一、研究動機與目的

先前 AI 加速器主要以 CNN 和 DNN 框架為主，然而近年來 Transformer 崛起，Vision Transformer 也在影像任務中展現卓越成果，包括圖像分類、物件偵測和影像分割，尤其是 Swin Transformer，已經超越了眾多 CNN 模型，在 ImageNet 上達到了 84.5%的 Top-1 準確率，從而展示了 Transformer 中的注意力機制在電腦視覺領域的可行性和潛力。不過在邊緣設備上即時處理高解析度的影像通常對計算效能和延遲時間有更高的要求，因此，本專題選擇基於 attention 的輕量化模型 EMO 來做為開發框架，使得其可以在邊緣設備上應用。

然而，傳統的硬體設計方法要求工程師使用 Verilog、VHDL 等硬體描述語言在 RTL 層級進行設計，需要工程師理解底層硬體元件的操作，而使得程式碼難以閱讀和編寫，也較容易出現人為錯誤，導致開發時間拉長。此外，軟體工程師通常不具備硬體描述語言的專業知識，不易為自己的應用程式設計硬體，與硬體工程師之間又存在溝通成本。

高階綜合（High-Level Synthesis，HLS）技術可以在一定程度上解決這些問題。HLS 允許工程師使用高階編程語言如 C、C++、System C 進行硬體設計，HLS 工具將高階語言轉換為 RTL 描述，並根據使用者的指示生成不同優化效果的 RTL 程式碼。

使用 HLS 方法設計硬體具有以下優點：

- 使用高階語言編寫：工程師只需使用高階語言編寫要實現的功能，程式碼更直覺，可以專注於功能和演算法的優化。
- 易讀易修改：HLS 生成的程式碼通常更容易閱讀、修改和除錯。
- 優化和靈活性：HLS 工具可以根據不同指示生成經過優化的 RTL 程式碼，提供了更大的設計靈活性和探索空間。
- 降低硬體設計門檻：雖然沒有傳統硬體設計的經驗，但只要對硬體有足夠的認識，軟體工程師也能為自己的應用程式設計硬體加速器。

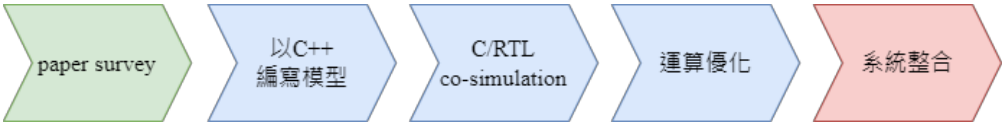


圖 1: 專題實作流程與規劃

在本次專題中我們使用了 HLS 方式來實作 attention-based 的影像辨識加速系統。神經網路加速器的設計本應十分複雜，但透過 HLS 可以讓我們更快速的進行設計及驗證，並從合成的報告中對效能及資源的使用進行分析，從而找出瓶頸來對其修改、優化。

二、系統設計

2.1 影像辨識模型架構

在影像辨識的模型上，我們選擇了 Rethinking Mobile Block for Efficient Attention-based Models[1] 論文中所提出的 EMO 模型架構。基於 Depth-Wise Convolution 的 Inverted Residual Block （IRB）被認為是輕量型 CNN 模型的基礎架構，並在 MobileNetv2 上展現了優異的表現，然而靜態 CNN 的 inductive bias 仍限制著純 CNN 的準確性，相較之下，vision transformer 則可以動態建模並從大規模數據集中進行學習，因此有顯著的改進。但因為 multi-head self-attention 的計算量相當龐大，導致不利於部署在邊緣設備或行動裝置上。這篇論文探討了如何將已在 CNN 取得成功的 IRB 架構應用在注意力模型上，以取得高效率且高性能的模型。

EMO 模型的核心思路為希望能夠結合 MobileNetv2 中高效率的 IRB 以及 Transformer 中高性能的 MHSA（Multi-Head Self-Attention）和 FNN （Feedforward Neural Network）模塊。Transformer 中每個 block 都具有兩個 residual，即為 MHSA 和 FNN 這兩個子模塊，這兩個子模塊與 IRB 有個相似結構，因而歸納設計出了只使用一個 residual 的 MMB，可接受 lambda 和 efficient operator（圖 2 中 F）從而結合 IRB、MHSA、FNN 這三個模塊。

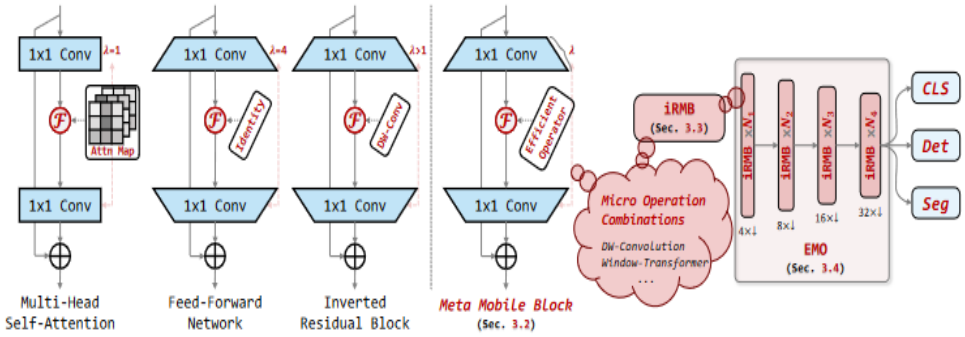


圖 2: Abstracted unified Meta-Mobile Block and ResNet-like EMO composed of only deduced iRMB.

