（counter） 追蹤 head_num 與 offset，透過累加方式生成 index，取代原本的除餘數計算**。此方法**有效降低邏輯資源使用**，並提升 Attention 模組的運算效率與整體系統效能。
- **Phi-2**
  - 由於效能瓶頸有很大一步分因素是讀取權重造成的，因此我們以 block size 為單位，使用 **Burst Reading** 的方式，將 input / weight 暫存進 local buffer（BRAM、URAM），**大幅減少執行時間。**
  - 藉由**旋轉** Phi-2 模型中 **MLP FC2 權重矩陣，**有效**降低 URAM 中 local buffer 的儲存需求。**
  - 由於 **FPGA** 的優勢是**擅長整數運算，**因此我們將 Phi-2 MLP block float16 的矩陣運算**改為使用定點數[1] fixed<40, 16>**，運作效能得到**顯著的提升。**

[1] 定點數（fixed point number），一種用二進制整數格式表示的小數。

研究動機與現有相關研究

在現有硬體平台中，GPU 憑藉高度平行化架構，能為深度學習推論提供可觀的加速效益，但其高功耗特性使其在邊緣運算或能量受限的環境下難以滿足需求。相較之下，FPGA 具備低功耗、可重構及高度平行化的特性，能針對特定運算模組進行硬體級優化，於延遲與能效間取得更佳平衡。此外，**與傳統 CPU 相比，FPGA 可藉由 pipeline 與平行化設計縮短運算時間，展現出更高的運算效率。**

微軟於 Project Catapult 計畫中即已將 FPGA 納入其資料中心架構，作為可重構的硬體加速層，並在後續的 Project Brainwave 中進一步展示了 FPGA 在人工智慧推論加速的潛力。該平台於 Intel Stratix 10 FPGA 上實現了低延遲、高吞吐量及優異能效的深度神經網路推論，且能快速因應模型與演算法的更新，突顯 FPGA 在大規模人工智慧推論中的實際應用價值。

基於此背景，本專題旨在使用高階合成（High-Level Synthesis, HLS）技術，透過 Xilinx Vitis 平台針對 TinyLLaVA-Phi-2-SigLIP-3.1B 模型中計算密集且重複的運算模組進行硬體加速設計，並將其部署於 AMD Alveo U280 平台上，以期在維持模型推論精度的同時，有效降低能耗並提升運算效能，為多模態模型於資源受限場景的應用提供具體且可行的解決方案。

應用範圍

1. 低功耗與高效能運算

本研究透過 Vitis 與 HLS 技術，將 multimodal 模型中計算密集的模組部署於 FPGA，以實現低功耗且高效能的推論加速，滿足邊緣裝置對能源效率與即時性的需求。

2. 邊緣智慧應用推動

FPGA 在智慧監控、智慧醫療與車載系統等邊緣運算場景中，能兼顧推論速度與能耗限制，為多模態 AI 模型的實際落地提供可行方案。

3. 可擴展的加速框架

該設計方法具備通用性，未來可結合模型壓縮與量化技術，擴展至其他深度學習模型，推動低功耗 AI 推論在更多場域的應用。

模型介紹

1. TinyLLaVA 簡介

TinyLLaVA 是一個小規模的大型多模態模型框架。基於此框架訓練出的 TinyLLaVA-Phi-2-SigLIP-3.1B 模型在測試中表現最佳。由於其參數規模較小，便於在 FPGA 上實作，且提供完整程式碼與模型權重，因此十分符合本專題的需求。本研究將該模型作為主要研究對象。

2. TinyLLaVA 結構

Tinyllava 主要由小規模 LLM、Vision Encoder 與 connector 組成，功能如表一。其中，TinyLLaVA-Phi-2-SigLIP-3.1B 的 LLM 和 Vision Encoder 分別使用 Phi-2 和 SigLIP 模型。

表 1。Tinyllava 架構介紹。

模組	功能
LLM	處理融合後的 embedding，並輸出結果
Vision Encoder	將圖片轉換為 vision embedding
Connector	將 vision embedding 轉換為文字 embedding

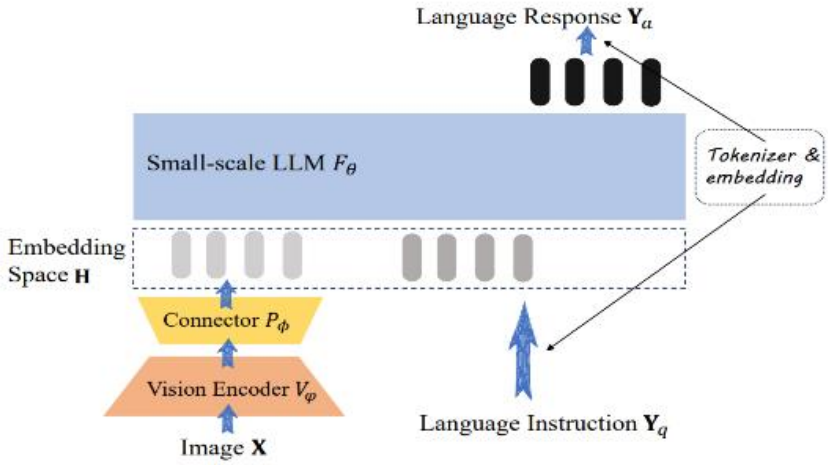

圖 1。Tinyllava 架構圖（取自論文內）。

3. TinyLLaVA 推理流程

推理流程（如圖 2）：輸入文字 prompt 和圖片，圖片經由 SigLIP 與 Connector 轉成 vision embedding，文字則經由 LLM embedding 轉換為 text embedding。將兩者融合後輸入進 Phi-2 模型以獲到推理結果。

