

Optimizing Large Language Model Interface on RISC-V Platforms Using TVM Compiler

指導教授:陳鵬升

組員:

中正大學 411410005 莊翔鈞

中正大學 411410073 洪子傑

目錄：

(一)摘要.....	p3
(二)研究動機與研究問題.....	p3
(三) TVM 運作流程圖.....	p4
(四) RVV 版矩陣乘法介紹.....	p5
(五) 研究方法及步驟.....	p6
(六) 結果分析.....	p8
(七) 未來可改進的方向.....	p10
(八)相關研究.....	p10
(九)研究心得.....	p11
(十)參考文獻.....	p12
(十一) Source code 連結.....	p12

(一) 摘要

為了讓大語言模型能在資源受限的嵌入式設備上運行，我們重寫了模型的矩陣乘法運算子，並利用 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，大幅提升計算效率。優化過的模型與原版相比，運行速度提升 9.6 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。

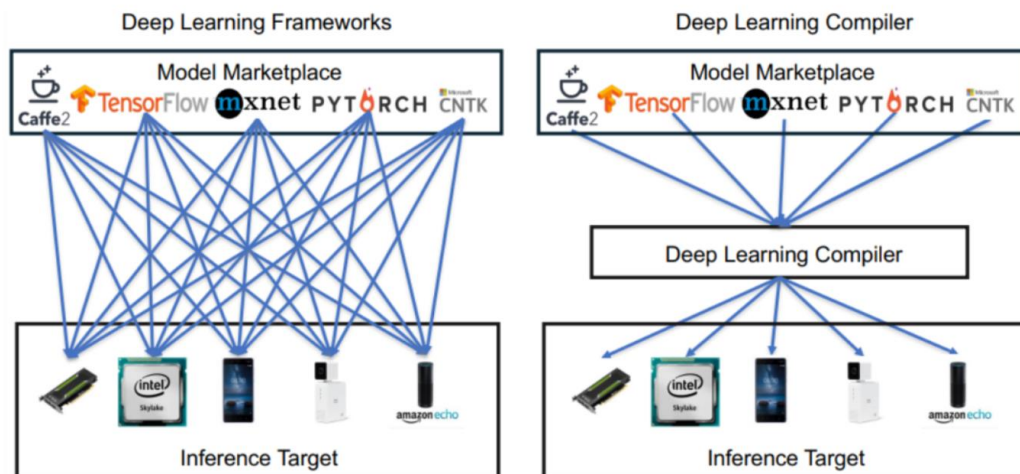


圖 1: 深度學習編譯器的優勢。

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

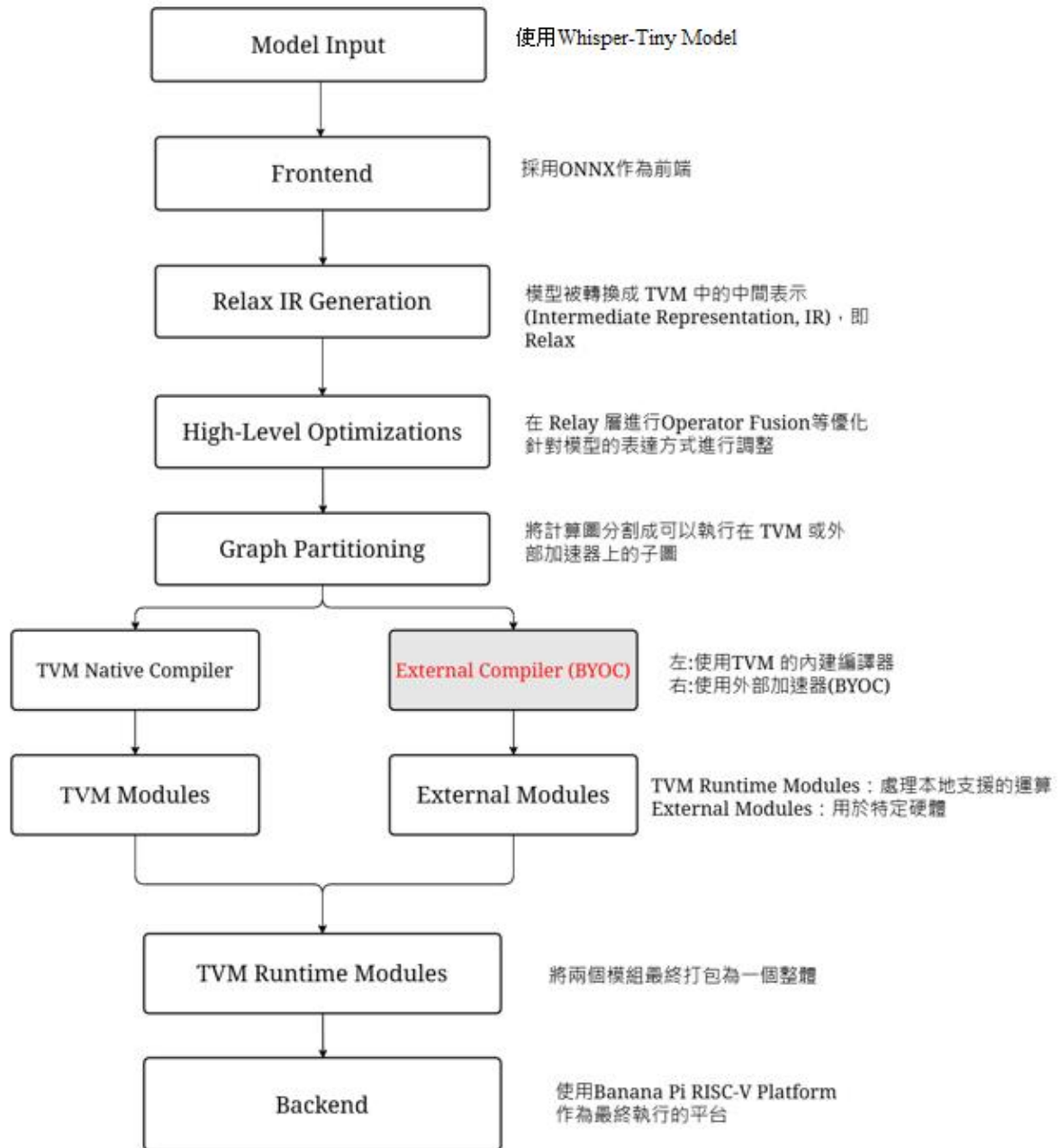


