國家科學及技術委員會補助

大專學生研究計畫研究成果報告

計畫名稱	:	Optimizing Large Language Model Inference on RISC-V Platforms Using TVM Compiler

報告類別： 成果報告

執行計畫學生： 莊翔鈞

學生計畫編號： NSTC 114-2813-C-194-066-E

研究期間： 114年07月01日至115年02月28日止，計8個月

指導教授： 陳鵬升

處理方式： 本計畫可公開查詢

執行單位： 國立中正大學資訊工程學系

中華民國 115年02月19日

目錄:

(一) 摘要

如何在資源受限的嵌入式設備 (如：RISC-V 平台) 上部署深度學習模型，有效利用硬體資源並提升模型推論效率，是我們所關注的議題。在本計畫中，我們針對 Large Language Model (LLM) inference 與 RISC-V 平台，透過 TVM compiler 優化 inference 的執行效能。我們使用 RVV 向量化指令重寫了模型的矩陣乘法運算子，充分地發揮此硬體的向量化處理能力。這麼做不僅能提升模型計算的吞吐量，還能減少運算時記憶體的訪問次數，大幅提升計算效率。最後再利用 TVM compiler 中的 BYOC (Bring Your Own Codegen) 技術，將優化後的運算子整合至模型的計算圖中。優化過的模型與原版相比，運行速度提升 **11** 倍。

(二) 研究動機與研究問題

隨著深度學習技術的快速發展，跨平台部署深度學習模型已成為一項重大的挑戰。由於每種硬體加速器 (如: GPU、NPU、XPU) 都有各自繁雜的軟體生態，即使是硬體供應商，為了在自家設備上順利運行深度神經網路 (DNN) 模型，也需要在軟體上投入巨大的開發和維護成本。為了改善這一問題，開發者迫切地需要自動化工具來生成針對硬體優化的程式碼，從而擺脫繁瑣的手工調整過程。

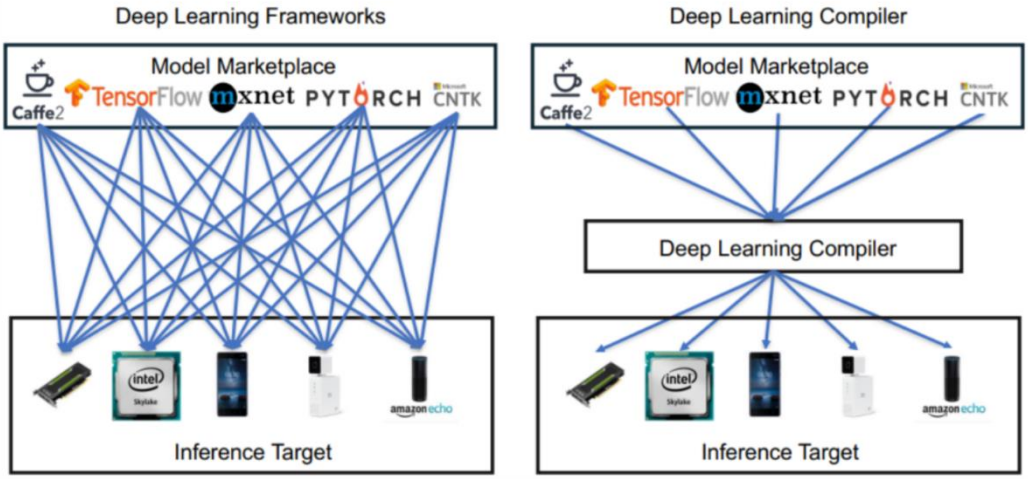

圖 1: 深度學習編譯器的優勢。

TVM deep learning compiler 是目前最受矚目的深度學習編譯器系統之一。做為一個開源框架，TVM 讓開發者接入自家硬體加速器，並打包成一個 runtime module，後續只需要關注自家程式碼生成器 (Codegen) 的工作即可，由於編譯器的其他模組能重複使用，開發與維護的成本也因此大幅降低。

而我們打算深入研究 TVM 中的 BYOC (Bring Your Own Codegen) 技術，它能讓我們把自己設計的外部運算子整合至 TVM 的編譯流程裡，從而提升模型的執行效能。我們將著重於以下幾個目標：