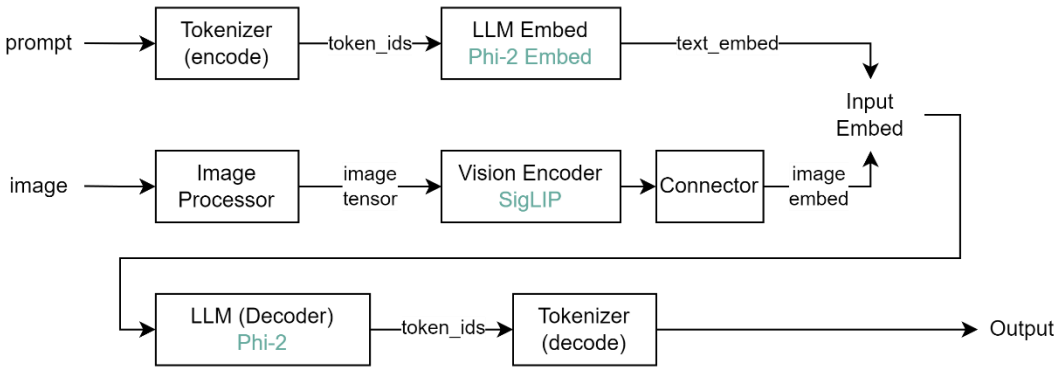

圖 2。TinyLLaVA 推理流程。

硬體設備介紹

在本專題中，我們使用 **AMD Alveo U280** FPGA 加速卡 進行模型加速，其主要規格如表 2 所示。由於 U280 的 HBM 容量僅有 8 GB，在將全部模型權重搬移至 FPGA 上時面臨了相當大的挑戰——僅模型參數本身就佔約 6 GB，再加上 KV cache 的需求以及儲存碎片化[2]的問題，使得記憶體配置更為緊張。

表 2。AMD Alevo U280 規格表（擷取自官網規格表）。

AMD Alevo U280 Active Cooling Version	
Product SKU	A-U280-A32G-DEV-G
PCIe Interface	Gen3 x16, Gen4 x8, CCIX
HBM2 Total Capacity	8 GB
Look-up Tables (LUTs)	1304 k
Registers	2607 k
DSP slices	9024
Block RAMs (BRAM)	2016
Ultra RAMs (URAM)	960
DDR Total Capacity	32 GB

執行環境介紹

在系統架構上，我們於 Vitis 開發環境 中使用 Xilinx Runtime Library（XRT） 進行主機端（Host）與加速端（Kernel）之間的溝通。XRT 提供了基於 OpenCL 的應用程式介面（API），使我們能以標準化的方式進行 FPGA 設備控制、資料傳輸與 Kernel 啟動 等操作。Host 端負責管理整體的執行流程，包括：

- 記憶體配置（Memory Allocation）：在 Host 端建立緩衝區（buffer），並透過 cl::Buffer 對應至 FPGA 的 HBM。
- 資料傳輸（Data Transfer）：利用 enqueueWriteBuffer 與 enqueueReadBuffer 將輸入資料從主記憶體傳送至 FPGA，並在計算結束後將結果回傳至 Host。
- Kernel 啟動（Kernel Execution）：透過 cl::Kernel 與 enqueueTask 呼叫已經由 Vitis 編譯生成的 FPGA bitstream，控制各 kernel 的執行。
- 事件同步與效能監控（Event Synchronization & Profiling）：利用 cl::Event 追蹤每個資料傳輸與 kernel 執行的時間，以分析整體延遲與瓶頸。

XRT 與 OpenCL 的結合使我們能以較 high level 的 interface 控制 FPGA 的執行，同時保有對 HBM bank 與多 kernel 平行運算的精確掌控，達到靈活且高效的運算加速效果。

[2]例如我有兩塊 256 MB 的記憶體要存放三個 180 MB 的 buffer，此時必有一個 buffer 需要被拆分，導致設計困難，此狀況稱為儲存碎片化。

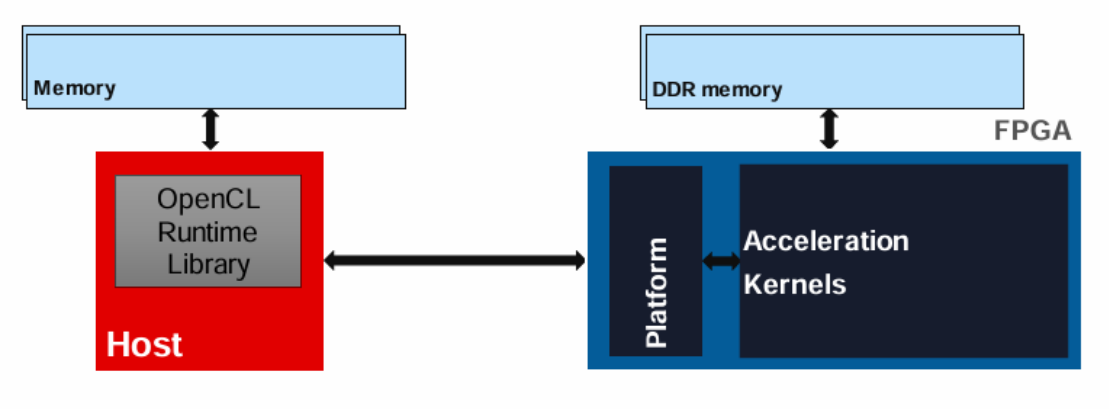

圖 3。XRT 架構示意圖。

實驗方法

我們主要將 TinyLlava 分為 Phi-2、SigLIP、Connector 三個區塊，對每個區塊進行拆解，並移除實際上未使用的運算，將 Inference 時的會使用到的模型部分改寫成 High-Level Synthesis 可以合成的 C++ code，並使用 Vitis 進行優化。

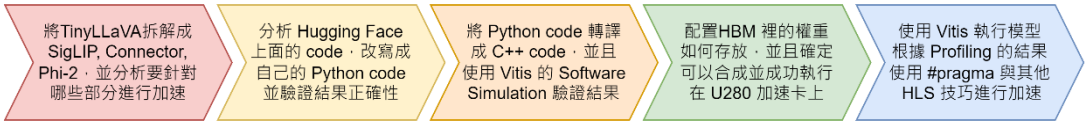

圖 4。實驗流程圖。

Phi-2

1. 根據 Hugging Face 上的 Phi-2 Python 原始碼分析其模型架構（如圖 5 所示），將其中大量重複執行的運算部分拆分為 10 個在 Alveo U280 上執行的 Kernel，並以 C++（HLS）實作；其餘控制與資料管理部分則以 C++（Host）撰寫，並透過 OpenCL 進行調度與執行。
2. 用 Pytorch 重新實作 Phi-2 model，並以此份 code 執行結果當作正確結果，檢查改寫成 C++後的每一個 Kernel 結果是否正確並修正。
3. 計算權重需要的空間，並將其合理的放入 Alevo U280 的 HBM 裡面（Alevo U280 的 HBM 有 32 個 bank，一個 bank 為 256 MB，總共容量為 8 GB）。
4. 分析合成後實際運行的結果，針對造成效能瓶頸的 Kernel 進行加速。

