國立清華大學 電機工程學系

實作專題研究成果摘要

Phi-2 Inference Acceleration on FPGA Via High Level Synthesis

在 FPGA 基於 High Level Synthesis 的 Phi-2 推論加速

專題領域：系統領域

組　　別：B615

指導教授：黃稚存教授

組員姓名：黃威儒、吳冠廷

研究期間：2024 年 12 月 23 日至 2025 年 11 月 23 日止，共 11 個月

摘要

本專題以 HLS (High-Level Synthesis) 於 AMD/Xilinx Vitis 平台實作語言模型 Phi-2 在 FPGA - Alveo U280 上之硬體加速。我們針對原始 phi2-2.7B 模型進行量化，對 Linear 層作 8-bit Weight & Activation Quantization 並使用 integer-only 矩陣運算。接著於 C++ 平台實作 Phi-2 模型，確保內部結構和 Python 完全對齊。最後於 AMD/Xilinx Vitis 平台，進行硬體排程設計，結合多 kernel 切分（Attention/MLP 與 Projection Q/K/V 平行化）並優化權重參數的 HBM 記憶體配置，同時提升帶寬利用與並行度。

整體結果顯示，本研究成功驗證 CPU+FPGA 架構於語言模型推論的可行性，**並於 Alveo U280 上達到 0.8 s/token 的推論速度**，達原始 Python 經過 PyTorch 優化之 **CPU 的 4.1 倍**。同時功耗僅 **11.8 W**，遠低於 CPU、GPU 的使用量，且量化後參數**僅需原本 46%的記憶體空間**。顯示在功率受限或邊緣部署應用中具備實用價值。本專題的重點有：

- Python 開源語言模型 C++ 重現、HLS 實作
- 語言模型的 8-bit Post-Training Quantization 實作，減少記憶體需求
- CPU + FPGA 跨平台架構
- 硬體並行化與排程、記憶體配置，加速模型推論、降低能耗

1. Introduction

語言模型規模快速增長，實務部署常面臨高延遲、高功耗與記憶體帶寬／容量壓力，影響即時互動與長時間穩定運作，難以在 edge device 上實現高效能部署。因此我們著手研究適合部署的模型，並實作系統架構。目標是在功能與數值對齊下，**加速推論、降低能耗，並減輕外部記憶體流量**，評估語言模型於 edge device 部署的實用性。

High-Level Synthesis（HLS）允許設計者從 C/C++ 或 SystemC 等高階語言描述硬體邏輯，再經合成得到可在 FPGA 上執行的 RTL (Register-Transfer Level) 電路。這種方式大幅降低傳統 RTL 設計的門檻，並在深度學習加速領域已有成熟應用。

近期已有研究將語言模型移植至嵌入式 FPGA。2024 年 Xu 等人提出 LlamaF [1]，結合 Group-wise 8-bit Weight & Activation Quantization (W8A8) 與 Fully Pipelined 加速器等方法，驗證在量化與管線化設計加持下，LLM 於受限資源的嵌入式平台具可行性。另一方面，針對 HLS 工具於 Transformer／語言模型加速應用上的研究亦開始出現。例如 HLSTrans-form 一文採用 HLS 實作 Llama2 模型，在 FPGA 上相較於 CPU 約減少 **8.25**× 的每 token 能源消耗。[2]

基於上述趨勢，本研究選擇參數量較小且性能佳的語言模型（Phi-2, 2.7B）。先參照其官方模型架構，自行使用 PyTorch 建立量化模型，降低權重與特徵的記憶體與運算負擔；接著採用 CPU+FPGA 異質化設計，將推論中運算／存取占比最高的關鍵路徑交由 FPGA 加速；實作 HLS 建立模型，提升開發效率與架構探索彈性，並設計硬體排程與記憶體配置優化。最終我們的架構達成的推論速度能比原本 C++ 模型快 14x，也比針對 CPU 優化的 PyTorch 版本快 4.1x。且功耗是所有比對對象中最低的。

1-1. 系統架構

a. Phi2 Architecture [5]

圖 3 為 Phi-2 整體模型結構，Phi-2 是一個典型的 Decoder-only Transformer 架構，由 32 層堆疊的 Transformer Block 所組成，輸入經過模型內部多層結構運算，最後輸出成語意清晰的文字。每層包含 Attention 與 MLP 兩個主要運算單元，透過每層輸入的 LayerNorm 與中間的 Residual Add 結構，讓模型數值穩定，避免溢位的狀況出現。最後一層 lm_head 透過線性層 (linear classifier)，將 hidden size (2560) 投影到 phi2 詞彙表大小 (vocab = 51,200)。輸出為每個詞彙的機率分布，用於預測下一個 token。