(1) 設計並實現支持 RISC-V RVV 指令集的自定義 Codegen。

(2) 充分利用 RVV 的向量化運算優勢，將深度學習模型中的運算節點有效地卸

載至 RISC-V 平台上。

(3) 驗證和量化模型推理的加速效果，並分析在效能提升方面，BYOC 技術所帶來的提升。

(三) TVM 運作流程圖

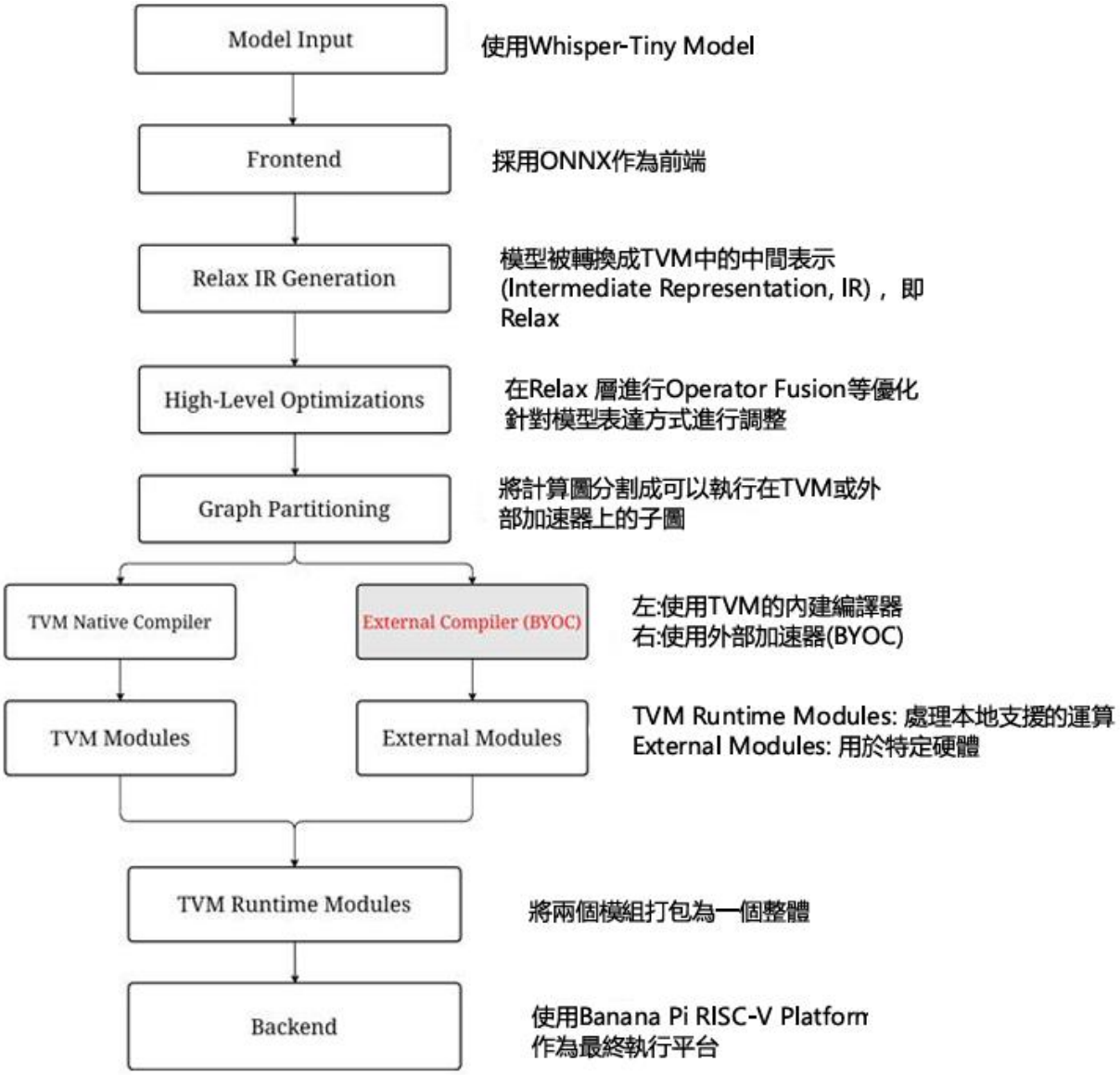

圖 2: TVM and BYOC 流程圖。

上圖為將深度學習模型部屬到 target 端的流程圖。其中紅色重點標記的 BYOC 是我們的專題著重研究的部分。

在這份流程圖裡，我們首先使用 **Whisper-tiny** 作為輸入模型，並採用 **ONNX** 作為前端。接著將模型轉換為 TVM 的中間表示（IR），即 Relax，並執行 Operator Fusion 等一系列高階優化。接下來，把計算圖拆分為可以在 TVM 或外部加速器上執行的子圖，以提高性能與適應不同的硬體環境。

對於可以在 TVM 上執行的子圖，我們採用 TVM 的內建編譯器進行加速；而需要外部加速器執行的子圖則是使用 BYOC 技術來整合外部加速器的編譯器進行加速。最後將兩部分打包成一個 runtime module，在嵌入式設備上運行。

(四) RVV 版矩陣乘法介紹

(1) 向量化計算與 SIMD:

向量化運算是把原本需要一條一條指令計算的「標量運算(scalar operation)」改成一次處理多個數據的方式。CPU / GPU 同時處理多個資料，能大幅提吞吐量和計算效率。而 SIMD (Single Instruction, Multiple Data)是典型的向量化做法，特別適合矩陣乘法這種計算量龐大、計算模式重複、資料依賴性低的運算。

(2) 矩陣分塊:

當矩陣過大，無法一次從記憶體搬入快取時，傳統矩陣乘法易出現頻繁的快取失誤。CPU 花大量時間在搬資料而非計算，因而成為效能瓶頸。

而分塊是將矩陣切割成多個能裝進快取的小區塊來計算，可減少對主記憶體的往返與快取失誤、提高資料重用率與快取命中率，進而提升吞吐量。

(3) RVV（RISC-V Vector）

RVV 是 RISC-V 的向量指令集，屬於 SIMD 家族，但與傳統 SIMD 的固定寬度計算相比，可彈性處理不同寬度的資料，有更強的可攜性和可擴充性。圖 3 的程式碼使用 RVV 指令實作了「沿輸出列向量化，用 A 分塊的純量與 B 分塊的向量做乘加，累加到 C 分塊」。

```
1. for (int i = 0; i < mc; ++i) {
2.         const float* A_row = Ablk + (size_t)i * (size_t)lda;
3.         float* C_row = Cblk + (size_t)i * (size_t)ldc;