圖 2: TVM and BYOC 流程圖。

上圖為將深度學習模型部署到 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，能大幅提吞吐量與計算效率。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.     int col = 0;
3.     while (col < jTail) {
4.         size_t vl = __riscv_vsetvl_e32m1((size_t)(jTail - col));
5.         float* Cptr = Cblk + (size_t)i * (size_t)ldc + (size_t)col;
6.         vfloat32m1_t vacc = __riscv_vle32_v_f32m1(Cptr, vl);
7.
8.         const float* Ai = Ablk + (size_t)i * (size_t)lda;
9.         const float* Bcol = Bp + (size_t)col * (size_t)kc;
10.
11.        for (int kk = 0; kk < kc; ++kk) {
12.            float a_scalar = Ai[kk];
13.            const float* Bvec_ptr = Bcol + (size_t)kk;
14.            vfloat32m1_t bv = __riscv_vlse32_v_f32m1 (
15.                Bvec_ptr, (ptrdiff_t)((size_t)kc * sizeof(float)), vl);
16.            vacc = __riscv_vfmacc_vf_f32m1(vacc, a_scalar, bv, vl);
17.        }
18.
19.        __riscv_vse32_v_f32m1(Cptr, vacc, vl);
20.        col += (int)vl;
21.    }
22. }
```

圖 3:RVV 指令的程式碼實例

(4) 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

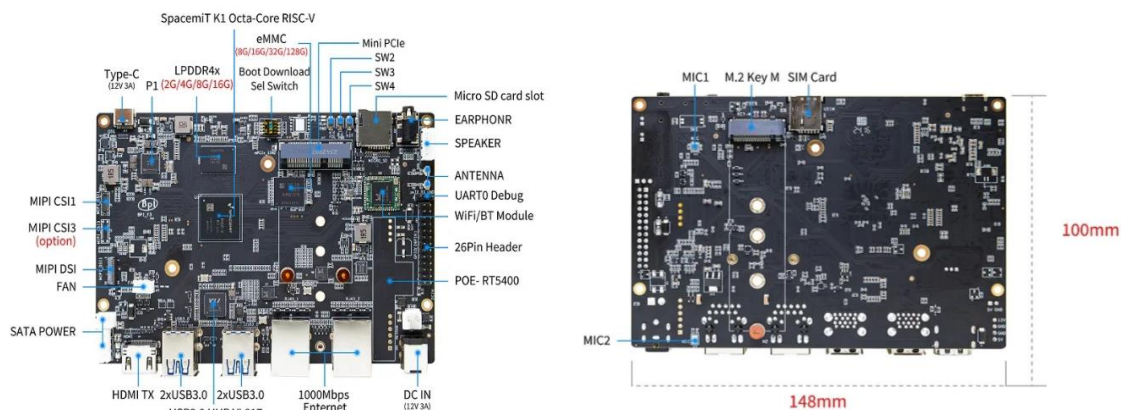


圖 4: Banana Pi BPI F3 的結構圖 (Reference from https://docs.banana-pi.org/en/BPI-F3/BananaPi_BPI-F3)。

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。

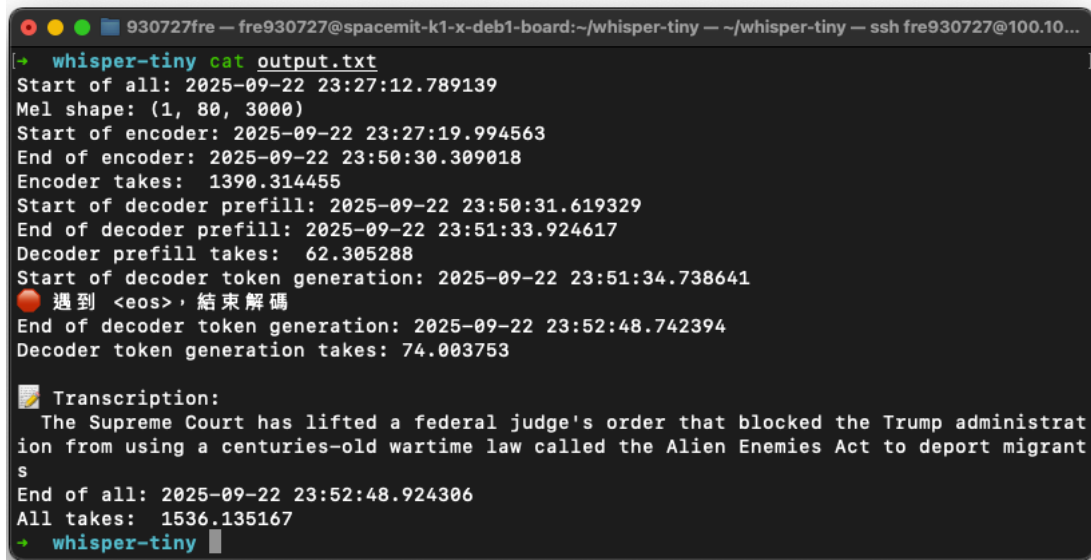
6. 推論測試

- 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：