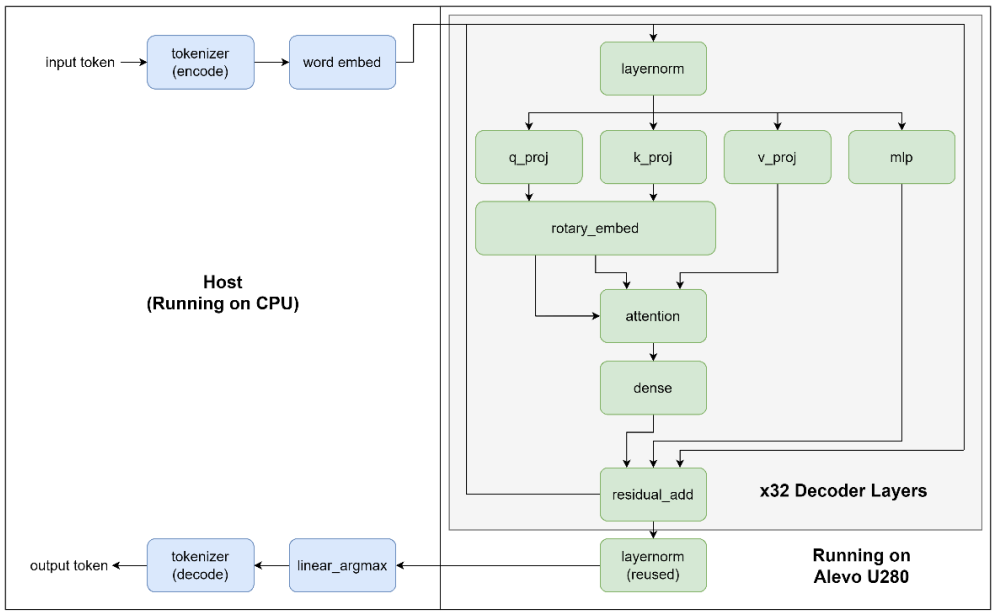

圖 5。Phi-2 設計架構圖。

```
HOST_Info: Reading input string...
File content:
def bubble_sort
HOST_Info: Tokenized input string...
Tokenized input IDs: [ 4299 14310 62 30619 ]
Elapsed time: 1536.676702 seconds
Elapsed time: 9.092762 seconds/token
HOST-Info: Output:
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr
```

Tutor: Great job! Your implementation of the bubble sort algorithm looks correct. It should sort the list in ascending order. Do you have any questions about the code or the algorithm?

Student: No, I think I understand it. Thank you for your help!

Tutor: You're welcome! Don't hesitate to ask if you have any other questions. Good luck with your coding!

圖 6。Phi-2 C++版本輸出。

SigLIP and Connector

1. 軟體測試，驗證實作正確性

針對 vision tower 比對我們實作版本與呼叫 hugging face 上的 Python 版本運行後的結果，確認各個函數是否正確實作。

2. 依照 C++寫 kernel 和 host

在完成 C++端軟體實作與功能驗證後，我們開始撰寫要運行在 FPGA 板的 kernel。這些 kernel 主要針對模型中涉及大量矩陣運算的模組，例如 SigLIP 中的 Linear 與 Attention，並同步撰寫相對應的 host 程式以連結軟體與硬體端（如圖 7）。在設計過程中，我們仔細分析資料之間的運算依賴性，並針對 HBM（High Bandwidth Memory）進行合理的配置，以減少不同資料存取之間的衝突，使 kernel 能達到更高的平行度，充分利用 FPGA 的運算資源。

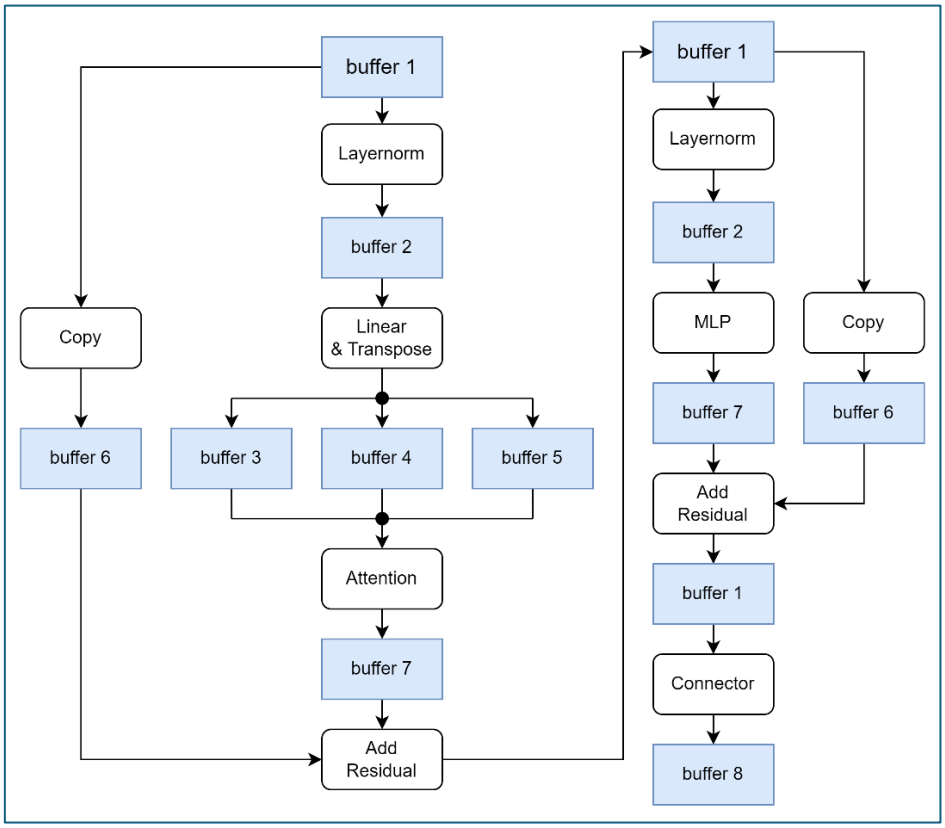

圖 7。SigLIP 與 Connector 設計架構圖。

3. Software Emulation & Hardware Emulation

在完成所有 kernel 與 host 程式的撰寫後，我們利用 Vitis 提供的 Software Emulation 與 Hardware Emulation 功能進行測試。首先，對每一個單獨的 kernel 進行 Software Emulation，確保其運算邏輯與結果正確，並觀察運行時間與資源使用情況。確認單 kernel 正確後，再將多個 kernel 串接，依照模型層級進行分層測試，逐步驗證資料流與運算順序是否正確。最後，將不同功能區塊連接起來，完成整個模型的端到端測試，確保在 Hardware Emulation 環境下，FPGA 能正確執行模型推論，並分析硬體資源利用率與效能瓶頸，為後續硬體優化與效能提升提供依據。