4.         // Middle loop: vectorize across columns of B (and C)
5.         int col = 0;
6.         while (col < nc) {
7.             // Set vector length for remaining columns
8.             size_t vl = __riscv_vsetvl_e32m1((size_t)(nc - col));
9.             // Load accumulator from C (to support accumulation across
   K-tiles)
10.            vfloat32m1_t vacc = __riscv_vle32_v_f32m1(C_row + col,
   vl);
11.            // Inner loop: reduction over K
12.            for (int k = 0; k < kc; ++k) {
13.                // Broadcast A[i][k]
14.                float a_scalar = A_row[k];
15.                // UNIT-STRIDE LOAD: B[k][col:col+vl]
```

```
                // Position in Bp: k*nc + col
                const float* B_vec = Bp + (size_t)k * (size_t)nc +
   (size_t)col;
                vfloat32m1_t bv = __riscv_vle32_v_f32m1(B_vec, vl);
                // FMA: vacc += a_scalar * bv
                vacc = __riscv_vfmacc_vf_f32m1(vacc, a_scalar, bv, vl);
            }
            // Write back to C
            __riscv_vse32_v_f32m1(C_row + col, vacc, vl);
            col += (int)vl;
        }
    }
```

圖 3: RVV 指令的程式碼實例。

(4) B-packing:

在讀取原始大矩陣的子區塊時，雖然單一列的資料是連續的，但列與列之間在實體記憶體中往往存在巨大的跨度（Stride）。這種不連續性會破壞 spacial locality，所以我們在運算前先複製 B 矩陣的子區塊，並重組成一維的陣列，大幅降低記憶體延遲並最大化頻寬利用率。