```

930727fre — fre930727@spacemit-k1-x-deb1-board:~/whisper-tiny — ~/whisper-tiny — ssh fre930727@100.10...
[+] whisper-tiny cat output.txt
Start of all: 2025-09-22 23:27:12.789139
Mel shape: (1, 80, 3000)
Start of encoder: 2025-09-22 23:27:19.994563
End of encoder: 2025-09-22 23:50:30.309018
Encoder takes: 1390.314455
Start of decoder prefill: 2025-09-22 23:50:31.619329
End of decoder prefill: 2025-09-22 23:51:33.924617
Decoder prefill takes: 62.305288
Start of decoder token generation: 2025-09-22 23:51:34.738641
遇到 <eos>，結束解碼
End of decoder token generation: 2025-09-22 23:52:48.742394
Decoder token generation takes: 74.003753

📄 Transcription:
The Supreme Court has lifted a federal judge's order that blocked the Trump administrat
ion from using a centuries-old wartime law called the Alien Enemies Act to deport migrant
s
End of all: 2025-09-22 23:52:48.924306
All takes: 1536.135167
[+] whisper-tiny

```

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	1536
BYOC (classic matmul) -O0	1512	78	73	1673
BYOC (classic matmul) -O3	912	18	73	1013
BYOC (RVV matmul) -O0	514	51	73.7	649
BYOC (RVV matmul) -O3	69	5.3	73.8	160

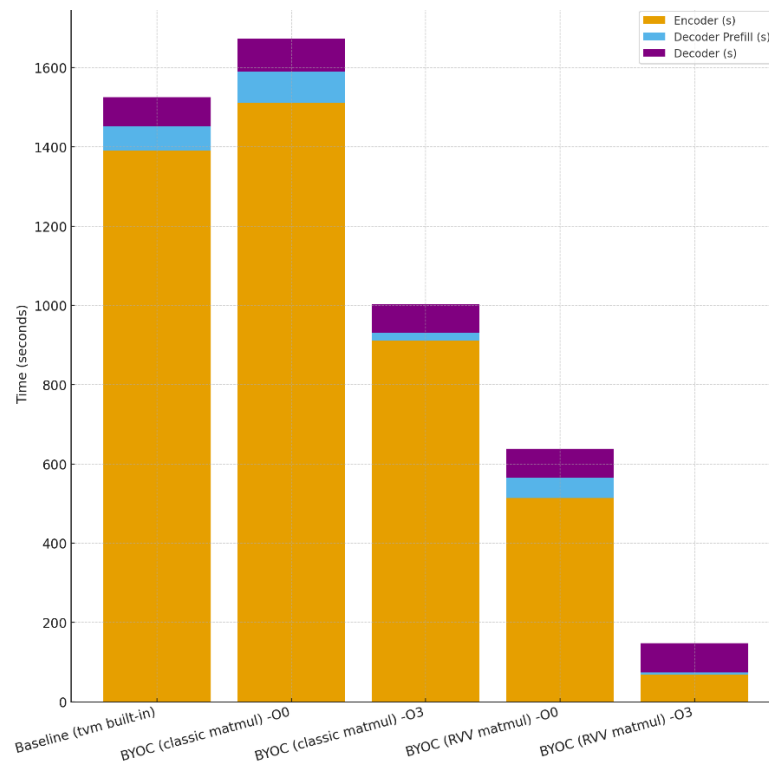


圖:6 各種執行時間對比(單位:秒)