[1] Jiangning Zhang, Xiangtai Li, Jian Li, Liang Liu, Zhucun Xue, Boshen Zhang, Zhengkai Jiang, Tianxin Huang, Yabiao Wang, and Chengjie Wang. Rethinking Mobile Block for Efficient Attention-based Models. *https://arxiv.org/abs/2301.01146*

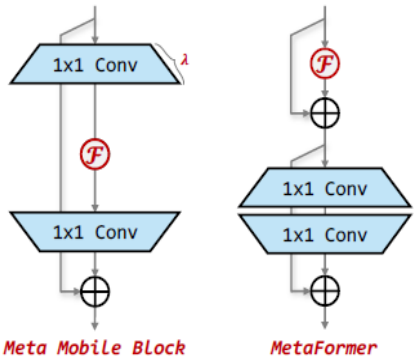


圖 3: MMB with MetaFormer

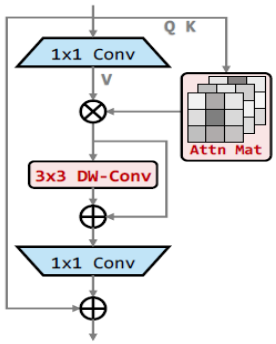


圖 4: Paradigm of iRMB

基於歸納出來的 MMB 架構，將特定 efficient operator F 設計進入其中，即成為了 EMO 模型架構之核心 iRMB（如圖 4）。在 F 的設計中，其結合 MHSA 和 DW-Conv，這麼做既可以吸收類似 CNN 的效率，用於建模局部特徵，也可以汲取類似 Transformer 的動態建模能力，用於學習遠距離的交互關係即全局特徵。

由一系列 iRMB 組成的 ResNet-like 模型 EMO，具有下列優點：

- 對於整體框架，EMO 僅由 iRMB 所組成，實作上簡潔有效。
- 對於特定模塊，iRMB 僅由標準 Convolution 和 MHSA 所組成，沒有其他複雜的運算符，並且因為 DW-Conv 帶來的好處，iRMB 可以透過 down-sampling operation 做採樣，並不需要 position embedding 來引入 inductive bias
- 對於參數的設置，可以逐步增加擴張率 lambda 以擴充 channels 獲得更多有用之資訊

總結來說，EMO 系列的模型能在維持相近的準確率的同時，能夠大幅減少計算量且僅由數個特定運算單元組成，為相對單純的模型架構，因此本專題最終採用 EMO-1M 來作為主架構，當模型擁有 1.3M 的參數量，可以在 ImageNet-1K 上的圖像分類任務中達到 Top-1: 71.5%的準確度。

2.2 加速器系統架構

Vitis 為一個支援異質計算的開發平台，支援使用 PS（Processing System）或 x86 compiler 來控制 PL（Programmable Logic）端，提供了一個框架，用於使用軟體和硬體元件的標準程式語言來開發和交付 FPGA 加速的應用程式，並可讓使用者利用 Vitis HLS 來以 C/C++編碼對硬體 kernel 進行開發。PS 執行 Linux OS，PL 則作為硬體加速和可編程 IO 的角色，可以輔助 PS 進行加速運算，或是合成高性能的 IO 介面。

我們的影像辨識系統將透過 Host Application 讀取圖片資料，並在經過前處理後放入 DRAM Buffer 作為輸入，而 HLS 生成的模型 IP 則在 PL 中運行，透過 AXI4 Master Interface 存取 PS DRAM，取得輸入和 kernel weight 等參數後進

行運算。運算過程中需要的大型 Buffer 也由 PS 的 DRAM 提供，會在 host.cpp 中預先分配連續的 DRAM 空間，並將記憶體位址作為參數傳遞給 PL，使得 PL 能利用 PS DRAM 的廣大空間，來降低 PL 中可用的高速記憶體 Block RAM（BRAM）和 Ultra RAM（URAM）的負擔，使得 PL 中的記憶體能專注於計算任務上，來達到資源使用的最佳化。

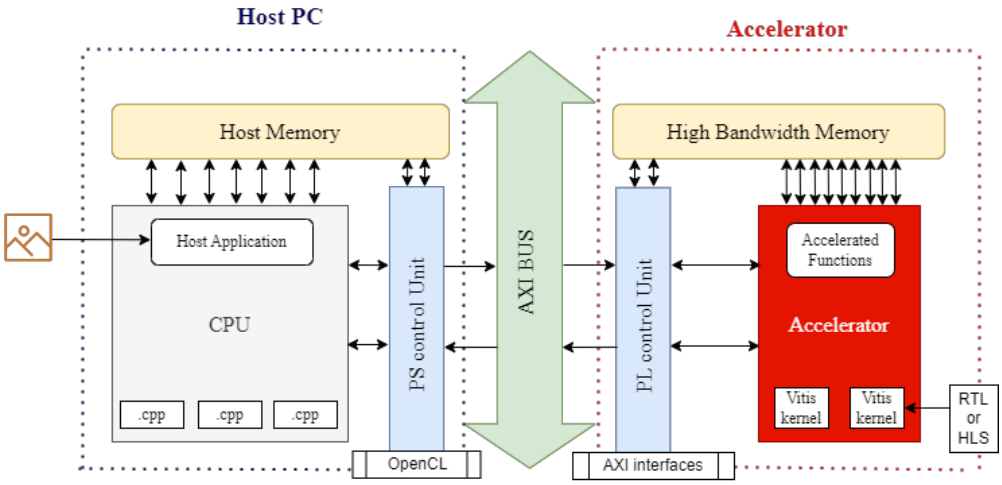


圖 5: Overall architecture of our accelerator.