(5) 包裝函數:

在分析模型的矩陣乘法節點後，我們發現 B 矩陣會出現兩種張量型態：

- **Batch×Batch（B 具批次維）**：多個 A 分別與多個 B 相乘，輸出多個 C。
- **Batch×Single（B 無批次維）**：多個 A 與同一個 B 相乘，輸出多個 C。

因此，我們在執行前會先檢查當前 B 的張量形狀，並呼叫對應的矩陣乘法實作（Batch×Batch 或 Batch×Single 版本），以確保遇到不同的張量型態都能以正確的參數數量傳入函數計算。

(五) 研究方法及步驟

(1) 研究設備

1. 欲優化之 LLM 模型：Whisper-tiny
2. 前端架構：ONNX
3. 目標平台：支援 RVV 的 RISC-V 平台
4. 使用設備：Banana Pi BPI F3

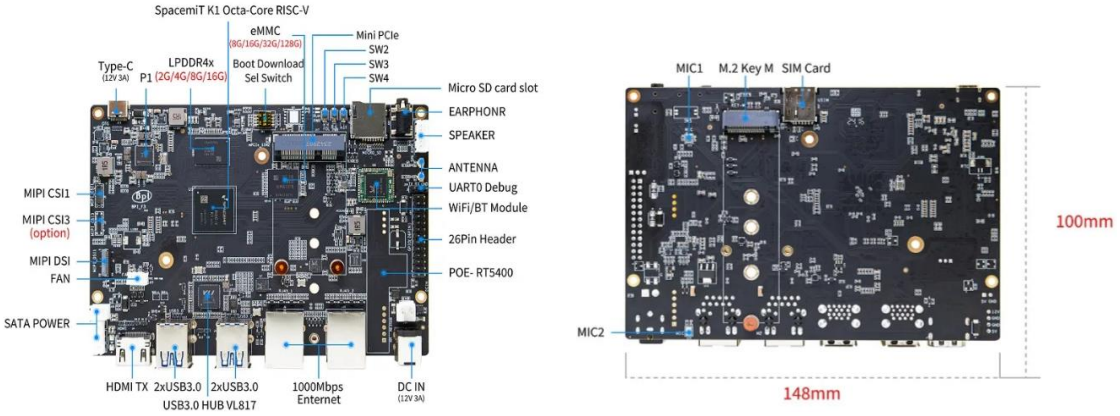

圖 4: Banana Pi BPI F3 的結構圖 (Reference from https://docs.banana-pi.org/en/BPI-F3/BananaPi_BPI-F3)。

表 1: Banana Pi BPI F3 的各項配置。

Feature	Specification
CPU	SpacemiT K1 8-core RISC-V chip with 2.0 TOPs AI computing power
Memory	2/4/8/16 GB LPDDR4 (Supports up to 16G LPDDR4)
Storage	8/16/32/128 GB eMMC flash (Optional 4M SPI NOR, 32M SPI NAND)
Networking	2x GbE Ethernet ports (supports PoE with add-on PoE HAT), 2.4G/5G WiFi and Bluetooth 4.2
USB	4x USB 3.0 Type-A HOST, 1x USB 2.0 Type-C OTG
PCIe	PCIe2.1 2-lane x 2, PCIeB 2-lane connect M.2 KEY M (Supports JMB582 expansion card to SATA), PCIEC 1-lane connect MINI PCIE
Other	26-pin header, reset, power, and burn buttons, power status LED

(2) 研究步驟

(a) x86 虛擬機環境建置與模型編譯

1. 環境準備

- 建立 Ubuntu 22.04 LTS 虛擬機(6 核心 CPU、20 GB 記憶體、100 GB 硬碟)。

2. 環境安裝與工具鏈建置

- 安裝模型編譯與 TVM 所需的相關套件與工具。
- 編譯並設定 RISC-V 交叉編譯工具鏈。

3. Runtime 與 Codegen 設定
   - 設定 Banana Pi 的 runtime 環境，並整合自訂的 codegen 元件至 TVM。
4. 模型準備
   - 下載 Whisper-tiny 模型（encoder、decoder、decoder with past）以供編譯。
5. 模型 cross-compile
   - 使用 TVM 與 RISC-V backend 將三個模型編譯為共享函式庫 (.so)。