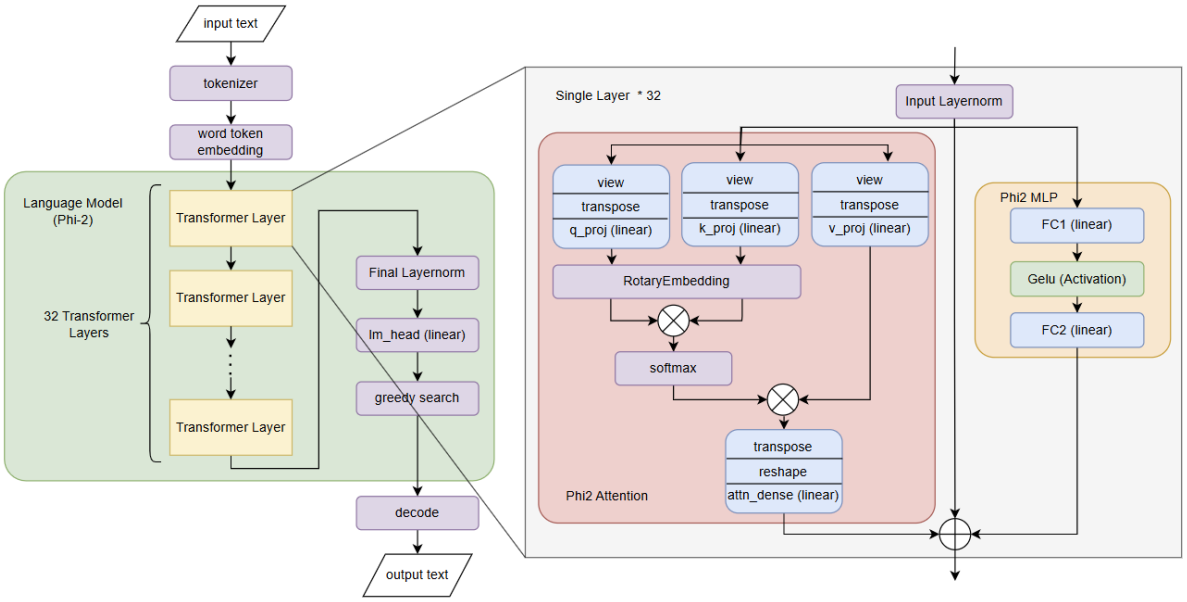

圖 1. Phi2 架構

b. Component attention

圖 7 為 Phi-2 中的 Attention 模組結構。Attention 是 Transformer 的核心運算單元，主要透過 Query/Key/Value 去計算輸入中不同 token 的關聯性。輸入向量經過線性層得到 Q、K、V，並在 Q、K 上加入 Rotary Position Embedding (RoPE) 讓模型學會相對位置資訊。接著計算 attention score (Q @ K)，通過 softmax 正規化後，將權重加到 V 上形成加權輸出。最後經過 attn_dense 線性層投影，將所有 head 的結果整合回 hidden size (2560)，提供後續層計算。

1. Research Methodology

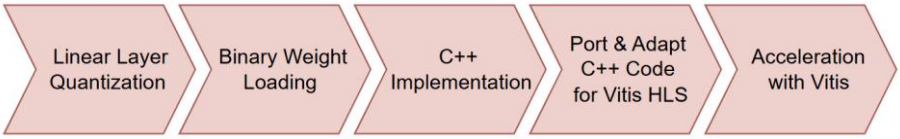

圖 2. 實作流程圖

- Linear Layer Quantization: 為降低權重儲存與記憶體搬運成本，我們對計算／存取占比最高的 Linear layer 進行量化。
- Binary Weight Loading: 將量化後的.pt 權重檔轉為.bin 檔格式載入 C++ 環境中。
- C++ implementation: 我們完成原始 Python 開源模型的 Phi-2 的 C++ 實作，針對每個 Component 實作 C++ Code，並確保結果與 Python golden 完整對齊。
- Port & Adapt C++ Code for Vitis HLS: 將核心模組在 C++ 基礎上改寫為 HLS C++，同時實作 Host (XRT/OpenCL) Code 對 kernel 進行整合與硬體排程。
- Acceleration with Vitis: 確認功能正確無誤後，依 URAM／BRAM／DSP／LUT 資

源使用情況，以 pipeline、unroll、ARRAY_PARTITION 等 pragma 進行調參與優化，以及調整記憶體配置，於既定資源下最大化模型加速。

2-1. 軟體設計 – Linear layer 8-bit Weight & Activation quantization

我們對線性層作 8-bit 量化，其中模型最後一層 lm.head 線性層未做量化。達成困惑度僅增長 0.78，卻能減少 46% 記憶體存取、且能作整數運算大幅減少運算時間。

1.1.1. 我們的量化方法

經過多次實驗 (我們會在「效能評估與實驗成果」詳細解釋) 我們決定使用 Linear layer 8-bit Weight & Activation 量化與 integer only 計算，並透過以下流程達成：