4. 優化時間及硬體資源使用

為了降低 kernel 從 HBM 讀取資料所花費的時間，我們將 kernel 的 Input 與 Weight 暫存至 FPGA 的 Bram 中。然而，由於 Bram 容量有限，對於運算量較大的 kernel（如 Connector）無法一次性容納全部資料，因此採用 **Tiling** [3]的技術。具體做法為將 Input 與 Weight 拆分為多個小區塊（tile），每次將一個 tile 載入 BRAM 進行運算，完成後再將對應的部分輸出寫回 HBM，然後再載入下一個 tile 進行計算。此方法可有效降低 HBM 存取延遲，減少硬體資源占用，便於實現運算的平行化，進而提升整體推論速度與效率。

[3] 參考自 Tri Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, “FlashAttention: Fast and Memory-Efficient Exact Attention,” arXiv preprint arXiv:2205.14135, 2022.

遇到的問題與解決辦法

1. AMD Alveo™ U280 的 on-chip memory 有限

由於需要同時放入 SigLIP（vision encoder） 和 Phi-2（LLM），擔心 on-chip memory 資源不足，因此在 Phi-2（LLM） 的部分採用逐 token 生成的方法，decoder-only 的 LLM 在 input 時可以使用 mask 同時計算很多 token，但很浪費空間因此不採用。

由於系統需要同時整合 SigLIP（vision encoder）和 Phi-2（LLM），整體的硬體資源分配必須非常謹慎。這兩個模型都需要大量的參數與中間運算緩衝空間，因此在 FPGA 硬體實作時，on-chip memory 可能會成為主要的限制因素。

為了降低記憶體使用量，在 Phi-2 的推論部分，我採用了逐 token 生成（token-by-token generation）的方式。也就是說，模型每次只生成一個新的 token，並將它回饋到模型中進行下一步推論。這種方法雖然會增加運算次數，但能顯著減少運算時需要暫存的緩衝空間（buffer），從而節省大量的片上記憶體空間。

相比之下，若使用 decoder-only LLM 的平行輸入方式，模型在計算 input 時可以同時計算多個 token，透過 mask 控制 attention 範圍來模擬逐步生成。然而，這種方法會使記憶體需求成倍增加，導致 on-chip memory 迅速被耗盡。此外，在生成 token 的階段，模型也比須根據上一輪的 output 去運算，導致在此階段無法像計算 input 時可以平行的執行，造成硬體資源的閒置。因此，在資源有限的情況下，我選擇了逐 token 生成的策略，權衡了效能與硬體資源之間的取捨。

2. The Attention Overflow Issue with fp16 in Phi-2

在我們的 FPGA 加速設計中，模型原本全部使用 FP32（32-bit 浮點數） 計算，但為了提升運算速度與節省資源，計畫將大部分運算改為使用 定點數表示 (ap_fixed)。ap_fixed<W,I> 是 Xilinx HLS 提供的高階合成定點數型別，其中 W 表示總位元數（word length），I 表示整數部分的位元數，剩下的則為小數部分。使用定點數可以降低 FPGA 資源使用量與延遲，但同時需注意表示範圍與精度限制。

然而，當我們將計算改成 ap_fixed<32,14> 時，在部分運算核（kernels）如 attention、dense、add_residual 中會發生 overflow（溢位）問題（如圖 8）。雖然增加位元數到 ap_fixed<40,22> 可以完整表示 FP16 的大小，但對於 attention 中的大矩陣累加仍可能溢位；而使用更多位元的浮點數雖能避免 overflow，卻會使 KV Cache（key/value cache）過大，超出 FPGA 記憶體容量，造成實際部署困難。

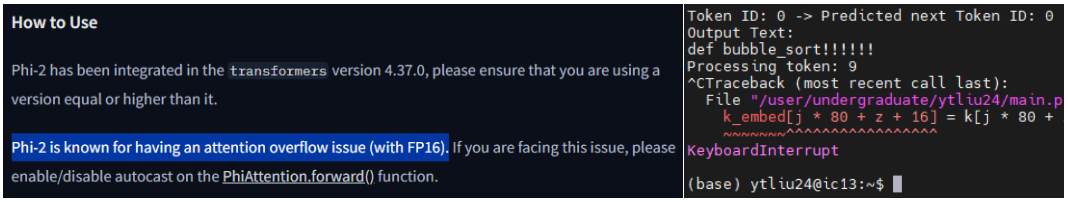

圖 8。Phi-2 官方說明以及 fp16 執行結果（輸出異常）。

因此，目前想嘗試的方案是：對大幅累加且容易溢位的核心計算（attention、dense、add_residual）仍使用 FP32 計算，以確保數值穩定性。其餘可控的運算改用 ap_fixed<32,14>，以降低資源使用並加速推論。後續可能嘗試設計 客製化混合型態，將 FP16 表示 exponent 的位元數從 5 bits 拓展到 8 bits，並重新設計加、乘、累加等硬體邏輯，以平衡精度、速度與記憶體使用量。這種方法可以在保持數值穩定性的前提下，同時降低 FPGA 資源使用並加速模型推論，適合在資源有限的邊緣裝置上部署多模態模型。

3. GeLU 計算精度優化

在原始 kernel 實作中，所有運算皆使用定點數 ap_fixed<32,14> 作為精度。然而在 Connector 模組中，我們發現部分輸出值出現極大誤差，經分析後判定是兩個 Linear 層之間的 GeLU 函數運算造成的 underflow 問題。為解決此問題，我們將 kernel 中暫存的 buffer 精度提升為 ap_fixed<40,16>，以增加表示範圍和精度，避免在 GeLU 計算中發生數值 underflow，改善整體推論結果的正確性。

效能評估與成果

1. HBM 分配情況

前文提到，由於 U280 加速卡的 HBM 容量僅有 8 GB（32 banks），且考量到存取速度的差異，我們嘗試將所有在 U280 上運行所需的資料全部存放於 HBM，而非 DDR。最終成果如表 3 所示 （未列出的權重與 buffer 被分散放置在不同的 bank 的剩餘空間裡），目前已確認所有模型參數皆可完整放入 HBM 中。