(b) Banana Pi F3 環境建置與推論測試

1. 系統準備
   - 在 Banana Pi F3 (Bianbu 2.2，基於 Ubuntu 24.04) 上進行測試。
   - 確認硬體規格（Spacemit X60 處理器，8 核，約 4 GB 記憶體）。
2. 依賴套件與環境配置
   - 安裝必要的建置工具與 TVM runtime。
   - 完成 Banana Pi 上的 runtime 與 codegen 設定，確保能正確執行 TVM 編譯後的模型。
3. 優化核心編譯
   - 為 RISC-V 架構編譯自訂矩陣乘法核心(libmatmul.so)。
4. 模型準備
   - 下載 Whisper-tiny 的完整模型檔案，包含 tokenizer 與 vocab.json，以支援推論所需。
5. 部署交叉編譯模型
   - 將 x86 虛擬機編譯完成的.so 模型檔傳送至 Banana Pi F3。
6. 推論測試
   - 在 Banana Pi F3 上執行推論程式。
   - 紀錄評測資料，並驗證模型輸出是否符合 Whisper-tiny 的預期行為。

(六) 結果分析

以下表格為在 Banana Pi BPI F3 上使用 Whisper-tiny 模型分析約 30 秒的英語音訊，並生成文字所花的時間。輸出格式類似下圖:

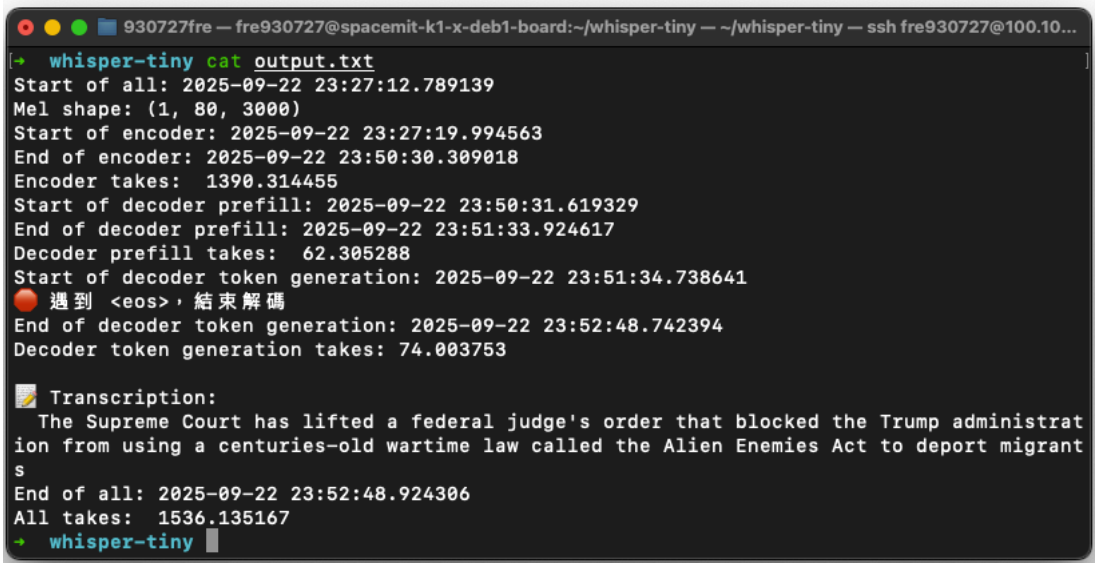

圖 5: 執行結果範例 (*注:此圖為 Baseline (tvm built-in)的執行結果)。

我們以 BYOC 是否使用 RVV 版矩陣乘法、O3 優化是否開啟做為操作變因，將四種輸出結果與原版比較，並整理成圖表進行分析:

表 2: 各種執行時間對比 (單位:秒)。

Config	Encoder (s)	Decoder Prefill (s)	Decoder (s)	Total (s)
Baseline (tvm built-in)	1390	62	74	1526
BYOC (classic matmul) -O0	1512	78	73	1663
BYOC (classic matmul) -O3	912	18	73	1003
BYOC (RVV matmul) -O0	518	50	75	643
BYOC (RVV matmul) -O3	60	3.24	75	138