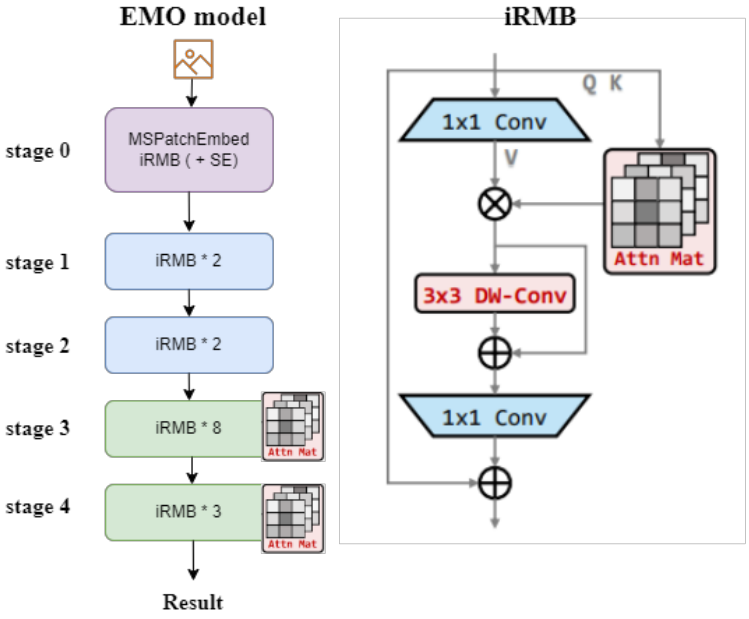


圖 6: EMO 模型於本專題應用之架構圖

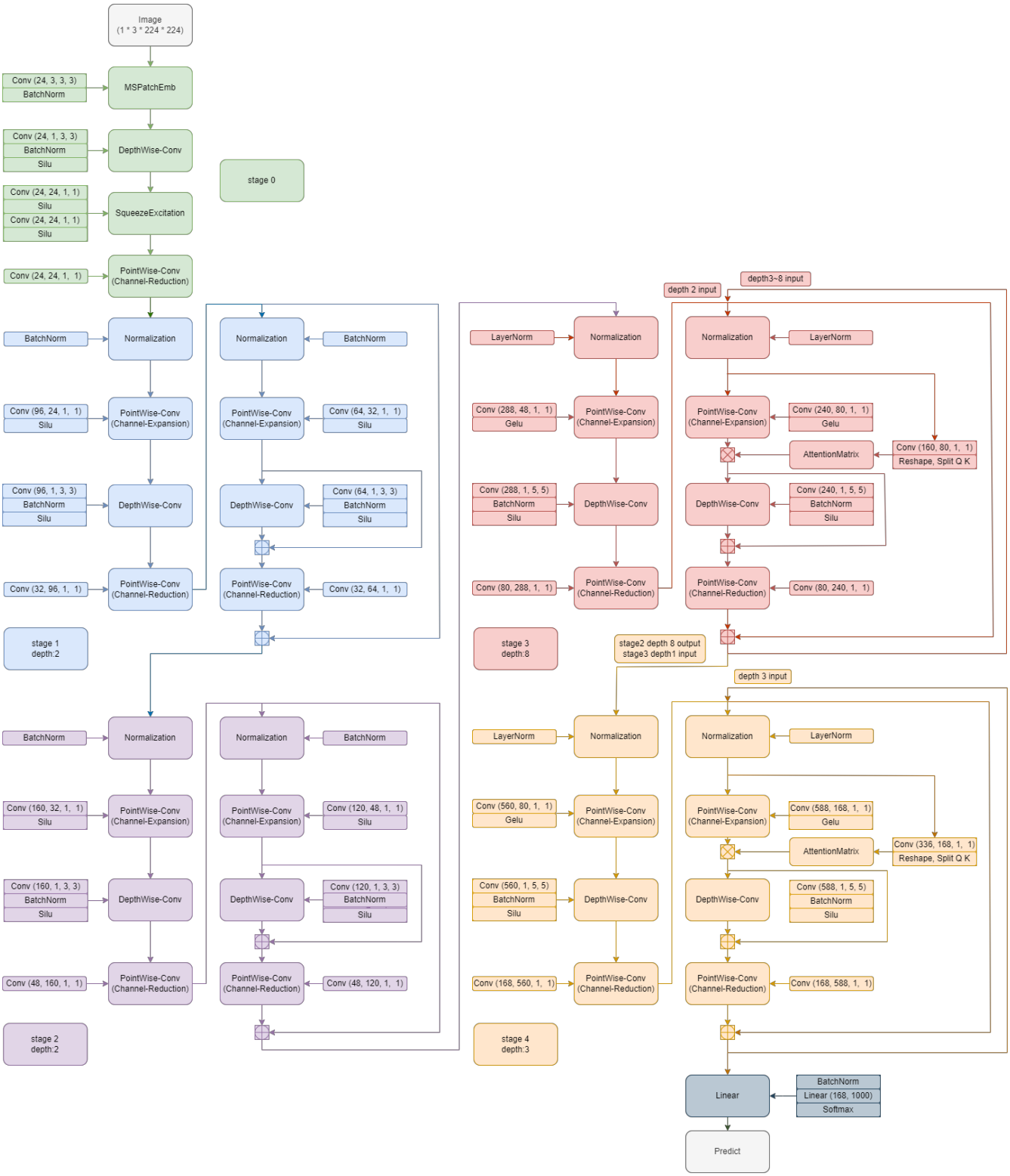


Image
(1 * 3 * 224 * 224)
Conv (24, 3, 3, 3)
BatchNorm
MSPatchEmb
Conv (24, 1, 3, 3)
BatchNorm
Silu
DepthWise-Conv
stage 0
Conv (24, 24, 1, 1)
Silu
Conv (24, 24, 1, 1)
Silu
SqueezeExcitation
Conv (24, 24, 1, 1)
PointWise-Conv
(Channel-Reduction)
BatchNorm
Normalization
Conv (96, 24, 1, 1)
Silu
PointWise-Conv
(Channel-Expansion)
Conv (96, 1, 3, 3)
BatchNorm
Silu
DepthWise-Conv
Conv (32, 96, 1, 1)
PointWise-Conv
(Channel-Reduction)
stage 1
depth:2
Normalization
BatchNorm
PointWise-Conv
(Channel-Expansion)
Conv (64, 32, 1, 1)
Silu
DepthWise-Conv
Conv (64, 1, 3, 3)
BatchNorm
Silu
PointWise-Conv
(Channel-Reduction)
Conv (32, 64, 1, 1)
BatchNorm
Normalization
Conv (160, 32, 1, 1)
Silu
PointWise-Conv
(Channel-Expansion)
Conv (160, 1, 3, 3)
BatchNorm
Silu
DepthWise-Conv
Conv (48, 160, 1, 1)
PointWise-Conv
(Channel-Reduction)
stage 2
depth:2
Normalization
BatchNorm
PointWise-Conv
(Channel-Expansion)
Conv (120, 48, 1, 1)
Silu
DepthWise-Conv
Conv (120, 1, 3, 3)
BatchNorm
Silu
PointWise-Conv
(Channel-Reduction)
Conv (48, 120, 1, 1)
depth 2 input
depth3~8 input
LayerNorm
Normalization
Conv (288, 48, 1, 1)
Gelu
PointWise-Conv
(Channel-Expansion)
Conv (288, 1, 5, 5)
BatchNorm
Silu
DepthWise-Conv
Conv (80, 288, 1, 1)
PointWise-Conv
(Channel-Reduction)
stage 3
depth:8
Normalization
LayerNorm
PointWise-Conv
(Channel-Expansion)
Conv (240, 80, 1, 1)
Gelu
AttentionMatrix
Conv (160, 80, 1, 1)
Reshape, Split Q K
DepthWise-Conv
Conv (240, 1, 5, 5)
BatchNorm
Silu
PointWise-Conv
(Channel-Reduction)
Conv (80, 240, 1, 1)
stage2 depth 8 output
stage3 depth1 input
depth 3 input
LayerNorm
Normalization
Conv (560, 80, 1, 1)
Gelu
PointWise-Conv
(Channel-Expansion)
Conv (560, 1, 5, 5)
BatchNorm
Silu
DepthWise-Conv
Conv (168, 560, 1, 1)
PointWise-Conv
(Channel-Reduction)
stage 4
depth:3
Normalization
LayerNorm
PointWise-Conv
(Channel-Expansion)
Conv (588, 168, 1, 1)
Gelu
AttentionMatrix
Conv (336, 168, 1, 1)
Reshape, Split Q K
DepthWise-Conv
Conv (588, 1, 5, 5)
BatchNorm
Silu
PointWise-Conv
(Channel-Reduction)
Conv (168, 588, 1, 1)
Linear
BatchNorm
Linear (168, 1000)
Softmax
Predict