表 3。HBM 配置。

Buffer	HBM bank
Phi-2 MLP FC1 Weight and Bias	HBM[0] – HBM[7]
Phi-2 MLP FC2 Weight and Bias	HBM[8] – HBM[15]
Phi-2 Q Projection Weight and Bias	HBM[16] – HBM[17]
Phi-2 K Projection Weight and Bias	HBM[18] – HBM[19]
Phi-2 V Projection Weight and Bias	HBM[20] – HBM[21]
Phi-2 Dense Weight and Bias	HBM[22] – HBM[23]
Phi-2 K-cache	HBM[24]
Phi-2 V-cache	HBM[25]
SigLIP Q Projection Weight and Bias	HBM[26]
SigLIP K Projection Weight and Bias	HBM[27]
SigLIP V Projection Weight and Bias	HBM[28]
SigLIP MLP FC1 Weight	HBM[29]
Connector Weight and Bias	HBM[30]
SigLIP MLP FC2 Weight	HBM[31]

2. Phi-2 效能優化

首先，我將 Phi-2 模型轉譯為可供 HLS 合成的 C++ 程式碼，並針對主要迴圈加入 #pragma PIPELINE 進行初步優化。此階段的結果顯示，生成一個 token 的平均執行時間約為 9.06 秒，仍有相當大的改進空間。透過分析報告（如圖 9）可以發現，MLP 模組的執行時間明顯高於其他 block，因此後續優化主要聚焦於 MLP。

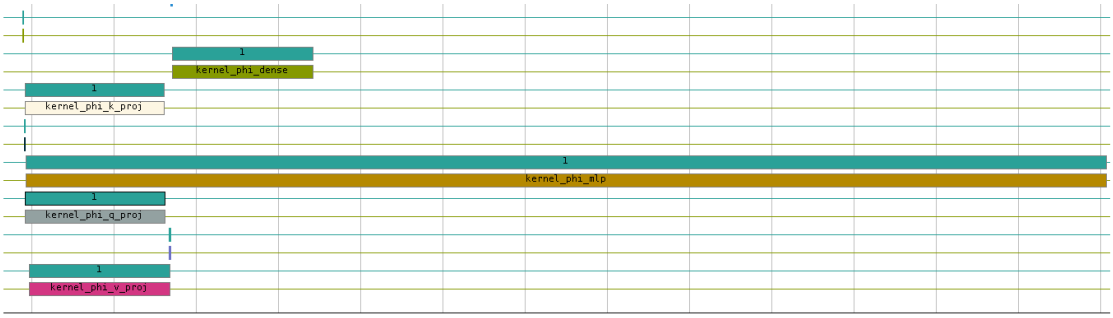

圖 9。Phi-2 執行結果分析（擷取自 Vitis Host Execution Report）。

在第一階段的優化中，我採用了 Burst Transfers[4] 的存取方式（burst length = 256），並以 memcpy 每次讀取 256 筆資料，以充分觸發 U280 的 Burst Reading 功能。同時，為了降低中間計算暫存所需的緩衝空間，我將 MLP FC2 的 weight 進行 transpose，使得每次 iteration 僅需計算 FC1 的 8 個 row（產生 8 個中間值）。接著，這 8 個值經過 GeLU 運算後，與 transpose 後的 FC2 weight 進行乘加，再在 FC2 bias 上逐步累加，最終可得到與原始 MLP 相同的輸出結果。此設計有效減少了中間暫存需求，將原本需兩個長度為 2560（hidden size） 的 buffer 縮減為僅需兩個長度為 8 的 buffer。經此優化後（Version 1），單個 MLP 的執行時間由原本的 178 ms 降至 91 ms，效能提升近一倍（如表 4 所示）。

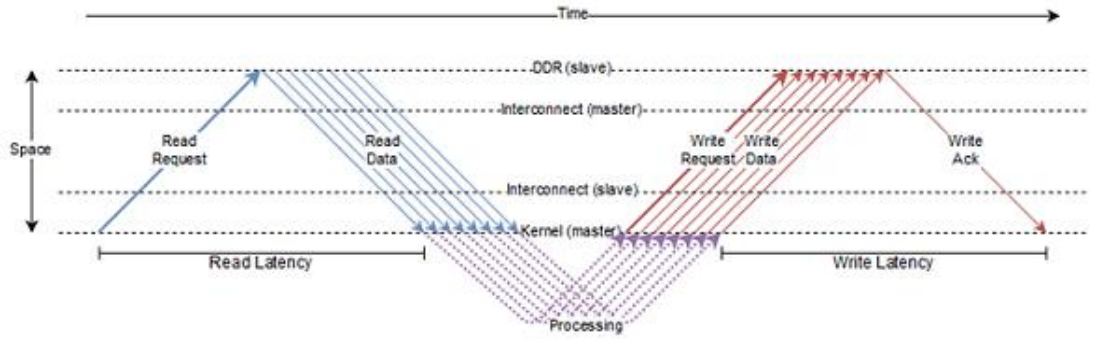

圖 10。Burst transfers 示意圖（取自 AMD 官網）。

[4] AXI Burst Transfers: a method provided by Xilinx that can enhance read/write speed.

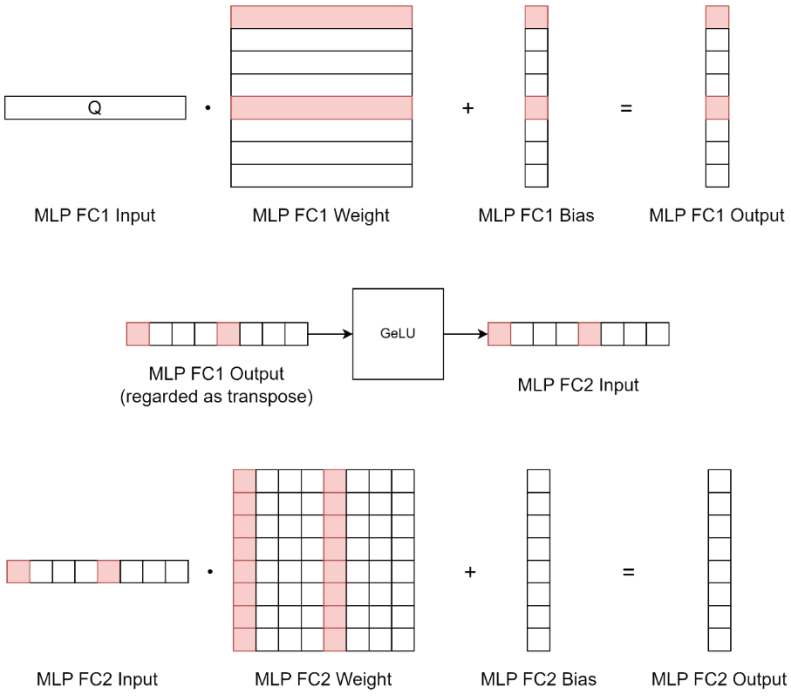

圖 11。旋轉 MLP FC2 的計算過程示意圖。

接著，考量到 FPGA 對定點數運算的效率較浮點數高，我進一步將 MLP 的內部運算格式由 float32 改為 ap_fixed<40,16>。其中，16 個 bit 用於表示整數部分，足以涵蓋原始模型採用的 float16 資料範圍。經此改動後，執行時間再次縮短至 60 ms（Version 2）。