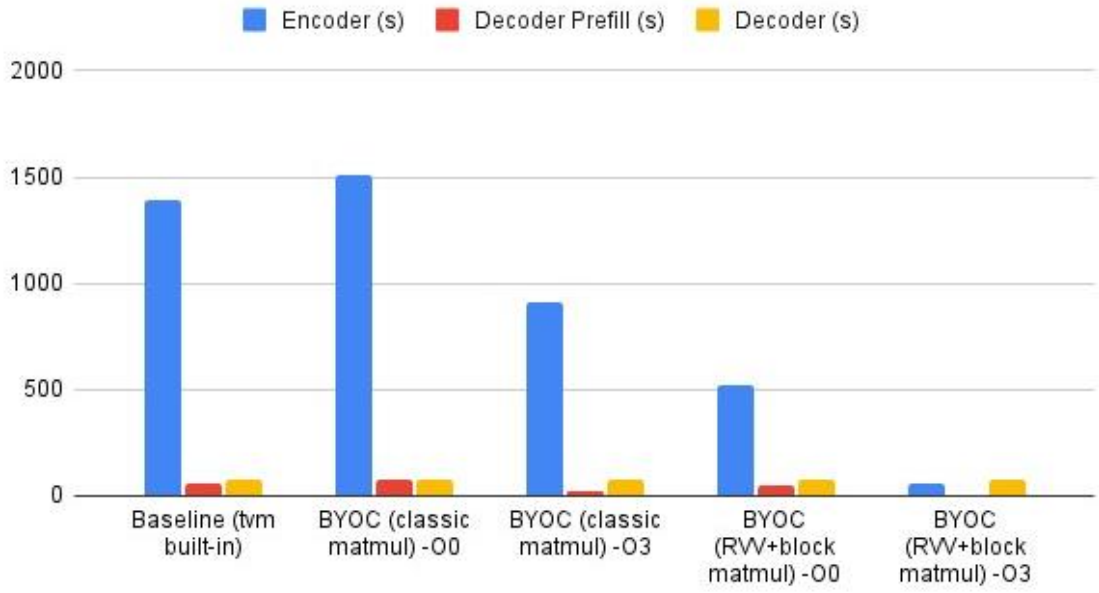

圖 6: 各項執行時間對比 (單位:秒)。

由以上數據可發現，無論是單獨使用 RVV 版矩陣乘法或開 O3 優化，與原版模型的計算速度相比都有不錯的提升，但當兩者兼用時提升幅度更為顯著，以下將根據上述圖表進行詳細的分析:

當我們採用經典矩陣乘法（classic matmul：時間複雜度 O (n^3)，未使用向量化計算、矩陣分塊），且未開啟 O3 優化時，在相同硬體與輸入規模下，可以發現此版本的矩陣乘法計算速度甚至比模型內部的矩陣乘法還要慢，說明這不是一個好的演算法；但當 O3 優化開啟時，即使用這種比原版差的演算法，執行時間卻能降至 65%，可見 O3 優化提供的迴圈轉換、指令選擇與記憶體存取排程對於模型的推理速度有巨大的幫助。

當我們使用 RVV 版矩陣乘法，且未開啟 O3 優化時，可以發現執行時間與原版相比降至 42%，可見向量化計算能有效提升吞吐量，加快矩陣乘法的計算速度。

當兩者兼用時，執行時間更是降到**原版的 9%**，原本模型在 encoder 這個步驟要花 1390 秒，優化後縮減到只要 60 秒，速度提升 23 倍；Decoder Prefill 也從 62 秒降至 3.24 秒，速度提升 19 倍，**總體運行速度提升 11 倍**。顯示 RVV 版矩陣乘法搭配編譯器的優化，能充分發揮向量化計算的優勢，因此出現遠優於原版的性能提升。