A. 先取得模型參數，並將線性層 (除了 lm_head 線性層) 的 weight (bias 不做量化) 對稱量化後存下 8 bit 權重 weight 以及量化參數 scale。
B. 遍歷模型的 modules 找到 nn.linear 層，並透過 Pytorch setattr 置換成我們做的、能讀入 8 bit 權重與量化參數的線性層。
C. 從 Phi-2 訓練集 ultrachat_200k [10] 隨機取出 1024 筆校正資料集，校正 activation 的數值範圍，紀錄並存下線性層前的 activation 非對稱量化所需的量化參數 scale、zero。如此一來，我們就能在實際推論時將預先儲存的參數與輸入做量化與 integer only 計算，詳細公式如下，其中 x 為輸入向量，W 為權重矩陣，s_x、z_x、s_w 是量化參數，b 是 bias 向量。

量化：

$$q_x = round\left(\frac{x}{s_x} + z_x\right)$$

$$q_w = round\left(\frac{W}{s_w}\right)$$

INT 8 整數乘法：

$$acc_i = \sum_{j=1}^{N} q_w(i,j)(q_x(j) - z_x)$$

反量化：

$$y_i = (s_x s_w)acc_i + b_i$$

1.1.2. 困惑度（Perplexity）

另外，我們以困惑度作為量化後模型表現的指標。對一段長度 N 的 token 輸出序列 $(x_1, \dots, x_N)$，模型在每個位置 t 輸出值為 x_t 的機率為 $P(x_t|x_{t-1}, \dots, x_1)$。困惑度越低代表輸出的文字機率越高，模型回覆的信心越高。由於不需要依靠資料集測量表現，僅需跑幾筆測試資料即可算出，方便於前期比較量化後模型與原始模型的表現。我們使

用 512 筆測試資料集測量模型困惑度。

$$Perplexity = \exp\left(-\frac{1}{N}\sum_{t=1}^{N}\log\left(P(x_t|x_{t-1},\ldots,x_1)\right)\right)$$

2-2. Host Program 設計

透過分析 phi2 架構，發現 Attention 與 MLP 為兩個相互獨立的運算單元，除此之外，在 Attention 內部，Q/K/V projection 三者之間亦不存在 data dependency，因此我們重新設計了 Host 管理流程，具體包含以下幾點：

2.2.1. 逐層切分 Kernel

將原本單一大型 kernel，切分為多個小型 kernel，每層分別對應 Attention 或 MLP。這樣的設計不僅能在跨層級上進行 kernel 調度，還能將 Attention 內部的 Q/K/V projection 進一步拆分，同時交由不同的計算資源處理。透過 Host 的排程，我們得以實現跨 kernel 的平行化，大幅縮短整體執行時間。

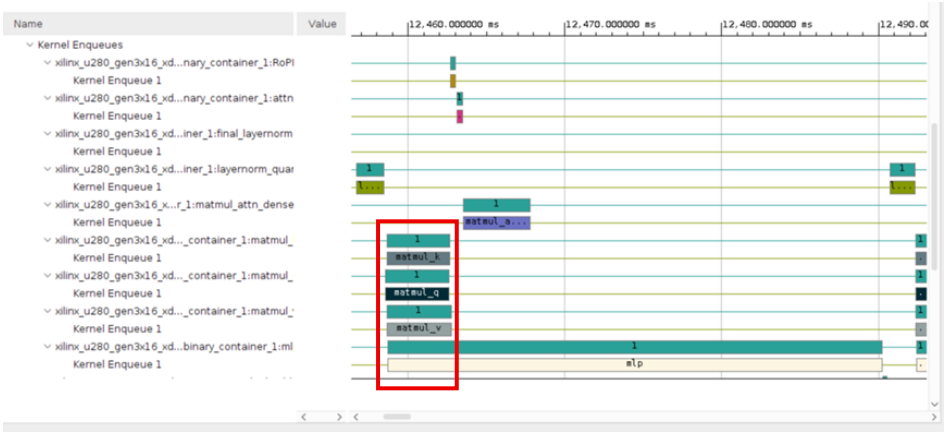

圖 3. kernel parallel 示意圖

2.2.2. HBM Bank 分配

U280 的 HBM 被劃分為多個獨立的 bank，每個 bank 容量有限，且 BW 在同一時間只能處理有限數量的訪問。若多個 kernel 在同一時間訪問同一個 bank，會造成 帶寬衝突，不但沒運用到 HBM 的優勢，同時還會因過長的等待時間導致效能降低。因此，我們採用了以下設計方法進行權重與 buffer 的空間映射設定：

- 權重分散存放：將不同 kernel 的權重分配到不同的 HBM bank，避免多個 kernel 同時讀取相同 bank。
- 佈線空間：讓 kernel 與其主要使用的 bank 放置在鄰近區域，避免跨 SLR(運算單元) 長距離走線。

這樣的設計同時考慮了 時間上的併行 (kernel 排程) 與 空間上的分配 (HBM/PLRAM 分區)，使得每個 kernel 在執行時都能完整地運用到 HBM 的高頻寬速

度進行資料存取。經過上述優化後，整體系統效能大幅提升：

2-3. Phi2 kernel 設計

2.3.1. Architecture - phi2-single token inference

在 LLM 推論流程中，通常將運算分為 Prefill 與 Generate 兩個階段

- Prefill：一次性處理整個輸入 token，並建立 Key/Value (KV) Cache，過程中並不會產生文字(next token)。
- Generate：每次只輸入一個 token (上一回產生的 token)，並利用 KV Cache 遞迴生成下一個 token。