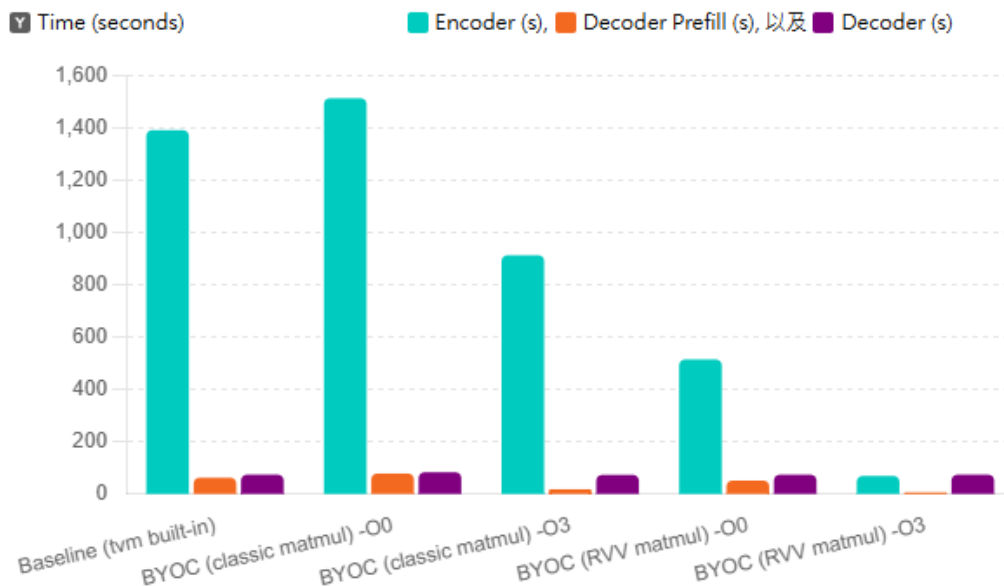


圖 7: 各項執行時間對比(單位:秒)

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 66%，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 10.4%**，原本模型在 encoder 這個步驟要花 1390 秒，優化後縮減到只要 69 秒，速度提升 20 倍；Decoder Prefill 也從 62 秒降至 5.3 秒，速度提升 11.7 倍，**總體運行速度提升 9.6 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。

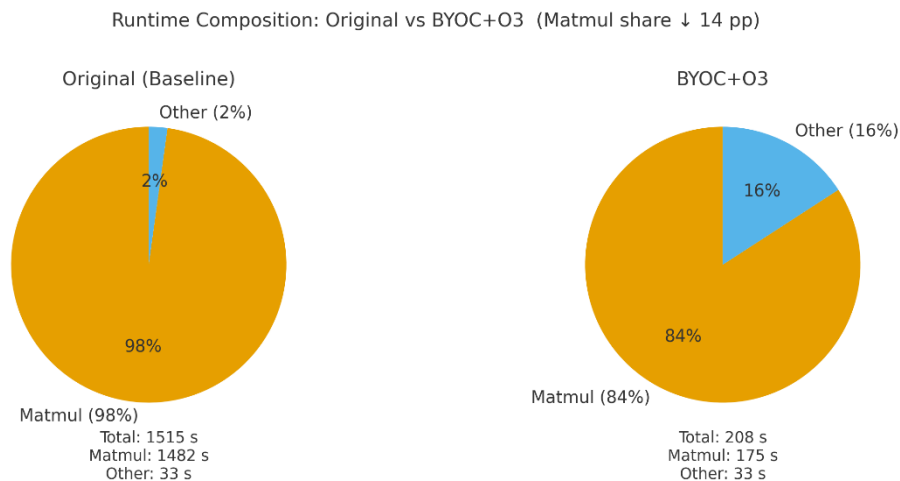


圖 8:矩陣乘法運算子佔比

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 98%降至 84%，這代表我們成功解決了原本的效能瓶頸。

(*注:圖表中的執行時間為包含評測的執行時間，非真實執行時間，但與真實執行時間的分布相同。)

(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) 相關研究

在本研究中，我們首先需要處理模型編譯的問題。TVM 提供了兩種中間表示(IR)，分別是靜態編譯的 Relay IR 與動態編譯的 Relax IR。然而，目前線上資源與教學幾乎

都以 Relax IR 為主，關於 Relax IR 的資料相對稀少，甚至連官方文件也缺乏完整說明，使得我們必須透過閱讀 TVM 團隊的 GitHub 原始碼來逐步釐清其運作原理。

在硬體平台方面，我們選擇了 Banana Pi BPI-F3 作為實驗開發板，該平台搭載 SpacemiT K1 處理器，支援 RISC-V RVV 指令集。然而，Ubuntu 官方並未提供對 RISC-V 平台的完整支援，我們最終採用了基於 Ubuntu 24.04 修改而成的 Bianbu OS 2.2 作為運行環境。

針對運算核心的優化部分，我們參考了 GitHub 上的 RVV 版矩陣乘法程式碼，並嘗試將其移植到 TVM。然而，在處理 fp16 模型時持續遇到溢位問題，最後發現需改用 fp32 模型 才能使 RVV 矩陣乘法正常運行。此外，在效能測試中，我們最初發現 RVV 實作的效能反而不如標準矩陣乘法，經過資料查閱後得知 矩陣分塊 (blocking) 是發揮 SIMD 優勢的關鍵。加入此技巧後，我們成功顯著提升了運算效能。