圖 7: EMO 完整架構圖

三、系統開發流程

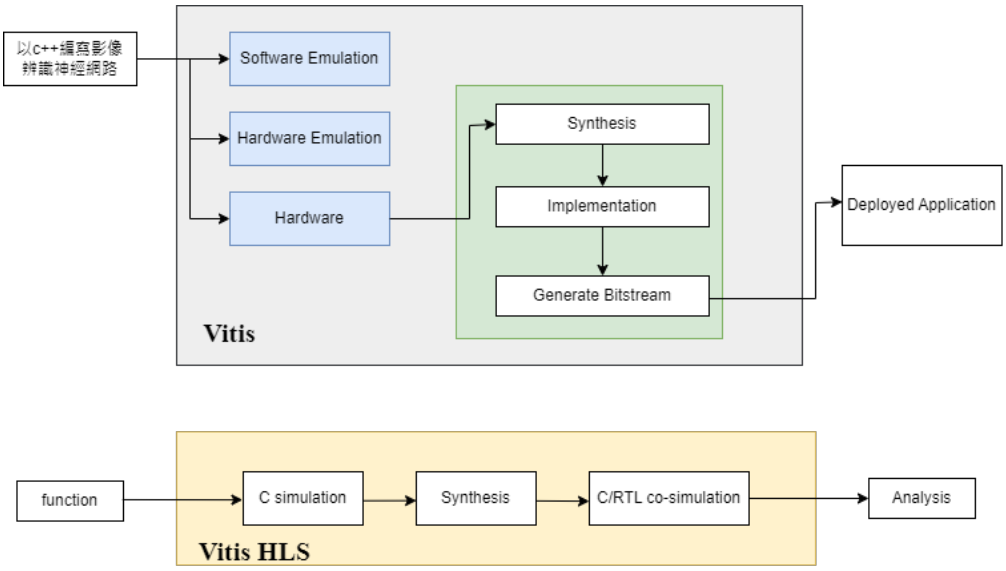


圖 8: Deployment Flow

3.1. 編寫以高階程式語言 C++設計之 EMO 模型。

3.2. 使用 Vitis 軟體加速應用程式來設計加速器

- 確定應用程式中需要加速的效能關鍵部分。
- 使用 Vitis Accelerated 函式庫設計加速器，本專題中使用 C、C++、OpenCL 來進行開發。
- 建置、分析和偵錯以驗證功能正確性並驗證效能目標是否已滿足。
  - 進行 Software Emulation。
  - 進行 Hardware Emulation。
  - 進行 synthesis、placement 和 routing 等步驟以產生 bitstream，並驗證 RTL code 之執行結果是否正確。

原先我們的設計是透過 Vitis 平台來做開發，並且使用 Vitis HLS 工具來協助合成以更快速的驗證功能正確性。然而，在合成時間過長和硬體限制的考量下，我們最後僅使用 Vitis HLS 來進行神經網路模型的合成，以及針對其合成報告來做效能上的分析。

四、系統實現與實驗

4.1 以 PyTorch 的 model 架構為基準編寫 C++ code

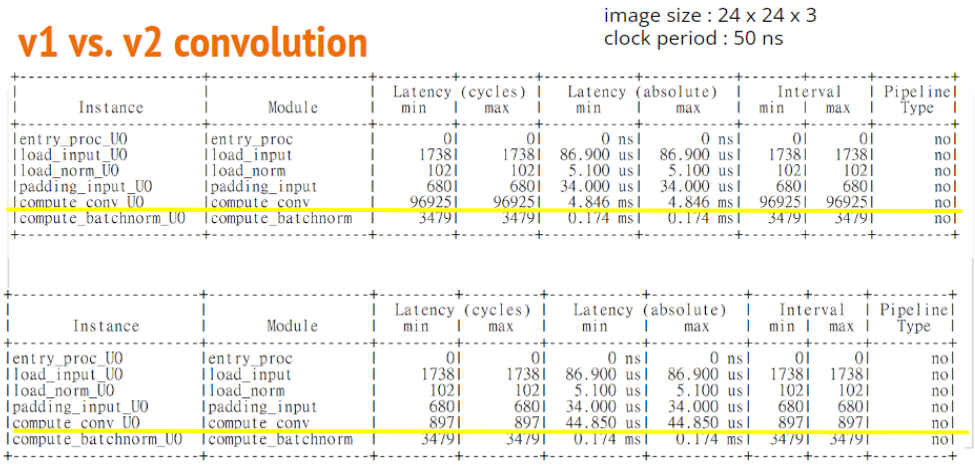
- 在初步了解 EMO 模型架構後，我們著手進行用 C++編寫出 model，並將較底層的運算單元，如 MSPatchEmb、Depth-Wise Convolution、Multi-head Self-Attention 等，化為 function 的形式，以便更有效率的架構模型和支援更廣泛的輸入資料。
- 由於 HLS 不支援 pointer-to-pointer 轉為 RTL 形式，無法傳送多維陣列，因此我們將所有不同維度的陣列都展開作為一維陣列處理。
- 為了硬體加速的平行化，我們將所有運算避開 input 和 output 為相同陣列之情形。

4.2 C++驗證

- 撰寫好底層的計算 function 後，使用 PyTorch 所提供的 module 先進行 unit 驗證，確認功能性正確後，將其整合成較大的 block。
- C++與 Python code 交互驗證正確後，使用 vitis_hls 來進行 C/RTL Cosimulation，確保在硬體電路上的功能表現與預期一致。
- 將 block 擴展成 layer，結合多個 function 測試從整體輸入到整體輸出其 dataflow 是否正確，並使用 EMO 原生模型程式碼與其所需環境進行 layer 驗證，以確認整體架構和功能。