在原始 python 中，Prefill 可以對所有輸入 tokens 進行平行運算。然而，在 FPGA 上，並沒有足夠的運算資源以及 on-chip buffer 能夠進行這樣的操作。因此，我們提出單 token 推論 (single-token inference) 的架構，將 prefill 與 generate 階段統一為逐 token 計算，這樣的實作在 FPGA 上有以下優勢：

- prefill 與 generate 共用一組硬體 kernel：不需要特別實作兩份 kernel，並且能針對共同架構進行優化。
- 提升 kernel 資源使用率：避免 kernel 大多時間處於 Idle 狀態而浪費硬體資源。
- 降低功耗、佈線複雜度

2.3.2. Attention

Phi2 attention 共有 32 個 head，每個 head dim = 80 (32 * 80 = 2560-hidden_size)，在原始 Python 中，會透過 Reshape、Transpose 的方式改變矩陣排列方式，能達到同時對所有 head 進行計算的效果，然而在 FPGA 上，並沒有足夠的資源支撐如此的運算，因此我們改採對每個 head 分別進行計算，並透過 index 的調整達到同樣的計算結果。

a. Phi2-KV cache

Phi 為 Decode-only 的語言模型架構，在 Attention 中，當前 token 的計算需要使用當前 token 的 Query 以及先前 token 的 Key、Value，因此需要使用 KV-Cache 將計算過的 KV 值保存下來，否則運算複雜度會隨 length 成正比。

我們的設計在 prefill 階段，會在每次計算完 key 和 value 值後直接將計算結果存入 KV-cache 中，供後續訪問使用，同時，考量到 KV-cache 的訪問次數會隨著 output 逐漸增長 O(length)，勢必會成為模型的 bottleneck ，因此我們分別設計 Key-Cache 與 Value-Cache 並且存放於不同的 HBM bank 之中，避免了兩者被同時訪問時所造成的衝突，並完整利用 HBM 的高頻寬優勢。

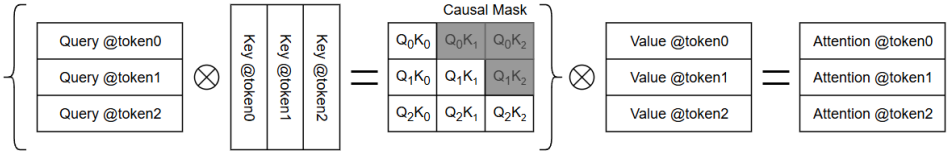

圖 5. Attention without KV cache

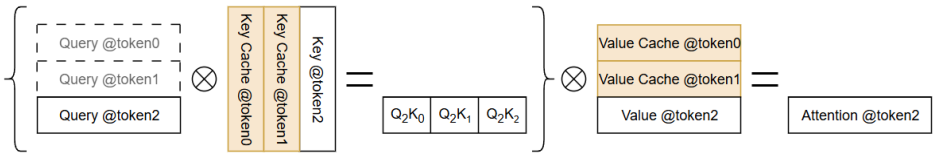

圖 6. Attention with KV cache

2.3.3. Linear (MLP/ q_proj/ k_proj/ v_proj/ attn_dense)

在硬體端，我們優化線性層的記憶體存取。我們主要使用三種方式作優化：

a. 權重分拆

將權重拆分成多份，存在多個 HBM 上。推論時同步讀入，並在多個複製電路同步做乘加。如此可以增加記憶體讀取的 bandwidth，高效消化讀入的資料。

b. AXI burst

善用 AXI burst 記憶體傳輸介面。根據 AXI Burst Transfers 的 burst 讀取規定，burst 與記憶體進行一次握手後能連續搬多個 beats 的資料。一個 beat 會在一個 clock 時間內讀入，且一個 beat 最高可容納 512 bits 的資料。由此可透過 AXI burst 的多個 beats、512 bits per beat 大量讀入資料。

c. 記憶體順序

按照記憶體順序讀取。因為在 HBM 中的資料是按照地址順序排列，所以若按照逐 row 順序讀取矩陣能比逐 column 跳著讀獲得更快的讀取效率。特別是 KV cache 部分有兩次矩陣乘法，使用合適運算方法即可按照記憶體順序讀取。

2-4. HLS 加速技巧 - Pipeline+Unroll

2.4.1 Pipeline

根據 pragma HLS pipeline [11]，#pragma HLS pipeline 方法會在硬體中插入 pipeline registers 與控制邏輯，讓迴圈/函式理想上能每拍接受新輸入，使 Initiation interval = 1。在不額外使用太多運算資源的情況，大幅提高運算效率。