在模型部署上，由於 RISC-V 平台的預編譯工具與相關套件不足，我們無法直接在開發板上透過 TVM 將 ONNX 模型轉換為可執行模組。我們最初嘗試使用 RPC 遠端執行，但仍受到環境相依性問題的限制。最終，我們採用 交叉編譯 (cross-compilation) 的方式，在主機上使用 riscv64-linux-gnu 工具鏈完成編譯，並將生成的 .so 檔部署至開發板，成功避開嵌入式設備的限制。

此外，在使用 TVM 進行 BYOC 整合時，我們遭遇了 runtime 無法載入 relax.Executable 對應 loader 的相容性問題。經過多方排查後，發現問題來源於我們安裝的 TVM 為測試版，改用相同版本號的正式版後便順利解決。

綜上所述，相關研究與工作顯示，TVM 在支援新一代 Relax IR、RISC-V 平台以及 RVV 優化上的文獻與資源仍十分有限，現有的成果多集中於 Relay IR 或傳統 CPU/GPU 架構。我們的研究則是補足了這一空缺，探索如何在 RISC-V RVV 平台上結合 TVM 的 Relax IR 與自定義運算子，並成功完成模型的優化與部署。

(九)研究心得

這半年來，我們深刻體會到了什麼是做研究。不像修一門有標準答案的課，只要找到合適的教程，按部就班的學習就能掌握。而做研究有一種站在識海的邊緣，探索不可知域的感覺。沒有前人為我們開闢道路，只能自己摸索，而我們甚至連前進的方向都不知道是否正確。老師能給我們建議，卻無法為我們兜底；學長可以傳授我們經驗，卻無法替我們走完該走的彎路。整個研究過程充滿著不確定性，不知道是否能成功。我們只能不斷嘗試，一條路走不通，就試著走另一條路；學了一門技術卻無法解決現有問題，那就去學另一門。在每次的錯誤中收斂假設，直到找出解法。即使實驗不斷經歷失敗，但每次看到專案的進度表又打了一個小勾勾時，心裡還是會感到由衷的振奮和成就感，並且獲得繼續邁進的動力。

這次的研究，對我們來說最珍貴的，不是得到一組漂亮的數據，也不是學會一套操作流程，而是學到一種方法與心態，讓我們在面對未知時，能保有繼續前行的韌性與耐心，在挫折中一步步將未知轉化為已知。

(十)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](https://github.com/riscv-non-isa/rvv-intrinsic-doc/blob/main/doc/examples/rvv_matmul.c)]
- [8] "risc-v rvv intrinsics-viewer", GitHub, [link: [Intrinsics viewer](#)]

(十一)Source code 連結

我們已把整個專案的所有程式碼都上傳到 GitHub 上，有興趣歡迎參考看看。
連結：https://github.com/930727fre/tvm_rvv_matmul_byoc