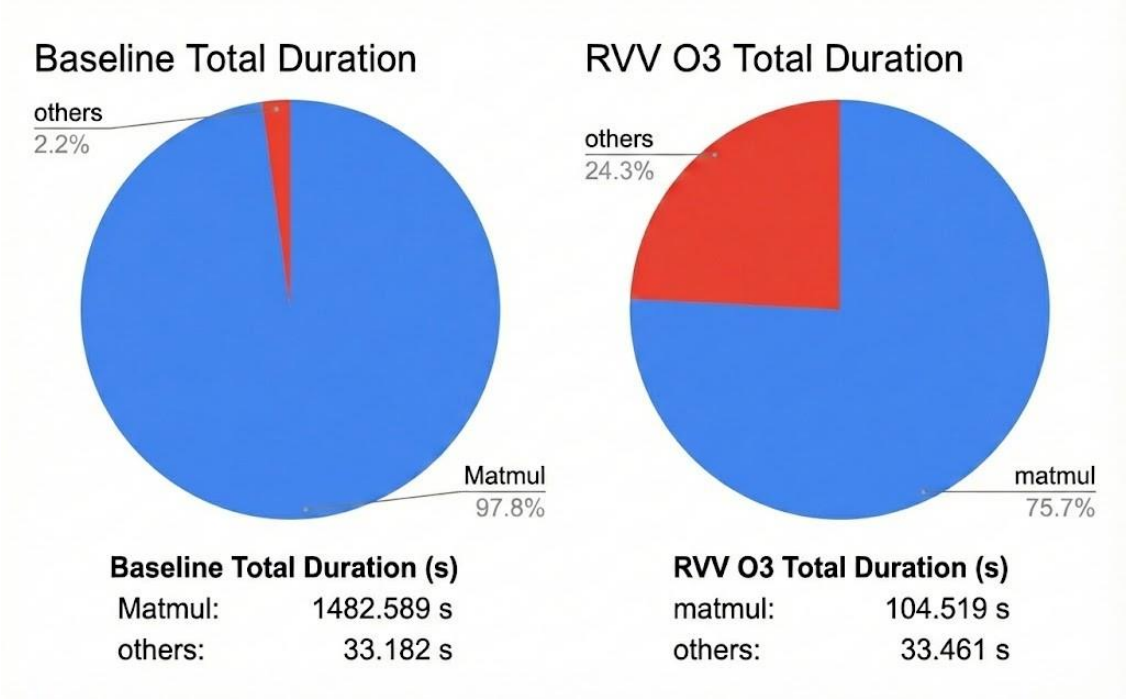

圖 7: 矩陣乘法運算子佔比。

如上圖，我們可以發現，由於優化了矩陣乘法，matmul 運算子在計算中的時間佔比也大幅降低，從 97.8%降至 75.7%，這代表我們成功解決了原本的效能瓶頸。

(七) 未來可改進的方向

目前的三個模型中，前兩個已經能在我們設計的 runtime 上執行，並成功使用 BYOC 導入自己設計的運算子。但 decoder_with_past.onnx 模型涉及 dynamic shape，礙於時間壓力，我們只能暫時放棄 decoder 部分的優化，未來可嘗試將 runtime 改寫成支援 dynamic shape 的版本，以得到更佳的效能提升。

此外在研究過程中，我們發現現有的文獻大多聚焦於 Relay IR，若能將這份使用 Relax IR 的研究經驗分享在開源社區，能為後續的開發者帶來更大幫助。

(八) 心路歷程與相關研究

這份專題真的做得一波三折，但我們也因此學到了很多。剛開始我們以為只要處理 BYOC 部分就好，但指導教授鼓勵我們全流程都跑一遍，於是我們就從模型編譯開始做起。

要學會怎麼編譯模型，第一步就是把模型轉成中間形式，但其實 TVM 有兩種中間形式，分別是靜態編譯 Relay IR 和動態編譯 Relax IR。教授要求我們使用較新的 Relax IR，但網路上有關的教程全是 Relay IR，和 Relax IR 有關的知識甚至連 TVM 官網都寫得不清不楚，要去他們團隊的 Github 研究原始碼才逐漸搞懂。

接下來就是設定板子的運行環境。當初選擇 Banana Pi BPI-F3 是因為它搭載的 SpacemiT K1 處理器支援 RISC-V RVV。但 Ubuntu 官方並未提供完整支援 RISC-V 平台的版本，經過一番查找後，我們改用從 Ubuntu 24.04 修改的 Bianbu OS 2.2。