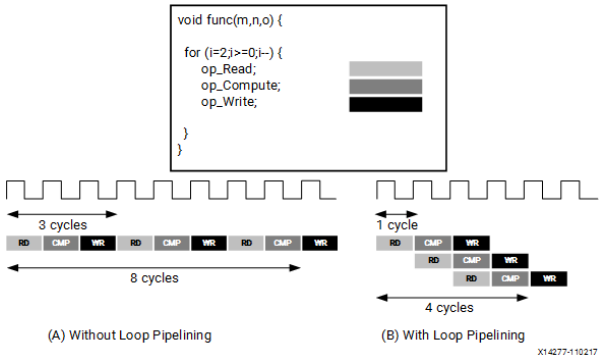

圖 8. AMD Pipeline 解釋圖

2.4.2. Unroll

根據 pragma HLS unroll [12]，#pragma HLS unroll 能把迴圈展開成多份「平行的硬體電路」，直接複製多份硬體來並行執行迴圈，讓多個迭代可以同時執行。透過適度搭配以上兩種主要 pragma，我們能在 HLS 中對 FPGA 上運行的核心做運算加速。

3. Experimental Results

3-1. 線性層量化

3.1.1. 8-bit PTQ 量化

我們對線性層的權重做 8-bit PTQ（Post-Training Quantization）。最終把 8-bit PTQ 的困惑度降至 4.80，接近原始模型的 4.02。未來有機會再考慮實作更進一步的量化手段 [13]。

3.1.2. Weight 對稱、Activation 非對稱量化

將 Weight 與 Activation 同時做量化，不過兩者的數據分布不一樣，需要使用不同的量化技巧 [14]。以下附上我們多次實驗後的數據（表 3）。當 Weight 對稱、Activation 非對稱量化時，可以達到兩者都量化時的最佳結果。推測是因為 weight 矩陣數值分布以 0 為中心，activation 的數值分布較散亂。另外，百分位量化部份我們做了幾組百分位參數，表中以 99.9 百分位為例。過程中發現不僅需要額外參數、需要耗費時間調整百分位參數、也調不出比 4.80 更小的困惑度，因此我們決定不使用百分位量化。

表 4. 不同 Weight、Activation 量化組合的困惑度

Weight / Activation	不量化	對稱量化	非對稱量化	百分位量化 (99.9 百分位)
不量化	3.32	3.35	3.81	3.81
對稱量化	—	16.7	17.1	16.7
非對稱量化	—	3.53	5.73	5.73
百分位量化 (99.9 百分位)	—	3.77	6.21	6.78

3.1.3. lm_head 層不量化

在優化量化後模型困惑度時，我們實驗發現模型最後輸出前的線性層對精度很敏感，推測因為與選擇生成 token 的機率分布密切相關。所以我們選擇跳過 lm_head 層不做量化以大幅降低困惑度。經過以上三種方法，我們**最終實作的量化模型困惑度僅增長 0.78，卻能減少 46% 的記憶體需求。**

表 5. 量化模型與原始模型比較

	原始模型（FP16）	量化模型（線性層 INT8）
困惑度↓	4.02	4.80
參數記憶體需求	5.86 GB	3.18GB
線性層運算方式	FP16 矩陣乘法	INT8 矩陣乘法

3-2. 記憶體存取搭配 pragma 的優化

依據先前「線性層大量存取記憶體」的結論，我們在 HLS 階段特別針對線性層優化記憶體使用效率。也對有矩陣乘法的 Attention 層作優化。

3.2.1. Linear Layer

對線性層我們進行多次實驗（表 5），當 Weight 矩陣放四個 HBM、512bit per beats 時，我們能獲得最大加速效益。加速約 16X，且硬體使用量僅相較未做優化前約 2X~3X。因此我們將此最佳結果應用在模型中。以下解釋目前線性層優化後的最佳結果（fc1，耗時 11.253ms）。首先，我們預先將線性層所需的 weight 在 hidden size 方向上拆成四等份。推論時使用 AXI burst、512 bits per beat 的方式，從四個 HBM 同步將 weight 逐 row 讀取到本地的四組 BRAM buffer 上。接著，將運算路徑複製四組，每組使用兩份運算單元處理資料，即可同時使用八條路徑運算讀入的 weight，搭配適當 pipeline 能夠大幅提升 linear layer 運行效率（圖 12）。

表 6. 以 fc1 線性層為例，展示優化實驗結果

簡介	耗時	硬體用量
Weight 矩陣放同個 HBM 8bit per beats	176.124ms	LUT 5,288 / Register 8,917 / BRAM 29 / URAM 0 / DSP 8
Weight 矩陣放四個 HBM 8bit per beats	45.129ms	LUT 10,480 / Register 16,997 / BRAM 62 / URAM 0 / DSP 11
Weight 矩陣放同個 HBM 512bit per beats	43.936ms	LUT 7,055 / Register 10,039 / BRAM 44 / URAM 0 / DSP 9
Weight 矩陣放四個 HBM 512bit per beats	11.253ms	LUT 12,815 / Register 16,572 / BRAM 70 / URAM 0 / DSP 23