4.3 優化

4.3.1 加速 convolution 計算



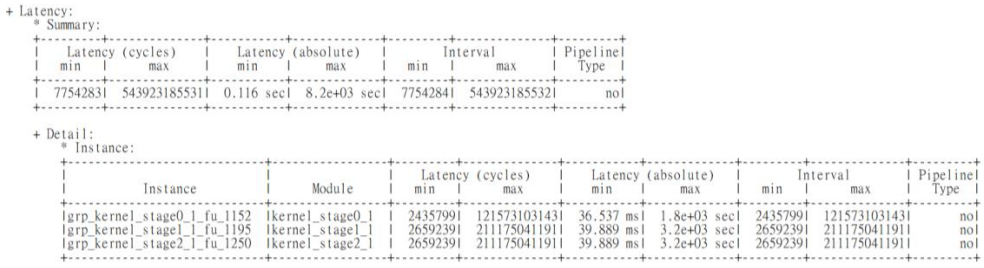
Instance	Module	Latency (cycles) min	Latency (cycles) max	Latency (absolute) min	Latency (absolute) max	Interval min	Interval max	Pipeline Type
entry_proc_U0	entry_proc	0	0	0 ns	0 ns	0	0	no
load_input_U0	load_input	17381	17381	86.900 us	86.900 us	17381	17381	no
load_norm_U0	load_norm	1021	1021	5.100 us	5.100 us	1021	1021	no
padding_input_U0	padding_input	6801	6801	34.000 us	34.000 us	6801	6801	no
compute_conv_U0	compute_conv	969251	969251	4.846 ms	4.846 ms	969251	969251	no
compute_batchnorm_U0	compute_batchnorm	34791	34791	0.174 ms	0.174 ms	34791	34791	no

Instance	Module	Latency (cycles) min	Latency (cycles) max	Latency (absolute) min	Latency (absolute) max	Interval min	Interval max	Pipeline Type
entry_proc_U0	entry_proc	0	0	0 ns	0 ns	0	0	no
load_input_U0	load_input	17381	17381	86.900 us	86.900 us	17381	17381	no
load_norm_U0	load_norm	1021	1021	5.100 us	5.100 us	1021	1021	no
padding_input_U0	padding_input	6801	6801	34.000 us	34.000 us	6801	6801	no
compute_conv_U0	compute_conv	8971	8971	44.850 us	44.850 us	8971	8971	no
compute_batchnorm_U0	compute_batchnorm	34791	34791	0.174 ms	0.174 ms	34791	34791	no

圖 9: Comparison of results among different convolution methods

在實作過程中我們發現 convolution 運算為最主要的運算之一且會消耗大量的資源，其延遲時間更是需要重視的瓶頸。我們嘗試調整 for loop 的順序，並加入#pragma HLS UNROLL 優化指令，將最內層的 for loop 展開來做平行化運算，成功大幅降低了 convolution 計算上的延遲時間。

4.3.2 內聯架構



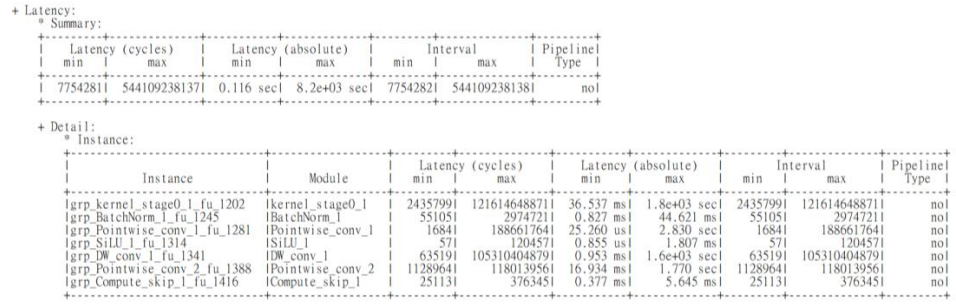
+ Latency:
  * Summary:

Latency (cycles)		Latency (absolute)		Interval		Pipeline
min	max	min	max	min	max	Type
77542831	5439231855311	0.116 sec	8.2e+03 sec	77542841	5439231855321	no

  + Detail:
    * Instance:

Instance	Module	Latency (cycles)		Latency (absolute)		Interval		Pipeline
		min	max	min	max	min	max	Type
grp_kernel_stage0_1_fu_1152	kernel_stage0_1	24357991	12157310314311	36.537 ms	1.8e+03 sec	24357991	12157310314311	no
grp_kernel_stage1_1_fu_1195	kernel_stage1_1	26592391	21117504119111	39.889 ms	3.2e+03 sec	26592391	21117504119111	no
grp_kernel_stage2_1_fu_1250	kernel_stage2_1	26592391	21117504119111	39.889 ms	3.2e+03 sec	26592391	21117504119111	no

圖 10: synthesis report with HLS inline off



+ Latency:
  * Summary:

Latency (cycles)		Latency (absolute)		Interval		Pipeline
min	max	min	max	min	max	Type
77542811	5441092381371	0.116 sec	8.2e+03 sec	77542821	5441092381381	no

  + Detail:
    * Instance:

Instance	Module	Latency (cycles)		Latency (absolute)		Interval		Pipeline
		min	max	min	max	min	max	Type
grp_kernel_stage0_1_fu_1202	kernel_stage0_1	24357991	12161464887111	36.537 ms	1.8e+03 sec	24357991	12161464887111	no
grp_BatchNorm_1_fu_1245	BatchNorm_1	551051	29747211	0.827 ms	44.621 ms	551051	29747211	no
grp_Pointwise_conv_1_fu_1281	Pointwise_conv_1	16841	1886617641	25.260 us	2.830 sec	16841	1886617641	no
grp_SiLU_1_fu_1314	SiLU_1	571	1204571	0.855 us	1.807 ms	571	1204571	no
grp_DW_conv_1_fu_1341	DW_conv_1	635191	1053104048791	0.953 ms	1.6e+03 sec	635191	1053104048791	no
grp_Pointwise_conv_2_fu_1388	Pointwise_conv_2	11289641	1180139561	16.934 ms	1.770 sec	11289641	1180139561	no
grp_Compute_skip_1_fu_1416	Compute_skip_1	251131	3763451	0.377 ms	5.645 ms	251131	3763451	no

圖 11: synthesis report with default settings

在研究 synthesis 報告時，我們注意到神經網路模型的 stage（module）層級不見了，因此發現 HLS 會自動做 inline，也就是把被調用的函數合併到調用他的函數中，以消除函數調用的開銷與時間，然而實際上內聯可能增加複雜性和合成時間，我們在把 inline 關掉後降低 186052606 個 cycles（max），也就是 2,790.789 milliseconds 的延遲。