設定好環境後，接著就是編寫 RVV 版矩陣乘法了，我們在 Github 上搜尋到了相關的程式碼，寫測資測試沒問題，但改寫各種接口並移植到 TVM 時卻發現編譯不過，一直發生溢位。我們花了將近一個月改程式碼、改模型，最後發現問題出在使用的模型節點型態是 fp16，改用 fp32 訓練的模型才能正常運行 RVV 版矩陣乘法。

但即使 RVV 版矩陣乘法能順利運行，使用 perf 跑測試時，卻發現效能比原版矩陣乘法差，cycle 數也明顯更多。我們做了許多嘗試，如翻轉矩陣、改用其它 RISC-V 指令(m1 改為 m2、m4)等等，但效果都不盡理想。後來查閱資料才發現要做矩陣分塊，也就是把大矩陣切分為可以完全放到快取中的小矩陣，這樣才能發揮 SIMD 的優勢。添加這一步驟後，果然 RVV 版矩陣乘法的效能得到了大幅的提升。

完成環境設定與自定運算子的測試後，我們著手將模型部署到板子上。但因 RISC-V 平台的預編譯工具與相關的套件不足，無法直接在板子上以 TVM 將 ONNX 模型編譯成可執行模組。我們一開始嘗試用 RPC 遠端執行，但連線與環境相依性問題仍然無法解決。幾番波折後，我們最終改用交叉編譯（cross-compilation），在主機以 riscv64-linux-gnu 工具鏈完成編譯，只將生成的.so 檔部署到板子上，避開嵌入式設備的效能限制。

在部署 TVM 時又遇到了 BYOC 的相容性問題，導致 runtime 始終無法找到 relax.Executable 對應的 loader。我們做了一系列的檢查，確認工具鏈設定正確、交叉編譯流程無誤，並排除了動態連結庫缺失的可能性，最後發現原因出在我們當初安

裝的 TVM 版本是該版本號的測試版，重新安裝同版本號的正式版後就順利解決了。上述問題全部解決後，我們成功在 Banana Pi BPI-F3 上運行含有我們自帶運算子的模型了。

(九) 參考文獻

[1] "Apache TVM Documentation", Apache, [link: https://tvm.apache.org/docs/index.html]

[2] Z. Chen, C. H. Yu, T. Morris, J. Tuyls, Y.-H. Lai, J. Roesch, E. Delaye, V. Sharma, and Y. Wang, "Bring your own codegen to deep learning compiler", arXiv preprint arXiv:2105.03215, May 3, 2021.

[3] G. Zheng, J. Li, W. Gao, L. Han, Y. Li, and J. Xu, "Operator Fusion Scheduling Optimization for TVM Deep Learning Compilers", in Proc. 2023 3rd International Symposium on Computer Technology and Information Science (ISCTIS), Jul. 2023.

[4] K.-T. Huang, T.-Y. Lin, P.-W. Cheng, and P.-S. Chen, "Enhancing TVM VTA Simulator Performance through SIMD Vectorization", National Chung Cheng University & ITRI, n.d.

[5] S.-Y. Cheng, R. Lai, C.-P. Chung, and J.-K. Lee, "Application Showcases for TVM with NeuroPilot on Mobile Devices", National Tsing Hua University & MediaTek, n.d.

[6] Y.-X. Huang, P.-H. Huang, J.-M. Lu, T.-J. Lin, and T.-F. Chen, "Efficient Inference of Transformers on Bare-Metal Devices with RISC-V Vector Processors", National Yang Ming Chiao Tung University, ITRI & National Chung Cheng University, n.d.

[7] "rvv-intrinsic matmul example",GitHub, [link: rvv-intrinsic-doc/examples/rvv_matmul.c at main · riscv-non-isa/rvv-intrinsic-doc · GitHub]

[8] "risc-v rvv intrinsics-viewer", GitHub, [link:Intrinsics viewer]

(十) Source code 連結

我們已把整個專案的所有程式碼都上傳到 GitHub 上，歡迎參閱與討論。

連結:**https://github.com/930727fre/tvm_rvv_matmul_byoc**