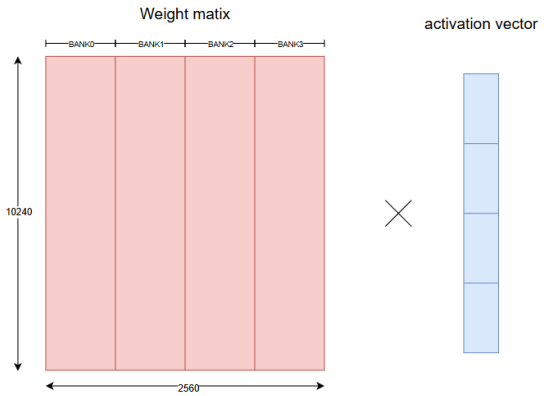

圖 9. Linear layer 優化方法示意圖，每個 bank 都使用兩份運算單元作乘加

3.2.2. Attention Layer

$$Attention(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

在這層進行了兩次矩陣乘法，所以若參數照順序存入記憶體，在乘上 V cache 時，矩陣乘法會跳記憶體地址順序的取讀 V cache，導致無法使用 burst 一次讀入資料。所以我們的優化方式是一次算完一個 token 與 V cache 的乘加運算。如下（圖 13），先做藍色區塊的乘加，再做紅色、黃色依此類推。如此一來可以不重新處理參數排列，巧妙利用 burst 一次讀入資料，提高讀取效率。搭配適當的 pipeline 與 unroll = 4，我們的優化結果與比較如下（表 6），目前得到最佳的 attention 層耗時為 11.39s / 500 tokens。

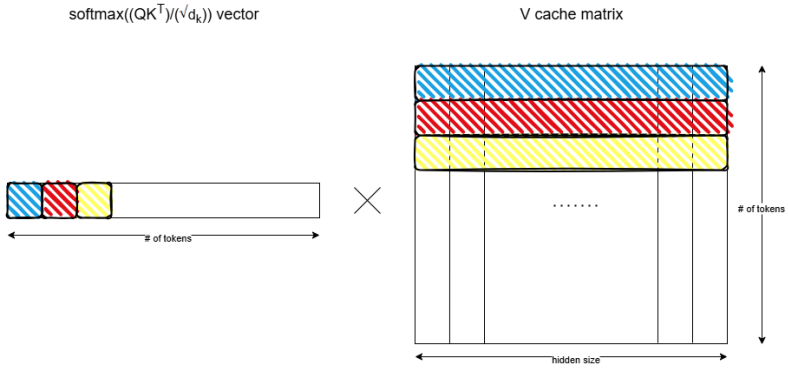

圖 10. V cache 讀取優化示意圖

表 7. Attention 層優化結果

簡介	耗時	硬體用量
未做 V cache 讀取優化	17.54s / 500 tokens	LUT: 13,298 / Register: 17,935 / BRAM: 32 / URAM: 0 / DSP: 44
做 V cache 讀取優化	11.39s / 500 tokens	LUT: 16,914 / Register: 16,442 / BRAM: 38 / URAM: 0 / DSP: 67

3-3. Phi-2 最終推論時間與功耗比較

最終我們的架構達成的推論速度能比 C++ 模型快 14x。且雖然經 PyTorch 優化的 CPU 架構在一般語言模型推理中已可取得不錯效能（例如在多核心環境下約 0.20 tokens／秒），但在我們的測試中，利用 i7-14900k CPU 搭配 Alveo U280 FPGA 的混合架構，針對 Phi-2 模型實作專用加速器後，吞吐提升至 0.82 tokens／秒，約為經過 PyTorch 優化之 CPU 的 4.1 倍。此結果顯示在 CPU 已高度優化的情況下，加入硬體加速仍具明顯效益。儘管尚未達 GPU 所展現之高吞吐（約 9.7 tokens／秒）水準，但因功耗僅為約 11.8 W，是所有比對對象中最低的，故在功率受限或邊緣部署應用中具備實用價值，成功驗證此架構於邊緣部署語言模型的可行性。

表 8. 不同平台的推論時間與耗能結果比較

Device Name	[1]TPS	功耗
Colab CPU (optimized by PyTorch)	0.20 (1x)	---[2]
Colab T4 GPU (optimized by PyTorch)	**9.73 (48.7x)**	70.15W
i7-10700 CPU (by C++)	0.059 (0.3x)	65 W
Our architecture (i7-14900k CPU + Alveo U280)	0.82 (4.1x)	**11.8W**