4.3.3 Burst Read

這是 Vitis HLS 高階合成工具中，對 I/O 時間有最大助益的功能之一。以下為概念示意圖：

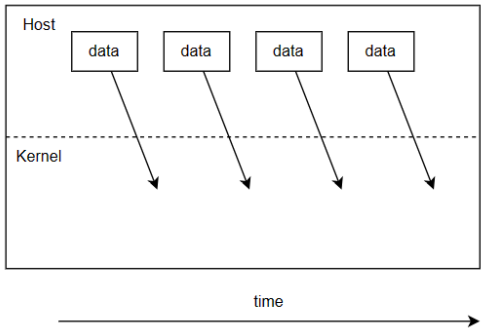


圖 12: host reads without pipeline burst

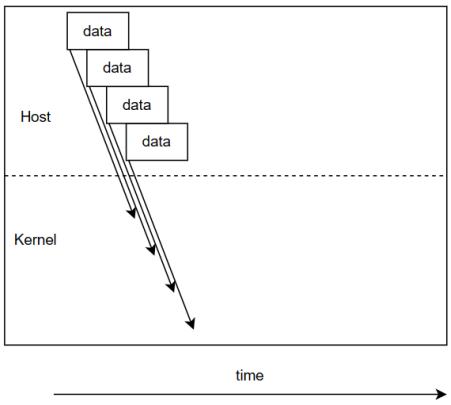


圖 13: host reads with pipeline burst

使用 burst read 時，Vitis HLS 會將連續的一段資料迅速且連貫的從 host 端讀入，而非各筆資料獨立讀入並處理完才讀下一筆；寫回 host 端時亦然。這樣一來可以大量減少跨裝置溝通所需的時間成本。在讀寫迴圈中，Vitis HLS 會嘗試自動判斷是否有符合這種讀寫方法的區塊，並自動帶入 burst read/write 功能，成立條件如下：[2]

一、讀寫來源為單向連續之資料
二、讀寫為連續之記憶體
三、讀寫不交錯
四、讀寫資料筆數為已知數
五、前後資料不具相互依賴性

在構成以上要素的狀況下，Vitis HLS 就會自動將讀寫資料的程序進行同步處理。而為避免運算過程中有資料上的依賴關係出現，我們在架構上使用 load/compute/store 的寫法，也就是在讀入時，設計一個函數專為了將 host 端資料讀進 kernel 本地端的記憶體（load），於本地端的記憶體中完成運算後（compute），再一併將所有本地端資料寫回 host（store）。這除了讀寫時能使用 burst read/write，也能最大化 pipeline 優化的成效。

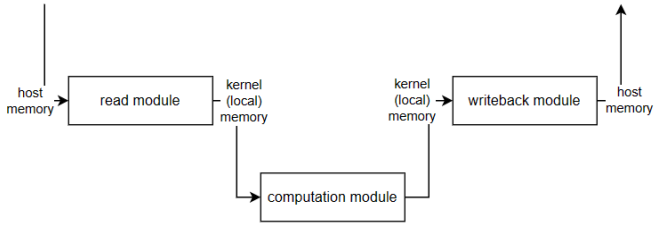


圖 14: load/compute/store 架構

4.4 實驗

4.4.1 ReLU / SiLU / GELU 效能比較

ReLU：$LU(x) = \max(0, x)$

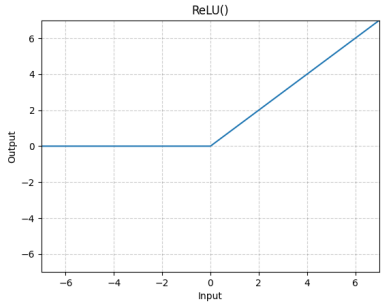


圖 15: ReLU 輸出曲線圖

[2] https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Preconditions-and-Limitations-of-Burst-Transfer

ReLU 在引入神經網路時，可以帶來一定的稀疏性，避免梯度消失的問題。而相比於 sigmoid，由於稀疏性，時間和空間複雜度較低，且不涉及成本更高的指數運算，使得 ReLU 比 sigmoid 的訓練更快，收斂更快。

SiLU：$SiLU(x) = x * \sigma(x)$, where $\sigma(x)$ is the logistic sigmoid.

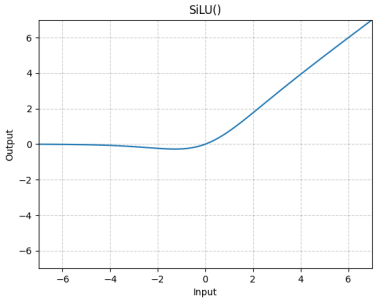


圖 16: SiLU 輸出曲線圖

SiLU 為 Sigmoid 和 ReLU 的改進版，具有較為平滑的輸出，因此不僅保留了 ReLU 的優點，又能解決 ReLU 在負數區段的梯度全部為零的缺點，在深層模型的效果優於 ReLU

GELU：$GELU(x) = x * \Phi(x) = \frac{x}{2} * \left(1 + \text{erf}\left(x/\sqrt{2}\right)\right)$

where $erf(x) = 1 - 2Q\left(\sqrt{2}x\right) = \frac{2}{\sqrt{\pi}}\int_0^x e^{-x^2}\,dx$

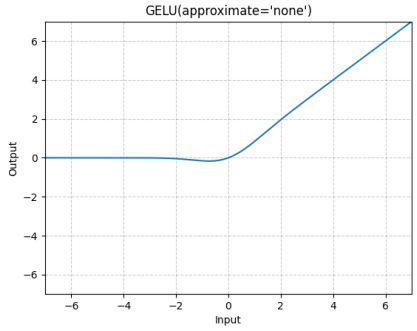


圖 17: GELU 輸出曲線圖

GELU 結合了 dropout，zoneout，和 ReLU 的一些性質所構造而成，在激活中加入了隨機正則化的思想，其輸出在 x 大於 0 時同樣維持為 x，而在負數區段的導數也仍為連續，使得在訓練過程中可以更容易地傳播梯

度。目前 GELU 在 NLP 領域表現較佳，尤其在 Transformer 的模型中更是優秀。

以下為本專題實驗中，觀察到模型配合不同 activation function 嘗試所呈現之效能：

表 1: 原始模型替換 Activation 的 Top-1 和 Top-5 Accuracy 比較

Acc\Activation	**SiLU & GELU**	**ReLU 替換 SiLU**	**ReLU 替換 GELU**	**ReLU 替換全部**
Top-1 Acc	71.48%	0.17%	1.24%	0.11%
Top-5 Acc	90.37%	0.79%	5.09%	0.51%