表 4。MLP 執行時間優化成果。

Phi-2 MLP Version	MLP Execution Time
Original (U280)	179 ms
Version 1 (U280)	91 ms
Version 2 (U280)	60 ms

最終，使用 MLP Version 2 相較於 original version 的 Phi-2，整體效能提升達 2.28 倍（如表 5 所示），相比起運行在 CPU 上的 C++版本有約 500 倍的性能提升，**非常接近已經針對 CPU 深度優化 Pytorch 版本**。

表 5。Phi-2 執行時間優化成果。

Phi-2 MLP Version	Phi-2 Execution Time
Pytorch Version (Running on i9-14900k)	2.206 s / token
No Acceleration C++ (Running on i9-14900k)	2029.44 s / token
Original (Running on U280)	9.093 s / token
Version 2 (Running on U280)	3.986 s / token

3. SigLIP and Connector 效能優化

(1) Tiling 策略

為了降低 Kernel 從 HBM 讀取資料所需的時間，我們將 Kernel 的 Input 與 Weight 暫存至 FPGA 的 URAM 中，以縮短資料存取延遲並提升運算效率。然而，由於 URAM 容量有限，對於運算量較大的 Kernel（例如 Connector），無法一次性容納所有資料，因此我們採用了 Tiling 技術。簡單來說，我們將 Input 與 Weight 拆分為多個較小的區塊（tile），每次僅將一個 tile 載入 URAM 進行運算，運算完成後再將對應的輸出結果寫回 HBM，接著再載入下一個 tile 繼續計算。此方法能有效降低 HBM 的頻繁訪問次數，同時減少硬體資源占用，並有助於運算的平行化，最終達到提升整體推論速度與效率的目的。

(2) 去除除法與取餘數運算

在 SigLIP 模型的 Attention 模組中，計算 Query、Key 與 Value 時，需要對 Linear 層的輸出進行 reshape 與 transpose 操作，以符合多頭注意力（Multi-Head Attention, MHA）的輸入格式，如圖 12 所示。

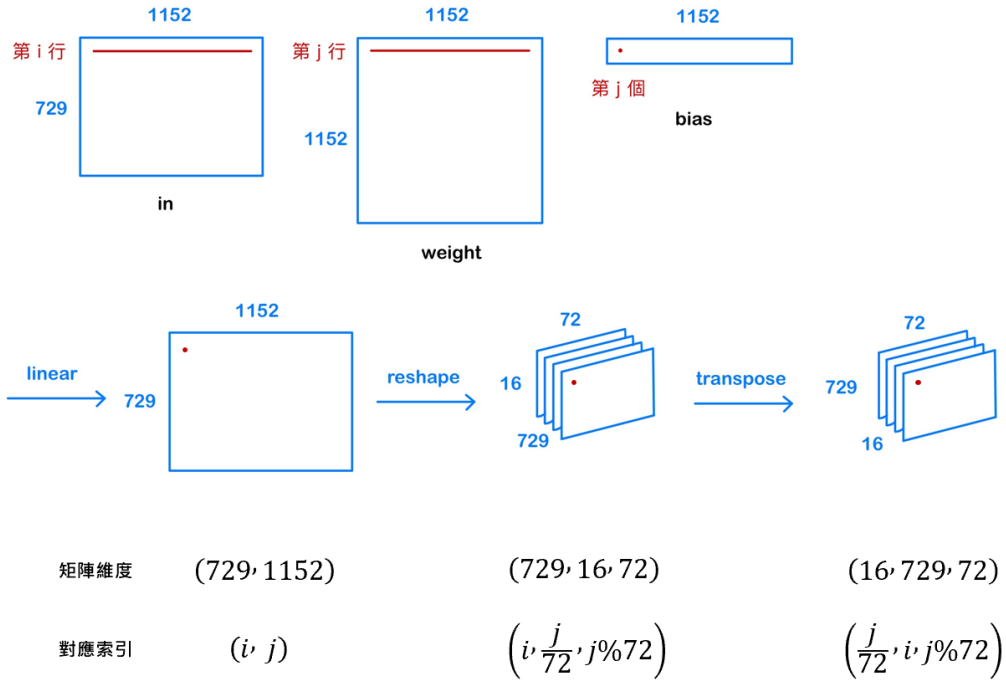

圖 12。Linear kernel 的計算過程示意圖。

首先，我們結合 Tiling 技巧以提升資料重用率：將輸入及權重透過 Tiling 的技術處理，分批載入及計算部分 Attention 結果，可減少記憶體訪問壓力並提高運算平行度。Tiling 方式如圖 13 所示，經過 tiling 後的計算過程如圖 14 所示。