[1] TPS(token per second) = (prompt tokens + generated tokens) / execution time

[2] Google Colab 的 CPU 能耗資訊並未公開

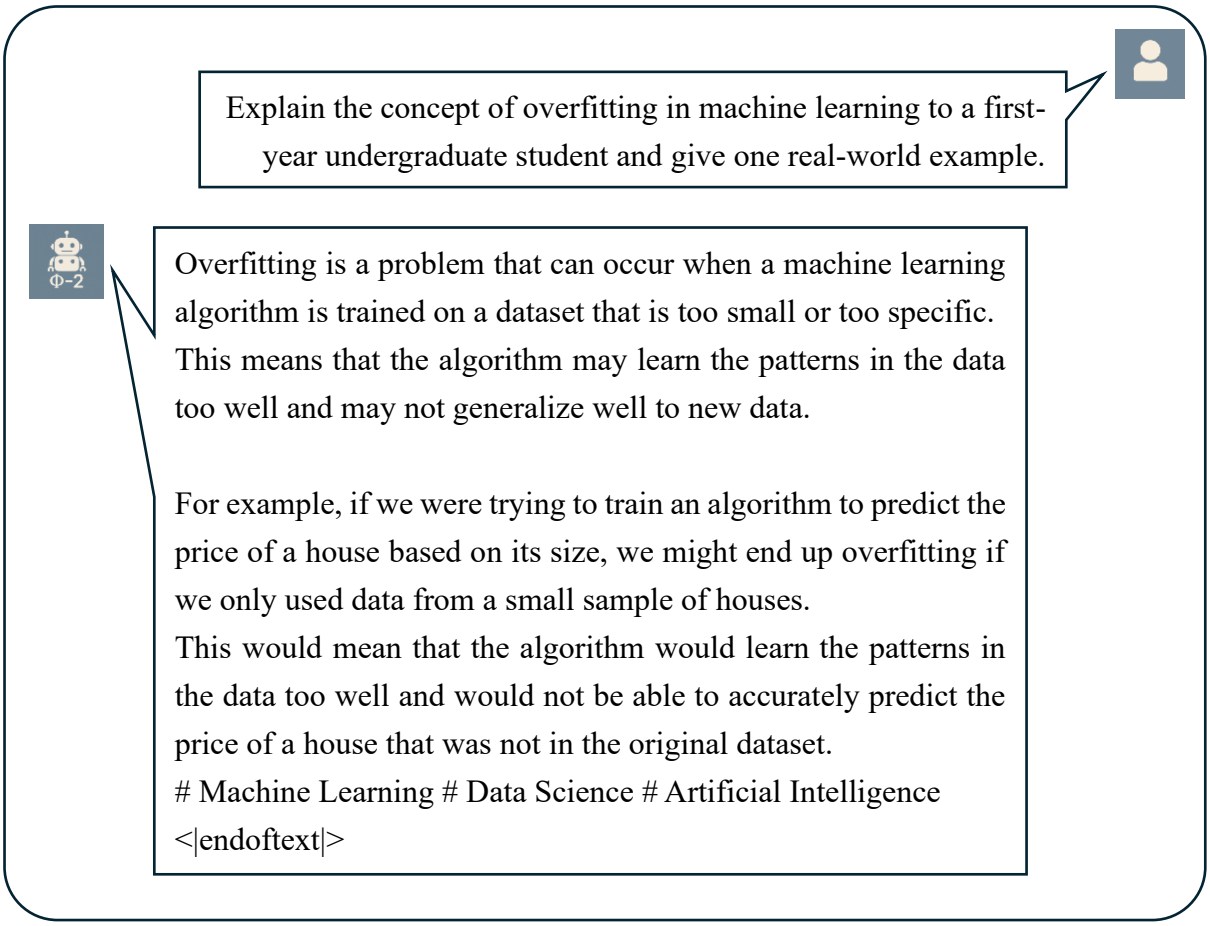

圖 11. Response of Phi2 on Alveo U280 (AMD Xilinx)

4. Conclusion

4-1. 結論

本專題針對 Phi-2 模型，在 Xilinx Alveo U280 FPGA 上採用 HLS 方法進行硬體加速，並結合 8 位元權重與激活量化、整數運算、記憶體配置、以及重構的 C++ 實作與硬體並行化排程。實驗結果顯示：在平均**僅消耗約 11.8 W 功率**下，我們架構的推論吞吐達成 **0.82 tokens／秒 (TPS)**，**比經過 PyTorch 優化的 CPU 架構快約 4.1 倍**。

儘管相較於 GPU (9.73 TPS) 間仍存在大幅差距，但我們的設計僅需原本模型 46% 的記憶體空間，在功耗受限或邊緣應用場景具備明顯優勢。此次研究證明，透過量化、HLS 管線化設計與 FPGA 加速，語言模型於 CPU+FPGA 架構上進行推論是可行且具實用價值。未來工作可進一步優化 FPGA 併行度、資料搬移與 HBM 頻寬利用，以期縮小與 GPU 間的效能落差。