在硬體合成上，因為 SiLU 和 GELU 都有涉及 exponential 的計算，甚至 GELU 還需要做 erf 的積分，兩者在硬體合成上都需要較長的時間，也耗費較多的計算資源，而 ReLU 簡單的運算可以達到更快的計算以及更佳的硬體效能表現。然而，根據我們在原始模型上替換 activation 的實驗，其模型表現的下降是無法接受的，所以不能為了追求較好的硬體效能而將 SiLU 和 GELU 替換成 ReLU。

4.4.2 型別轉換（Type casting）

在嘗試將我們的應用在 FPGA 上花費的記憶體容量降低的時候，我們考慮使用資料型別的轉換來實作。

我們首先嘗試將 int 數值轉換成 uint8 的型態，這樣我們的整數資料就能以 8 bits 的大小來呈現，同時因為我們要處理的是 RGB 三個 channel 表示的影像圖案，所以 uint8 能表現的 0~255 的數值都能含括所有的值域。

接著我們嘗試將 float 數值轉換成 fixed point 的型態，並將整數以上的位數以 9 bits 表示，其中第一個 bit 用以表示 sign，小數點以下的也用 8 bits 表示，這樣這些 float 的數值佔的記憶體就能縮小一半，同時因為我們的設計是以 float 數值的計算為主，所以會有很大程度的影響。不過在神經網路的計算過程中，數字會有值域範圍的變化，如果是將整數位固定用 fixed point 的幾個 bits 來表示的話，很容易會有 overflow 的情形，所以我們在部分計算的設計上，又會將資料型別由 fixed point 轉換回原本的 floating point 表示方式，直到數值經過 normalization 之後，值域回歸到原本-256~256 之間的時候，我們才又轉換成 fixed point 的形式。

然而，使用 fixed point 仍然有其他不可避免的問題，也就是他只能保證小數點下幾位的精度，以我們的例子來說，我們只記小數點下 8 bits 的數值，更小精度的數值會直接被我們忽略，而在神經網路上的複雜計算更會放大這個誤差值的所在，導致模型預測正確性下降，我們的實驗結果為，如果我們針對圖片輸入做 int 記為 uint8，attention 計算 float 記為整數

位為 9 bits、小數位為 8 bits 的 fixed point 的 type casting 的話，我們的 BRAM 記憶體使用量將減少 37.68%、LUT 使用量減少 10.21%、LUTAsMEM 使用量將減少 23.46%、register 使用量減少 23.26%、DSP 使用量將增加 5.88%（推測為大量 type casting 衍生的計算）。

除此之外，我們也嘗試使用 half 的型態，這樣這些 float 的數值佔的記憶體也能縮小一半，加上 half 記錄數值的模式又類似於 float 的方式，所以在值域的範圍也相對 fixed point 來的大，在我們使用影像的 attention 處理中，也比較不會有 overflow 的情形發生，如果我們針對圖片輸入做 int 記為 uint8，attention 計算 float 記為整數位為 half 的 type casting 的話，我們的 BRAM 記憶體使用量將減少 42.31%、LUT 使用量減少 34.77%、LUTAsMEM 使用量將減少 13.87%、register 使用量減少 34.94%、DSP 使用量則改為減少 30.58%，整體情況比使用 fixed point 好得多。

4.5 困難與瓶頸

4.5.1. 合成時間長導致開發效率不佳

於 Vitis 上進行開發時，需要較長的時間來進行硬體合成。因此我們先使用 Vitis HLS 來進行 kernel 的驗證，以加速開發和除錯流程。然而我們終將處理 PS 和 PL 的溝通、資料傳輸、記憶體讀取等問題，則必須面對 Vitis 合成相當耗時的問題。

在深入研究 HLS 的合成概念後，我們發現雖然先前的優化架構大幅地降低了延遲，卻使得我們必須花費更多的時間來等待合成。因為較接近硬體層次，合成工具需要處理更多的複雜性和細節，來確保生成的硬體達到性能和資源利用率的目標，而導致合成所需的時間增加。為此，我們只能暫時性的妥協於開發速度，將優化指令移除，讓原先需要耗費七小時來合成的架構得以在兩小時內完成。最後也僅使用 HLS 的合成結果分析，並未實際部署於 FPGA 上。

4.5.2. BRAM 不敷使用

== Utilization Estimates

* Summary:

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	32	-
FIFO	-	-	5	42	-
Instance	2910	188	278325	373479	0
Memory	256	-	0	0	0
Multiplexer	-	-	-	54	-
Register	-	-	9	-	-
Total	3166	188	278339	373607	0
Available SLR	1344	2976	871680	435840	320
Utilization SLR (%)	235	6	31	85	0
Available	2688	5952	1743360	871680	640
Utilization (%)	117	3	15	42	0

== Utilization Estimates

* Summary:

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	32	-
FIFO	-	-	5	42	-
Instance	30	136	267767	380592	204
Memory	-	-	32	2400	96
Multiplexer	-	-	-	54	-
Register	-	-	9	-	-
Total	30	136	267813	383120	300
Available SLR	1344	2976	871680	435840	320
Utilization SLR (%)	2	4	30	87	93
Available	2688	5952	1743360	871680	640
Utilization (%)	1	2	15	43	46

圖 18: stage 0 接受 1x224x224x3 的輸入圖片，
右圖為指定使用 URAM 後的生成報告。

從 Vitis HLS 的生成報告中，我們發現了 BRAM 的使用將會超過可使用之資源而導致無法將模型放入 FPGA，但我們卻都沒有使用到 URAM。因此，我們使用 HLS bind_storage 的編譯指令來指定儲存器使用 URAM 來實現，使得 BRAM 的負擔轉移到 URAM 上，BRAM 便能專於提供給計算過程中的暫存和 FIFO 緩衝區等需求。

然而在後續的開發上，我們仍然遇到記憶體不足的問題，便再次重新調整架構，將所有暫存陣列及模型參數放置於 PS 端，使用 host 端的記憶體，讓 kernel 中的記憶體資源能被計算任務充分的利用。