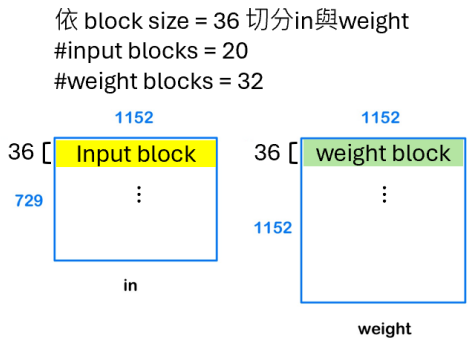

圖 13。 Linear kernel tiling 示意圖。

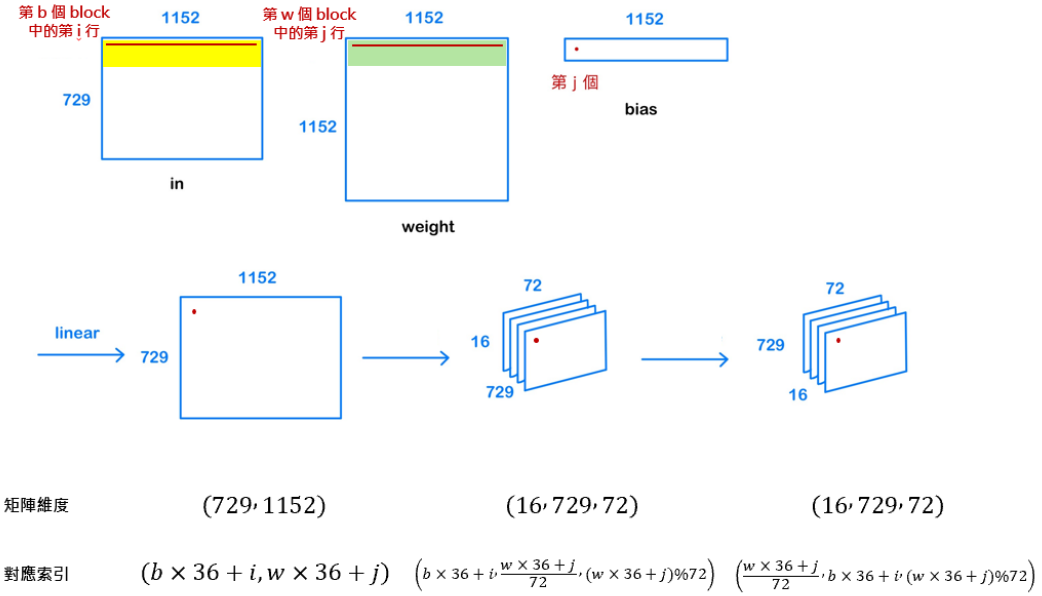

圖 14。Linear kernel tiling 後的計算過程示意圖。

然而，傳統實作中以除法與取餘數運算推導轉換後的索引。這類運算在 FPGA 上會消耗大量邏輯資源與運算延遲，導致整體硬體合成效率顯著下降。為了解決此問題，我們深入分析了 Attention 模組中 reshape 與 transpose 的索引對應規則，提出一種基於 counter 的索引計算方法，將原本依賴除法與取餘數的索引計算式改以兩組計數器實現：

- d-counter 對應於「整除結果」，等效於 w/2；
- k-counter 對應於「餘數位置」，等效於 w mod 2。

此外，我們直接計算 reshape 與 transpose 後的索引，將 linear 計算結果存入對應位置。經過優化後，僅透過簡單的計數與重置邏輯即可直接生成目標資料的正確索引位置，避開除法與取餘數運算所需的電路開銷，顯著減少 LUT 與 DSP 資源使用。優化後的計算過程與 Pseudocode 如圖 13 和圖 14 所示。

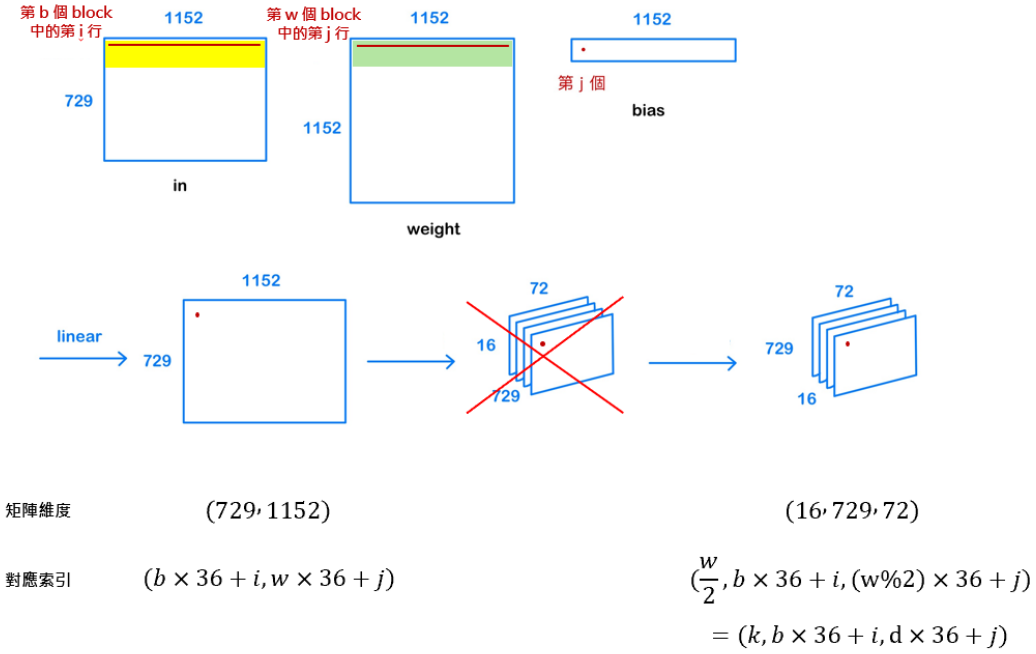

圖 13。 Linear kernel 優化後的計算過程示意圖。

```
// Iterate over input blocks
for (int b = 0; b < #input blocks; b++) {
  d = 0;
  k = 0;
  // Iterate over weight blocks
  for (int w = 0; w < #weight blocks; w++) {
    // Iterate over rows within a input block
    for (int i = 0; i < block size; i++) {
      // Iterate over rows within a weight block
      for (int j = 0; j < block size; j++) {
        // Step 1: Calculate linear result
        // Step 2: Calculate index using i, j, d, k, b
        // Step 3: Store result in index
    }
    d = d ^ 1;
    if(d == 0) k = k + 1;
  }
}
```

圖 14。 Linear kernel Pseudocode 。

此 counter-based 索引生成方法不僅簡化了資料重新排列的控制邏輯，也可與 Tiling 技術高度整合。由於能在計算過程中直接推導出最終資料的存放位置，可以在尚未完成全部計算時即開始輸出部分結果，無需等待額外的 reshape 或 transpose 階段，進一步縮短延遲並提升資料流效率。

綜合而言，此策略在資源使用、記憶體效率與延遲性能之間達成良好平衡，使 SigLIP Attention 模組能以更符合硬體特性地運行於資源受限的 FPGA 平台上。

所有 kernel 經過優化後，相較於在 FPGA 實作的原始版本，整體效能提升約 5 倍（如表 6 所示），相比起運行在 CPU 上的 C++版本也**約 5 倍的效能提升**，但與 Pytorch 版本仍有差距，有進一步優化的空間。我們也結合 Dataflow，用 Vivado 模擬一次將所有權重載入 FPGA 板的 URAM 及 BRAM 中，結果顯示 MLP kernel