5. Reference

[1] H. Xu, Y. Li, and S. Ji, “LlamaF: An Efficient Llama2 Architecture Accelerator on Embedded FPGAs,”. Available: https://ieeexplore.ieee.org/document/10811385
[2] Andy He, Darren Key, and Mason Bulling, “HLSTransform: Energy-Efficient Llama 2 Inference on FPGAs Via High Level Synthesis.” , Available: https://arxiv.org/abs/2405.00738
[3] LLaMA: Open and Efficient Foundation Language Models. Available: https://arxiv.org/abs/2302.13971
[4] M. Javaheripi and S. Bubeck, “Phi-2: The surprising power of small language models,” *Microsoft Research Blog*, Dec. 12, 2023. [Online]. Available: https://www.microsoft.com/en-us/research/blog/phi-2-the-surprising-power-of-small-language-models/
[5] HuggingFace Phi2, Available: https://huggingface.co/microsoft/phi-2
[6] Hugging Face, “Bitsandbytes,” *Transformers documentation*. [Online]. Available: https://huggingface.co/docs/transformers/quantization/bitsandbytes
[7] Hugging Face, “PrunaAI, microsoft-phi-2-GGUF-smashed”, 2025. [Online]. Available: https://huggingface.co/PrunaAI/microsoft-phi-2-GGUF-smashed
[8] Hugging Face, “GPTQ,” *Transformers documentation*, v4.53.1. [Online]. Available: https://huggingface.co/docs/transformers/v4.53.1/quantization/gptq
[9] Hugging Face, “GGUF,” *Hugging Face Hub documentation*. [Online]. Available: https://huggingface.co/docs/hub/gguf
[10] Hugging Face, ultrachat_200k, [Online]. Available: https://huggingface.co/datasets/HuggingFaceH4/ultrachat_200k
[11] AMD, “pragma HLS pipeline,” *Vitis High-Level Synthesis User Guide (UG1399)*, ver. 2025.1 English, Sep. 10, 2025. [Online]. Available: https://docs.amd.com/r/en-US/ug1399-vitis-hls/pragma-HLS-pipeline
[12] AMD, “pragma HLS unroll,” *Vitis High-Level Synthesis User Guide (UG1399)*, ver. 2025.1 English, Sep. 10, 2025. [Online]. Available: https://docs.amd.com/r/en-US/ug1399-vitis-hls/pragma-HLS-unroll
[13] J. Hasan, “Optimizing Large Language Models through Quantization: A Comparative Analysis of PTQ and QAT Techniques,” *arXiv* preprint arXiv:2411.06084, 2024. [Online]. Available: https://arxiv.org/abs/2411.06084
[14] WPG DADATONG, “模型壓縮原理與實踐：對稱量化、非對稱量化與基於百分位的量化,” *WPG DADATONG Blog*, Nov. 23, 2024. [Online]. Available: https://www.wpgdadatong.com/blog/detail/76111

6. 心得感想

在這次專題中，我們嘗試以 FPGA 加速大型語言模型 Phi-2，從模型量化、C++ 實作，到 HLS 硬體排程與實機驗證，都必須由我們自行完成。整個工作範圍極為龐

大，不僅跨越深度學習、C++ 系統架構、HLS 工程與 FPGA 平台優化，也充滿大量細節與挑戰。

在長達 11 個月的合作過程中，我們組員始終保持高度協作與良好溝通。前期我們花了大量時間閱讀論文並追蹤當前研究趨勢；同時透過 Coursera Machine Learning 與 Deep Learning 課程向 Andrew Ng 老師學習語言模型與機器學習的基礎知識；也向賴瑾教授在 NTU 2022 Fall 所開設的 AAHLS 課程影片中深入理解 HLS 的實務技術。這些前期扎實的學習，使我們能順利在中期完成模型量化策略與完整 C++ 模型架構。

暑假期間，我們全力投入推論加速的技術研究，包括 HBM 記憶體配置、kernel 平行化、排程設計等，並不斷嘗試不同的優化策略。整個專題期間，我們每週都與教授開會，討論進度、研究方向與技術問題；組員之間也維持每週 1～2 次的正式討論會，同時搭配線上線下數次非正式會議，確保任何問題都能即時溝通與解決。

透過這樣高密度的協作，我們逐步克服從 Python 模型到 C++ 再到 FPGA 的跨平台落差。量化模型的精度調整、integer-only 運算邏輯、Attention/MLP 結構對齊、HBM mapping、kernel 切分等，都必須反覆驗證與修正。在這些過程中，我深刻體會到跨領域整合的重要性，也學會如何從記憶體、帶寬、延遲、功耗等多面向思考硬體加速架構。

最終，我們成功在 U280 上達成 0.8 s/token 的推論速度，達到 PyTorch 優化 CPU 的 4.1 倍效能，功耗僅 11.8 W，同時量化後僅需原始 46% 的記憶體空間，證明 CPU+FPGA 架構在低功耗 LLM 推論上的可行性，也為未來的邊緣部署應用帶來實際價值。

回顧這段經驗，它不只是一個專題，而是一個真正完整的工程專案，更帶給我歸屬感。從基礎知識學習、模型理解、系統設計，到硬體架構與加速器實作，如同牙牙學語時的我們在父母的懷抱中，每一步都讓我獲得珍貴的成長。每次碰到專題就讓離家百里的我們感到家的溫暖，從教授的諄諄教誨想到不厭其煩在旁叮嚀的父母，從學長姐適時的提點想到在前方引領我們的兄姊，有些人也讓我們想到家中不務正業的頑皮弟弟。我相信，這段跨領域、跨平台、長期合作的經驗，將成為我們未來投入 AI 加速器與硬體系統設計的重要基礎。