五、效能評估與成果

最終我們有順利將 EMO 模型完整的各個階段串接進行改寫、輸出，並將其以 HLS 的形式完整合成，且驗證數據是正確的。

這意味著我們對架構的更動、加速並沒有改變原本模型的輸出，因此可以確定模型的準確度並沒有因為這些優化而降低。在這樣的前提之下，我們方進行模型調整的觀察、實驗，以維持正確性。然而，遺憾的是因為合成所需時間較長，我們尚未完善測試各種參數、變數之組合，而目前完整串接的版本運算速度上尚有相當大的進步空間（HLS 合成評估效能最低 0.18 秒、最高 38.22 秒，惟此為根據各模組中迴圈常數估計之最小最大值，僅供參考用途）。

故本專題在實現 HLS 合成神經網路模型的方面是相當成功，不過若要看出其於真實生活情境中的應用可行性，目前實驗數據尚不足以下定論，在完成其餘 I/O、平行處理的優化或是架構上的簡化後，再做進一步的功耗與效能分析會更具有應用層面上的參考價值。

六、結論與專題重要貢獻

在本次專題中，我們使用了 HLS 的方式來實作一個 Attention-based 的影像辨識加速系統。

神經網路加速器的設計本應十分複雜，而且如果沒有硬體背景實作起來更是困難。但透過 HLS，我們可以更容易的上手實作，並且比起 RTL 層級可以更快速的進行設計及驗證。

透過合成的報告，我們可以對效能及資源的使用進行分析，從而找出瓶頸來堆砌進行修改及優化。此外，HLS tool 提供了簡易的優化方式，透過 pragma 指令來對硬體的資源進行調用和限制。

雖然使用 HLS 來進行設計讓我們加速了開發流程，但正如本次專題中遇到最大的瓶頸為資料傳輸、讀寫的記憶體空間和 IO 時間，我們認為開發者若能有一定程度的硬體背景知識將會使得優化任務事半功倍，以利達到硬體資源使用效率的最佳化。

綜上所述，以下列出本專題中的主要貢獻：

- 對先前的 transformer 加速器的文獻進行探究，最終選擇一種新型的輕量模型 EMO（論文於 2023 發表），是一將 CNN 的 IRB 擴展到基於注意力的模型。
- 以 C++語言撰寫 EMO 模型。
- 使用 HLS 來將以 C++編寫之神經網路模型轉成 RTL code，並分析其效能。
- 透過 HLS 的指令優化，利用增加運算平行度、pipeline 等方式來減少運算的延遲，並透過 interface 接口與 array partition 的設計來增加吞吐量，以降低讀取 DRAM 的 I/O 時間。

七、未來展望

由於 HLS 合成時間較長，有一些優化未能及時實現於最終的設計之中。不過本專題在將來仍有許多可以持續進行的開發工作與延伸方向。

優化方面：

- 平行運算：嘗試沿著不相關且獨立的資料維度來進行劃分後，我們將可以對分區維度進行平行處理或更利於 HLS 做 pipeline，並且透過 Vitis HLS 提供的 burst read/write 可以減少 memory access 的延遲，在充分利用硬體資源的同時實現高效的加速運算。
- Tiling：原先的設計是透過 DRAM 讀取資料運算，因此在取得資料的過程中消耗不少時間。目前則在修正設計，預期將 memory buffer 切成可以放進 BRAM 的大小，藉此來減少存取 DRAM 的延遲且增加 data reuse。比起目前探討的 load/compute/store 架構，使用 tiling 能在保持讀取資料優勢的同時，降低記憶體需求，並減少硬體合成時間。
- 量化（quantization）：在深度學習中，量化是一個關鍵的技術，不僅可以縮小模型的大小，更是可以加速計算的運行。我們預計將 float 量化成 8 bit 大小，除了可以將模型縮小到近四分之一外，進行加速器的製作時可以使用更快速的 int8 計算，並且減少 memory access。我們將使用 pre-trained model 來實作量化，確認量化過後的準確度和原先不相差太多，再將 C++ code 改寫成 quantized 版本。

應用方面：我們將脫離本專題的 EMO 架構，使得加速器架構能支援具有更高辨識準確度之模型，並且加速模型推理時間以及降低硬體效能的需求，來應用在 real-time 的任務上，或得以部署於邊緣裝置上，如手機、感測器和穿戴式裝置等等。

八、團隊合作方式

本專題採用 GitHub Project 進行專案管理，使得五位成員可以良好並且有效率的開發，另外也記錄每次的開會內容，來追蹤開發進度和遇到的問題與解決方法。

專題前期，我們先各自研究適合用以加速的神經網路模型，同時熟悉 HLS 語法背後的硬體概念，熟悉硬體加速的原理。

中期則是以 C++去編寫 EMO 神經網路模型中的基礎運算，完成基礎運算所需的函式後，一組人負責架構的組成和分層測試，另一組人則負責量化與其他優化實驗。

後期則是藉由共同討論，來整合系統和對整體效能作分析與優化。

九、參考資料

[1] Ke Yu, Minguk Kim, and Jun Rim Choi. Memory-Tree Based Design of Optical Character Recognition in FPGA

[2] Jiangning Zhang, Xiangtai Li, Jian Li, Liang Liu, Zhucun Xue, Boshen Zhang, Zhengkai Jiang, Tianxin Huang, Yabiao Wang, and Chengjie Wang. Rethinking Mobile Block for Efficient Attention-based Models.

[3] A. Vaswani et al. Attention is all you need. *Proc. Adv. Neural Inf. Process. Syst., 2017, pp. 5998–6008.*

[4] H. Peng et al. Accelerating transformer-based deep learning models on FPGAs using column balanced block pruning. *Proc. 22nd Int. Symp. Qual. Electron. Design (ISQED), 2021, pp. 142–148.*

[5] X. Zhang, Y. Wu, P. Zhou, X. Tang, and J. Hu. Algorithm-hardware co-design of attention mechanism on FPGA devices. *ACM Trans. Embedded Comput. Syst., vol. 20, no. 5S, pp. 1–24, 2021.*

[6] Q. Li, X. Zhang, J. Xiong, W.-M. Hwu, and D. Chen. Efficient methods for mapping neural machine translator on FPGAs. *IEEE Trans. Parallel Distrib. Syst., vol. 32, no. 7, pp. 1866–1877, Jul. 2021.*

[7] S. Lu, M. Wang, S. Liang, J. Lin, and Z. Wang. Hardware accelerator for multi-head attention and position-wise feed-forward in the transformer. *Proc. IEEE 33rd Int. Syst. Chip Conf. (SOCC), 2020.*

[8] T. Wang et al. ViA: A Novel Vision-Transformer Accelerator Based on FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 41, no. 11, pp. 4088-4099, Nov. 2022, doi: 10.1109/TCAD.2022.3197489.*