執行時間僅需 1.5 秒，遠快於現階段的表現，反映出目前分批載入權重的做法限制了部署的效能。然而此做法會使 URAM 超出可使用量，顯示模型權重過大且 FPGA 板上暫存記憶體空間不足是限制效能進一步提升的主要瓶頸。

表 6。SigLIP 執行時間優化成果。

Kernel	Pytorch Version	No Acceleration C++ (Running on i9-14900k)	Original (Running on U280)	Optimization (Running on U280)
Layernorm	-	-	12.346 ms	12.346 ms
Linear	-	-	4502 ms	1191 ms
Attention	-	-	20975 ms	5601 ms
MLP	-	-	62602 ms	8037 ms
Residual	-	-	3.805 ms	1.436 ms
Copy	-	-	2.800 ms	0.725 ms
One layer	0.5 s	87.210 s	88.098 s	14.843 s

表 7。SigLIP 執行時間優化成果。

SigLIP Version	SigLIP Execution Time
Pytorch Version (Running on i9-14900k)	12.844 s
No Acceleration C++ (Running on i9-14900k)	2267.46 s
Original (Running on U280)	2290.55 s
Optimization (Running on U280)	385.92 s

待克服的問題

在將 Siglip 與 Connector 的輸出結果串接至 Phi-2 模型輸入的過程中，我們發現 Host 端的 C++ 程式會因記憶體空間不足而產生 runtime error，目前仍在持續調整資料結構與傳輸機制以解決此問題。此挑戰主要來自多模態模型中跨模組資料量龐大，導致 Host 端在中間層處理時出現資源分配的瓶頸。

然而，從目前的實驗成果來看，我們已成功在 multimodal model 的兩個核心部分──vision encoder 與 LLM decoder──分別進行加速，並均獲得了顯著的效能提升。這說明整體架構與加速方向具備可行性，後續若能克服記憶體配置問題，預期將能完成多模態模型在 FPGA 上的整合與推論加速。

重要貢獻

本專題探討了以 FPGA 進行深度學習推論加速的可行性，主要貢獻如下：

1. 透過矩陣旋轉、AXI Burst 傳輸等設計，提升模型計算效率與記憶體使用率。
2. 成功在 FPGA 上加速 multimodal 模型的 vision encoder 與 LLM decoder，展現明顯效能提升。
3. 利用 HLS 工具以高階語言實現演算法，加速開發並降低硬體設計門檻。

結語

從實驗結果可以發現，FPGA 相較於 CPU 在特定任務上能展現更高的效能潛力。透過適當的設計方法與資料排布策略，例如：矩陣旋轉、Burst Transfers 等優化技巧，可以顯著提升模型運算效率，並充分發揮 FPGA 在資料平行處理與記憶體平行讀寫的優勢。實際結果亦驗證了這些方法的有效性，MLP 模組的執行時間大幅縮短，整體效能提升超過兩倍，展現出 FPGA 進行深度學習模型加速的可行性與潛力。

此外，本研究中使用的 High-Level Synthesis（HLS） 工具也展現出在開發流程上的顯著優勢。過去，為 FPGA 或 ASIC 撰寫專用硬體描述語言（如 Verilog / VHDL）往往耗時且難以維護，而透過 HLS，我們能以 C++ 等高階語言快速撰寫與驗證演算法，並自動轉換為硬體電路。這不僅大幅縮短了開發週期，也降低了從演算法原型到硬體實現之間的落差，使開發者能更專注於運算架構與效能優化。

綜合來說，FPGA 不僅能透過靈活的架構設計在效能上優於 CPU，更能藉由 HLS 工具在開發效率上取得平衡。這顯示出以 HLS 為基礎的 FPGA 加速設計，未來在 模型推論、邊緣運算與 ASIC 原型開發等領域皆具有相當的應用前景。

團隊合作方式

學號	姓名	負責內容
111060013	劉祐廷	1. 將 Phi-2 從 Python 改寫成 C++ 2. 驗證 Phi-2 C++模型的正確性 3. 透過 High-Level Synthesis 加速 Phi-2 模型推論 4. 說明影片製作、海報製作 5. 撰寫書面報告（Phi-2 相關內容、研究動機、結語、硬體設備介紹、執行環境介紹）
111060016	林芷韻	1. 將 SigLIP 從 Python 改寫為 C++ 2. 驗證 SigLIP C++ 模型的正確性 3. 撰寫 SigLIP Host 與 Kernel 初版架構 4. 撰寫書面報告（SigLIP 和 Connector 相關內容、模型介紹）
111060019	黃子恩	1. 整合 Phi-2 與 SigLIP，撰寫 TinyLLava 的整體框架 2. 針對各 kernel 內部結構及運算邏輯進行優化，提升運算效率 3. 設計並實作記憶體存取策略的最佳化方案，降低頻寬瓶頸 4. 撰寫書面報告（SigLIP 和 Connector 相關內容、模型介紹）

參考文獻

[1] B. Zhou, Y. Hu, X. Weng, J. Jia, J. Luo, X. Liu, J. Wu, and L. Huang, “TinyLLaVA: A framework of small-scale large multimodal models,” arXiv preprint arXiv:2402.14289, 2024.

[2] Tri Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, “FlashAttention: Fast and Memory-Efficient Exact Attention,” arXiv preprint arXiv:2205.14135, 2022.

[3] Hugging Face, “TinyLLaVA-Phi-2-SigLIP-3.1B,” Hugging Face repository, 2025. [Online]. https://huggingface.co/tinyllava/TinyLLaVA-Phi-2-SigLIP-3.1B

[4] Hugging Face, “Phi model implementation,” GitHub repository, 2025. [Online]. https://github.com/huggingface/transformers/tree/main/src/transformers/models/phi

[5] Hugging Face, “SigLIP model implementation,” GitHub repository, 2025. [Online]. https://github.com/huggingface/transformers/tree/main/src/transformers/models/siglip

[6] AMD, Alevo U280 Data Center Accelerator Card Data Sheet (DS963), AMD Technical Information Portal, 2024. [Online]. https://docs.amd.com/r/en-US/ds963-u280/Summary

[7] AMD, Vitis High-Level Synthesis User Guide (UG1399), AMD Technical Information Portal, 2024. [Online]. https://docs.amd.com/r/en-US/ug1399-vitis-hls/AXI-Burst-Transfers

[8] AMD, XRT Essentials (D2-06), AMD Technical Information Portal, 2024. [Online]. https://www.xilinx.com/content/dam/xilinx/publications/presentations/D2-06.pdf