CASLab GPU: An Edge AI GPU Processor Supporting OpenCL and TensorFlow CNN

國立成功大學電機工程系 Computer Architecture and System Laboratory

近年來人工智慧、機器學習領域快速發展，資料處理量大幅提升，通用型繪圖處理器(GPGPU)開始被廣泛運用於需要計算大量資料且可高度平行化處理的應用上。談到 GPU 大家總會想到 NVIDIA、AMD、ARM 等大型國外公司，鮮少聽到有國內廠商自行開發的 GPU，而 CASLab GPU 則是由國立成功大學電機工程系 Computer Architecture and System Laboratory 的師生自 2013 年起規畫研製，從軟體端到硬體端完整的系統開發，目標在打造出國內自己的第一顆 SIMT 運算型 GPU。本文介紹 CASLab GPU 軟硬體設計的相關議題。

CASLab GPU 是以 Edge Computing 為目標，並符合 OpenCL/TensorFlow API 規範，建構包含軟體及硬體的整體系統。這包括根據 OpenCL 規範設計 CASLab GPU 的 Runtime，更自行開發了 CASLab GPU 的 OpenCL LLVM Compiler。透過優化的編譯流程，使軟體堆疊更能配合硬體的運作，獲得大幅整體效能之提升，提供開發人員便利的開源執行環境。軟體層無論是 OpenCL Runtime、Compiler 都是由 C 程式開發，CASLab GPU 無論是搭配 ARM、RISC-V CPU 都能夠在其上運作與進行開發。

CASLab GPU 透過電子系統層級 (Electronic System-Level, ESL)的 Full System 設計方案，軟體與硬體開發能在早期開發即進行系統驗證。利用 C 與 C++所實作的指令級模擬器(Instruction Set Simulator, ISS)可驗證指令集的功能正確性並提供時間模型(Timing Model)來做效能上的初步評估。而 SystemC 等高階硬體描述語言則提供彈性的硬體設計方法，因為是 Cycle-Accurate 行為，開發者在早期階段就能夠更準確的達到效能分析，也能作為後續如 Verilog 等暫存器傳遞語言(Register-transfer Language)的實作範本。目前 CASLab GPU 已在 FPGA 層級完成功能性的驗證，接下來會繼續優化與考慮實際硬體的成本，讓 CASLab GPU 能夠成為一顆高效能的 Edge Computing IP。

一、CASLab GPU 軟體開發

在設計開發上，CASLab GPU 軟體堆疊情形如 Fig. 1。最上層實作 TensorFlow Runtime 讓使用者在平台上面能使用 TensorFlow API 以支援機器學習、深度學習模型開發，並透過 OpenCL Runtime 支援 GPU 來達到大量平行運算效能提升，無論是

TensorFlow CNN Application 或是 OpenCL 應用程式都能在 CASLab GPU 上達到加速效果。GPU 的軟體層級如果沒有編譯器支援的話，是無法完整將整個 GPU 系統平台建立起來的，因此編譯器在整個軟硬體系統上佔有非常重要的地位。CASLab GPU 開發了自己的 OpenCL LLVM Compiler 以支援 CASLab GPU 硬體。因為是自行開發，編譯器能夠設計出適合我們 GPU 硬體的執行程式，使硬體與軟體間達到良好的配合，提升執行效率。最後透過 HSA runtime 提供一共同硬體介面，在軟硬體間搭載一個橋樑與 Device Driver 做溝通，降低 OpenCL Runtime 的設計複雜度，GPU 收到軟體端的資訊後便開始運作，最後在將結果傳回 CPU 端的記憶體中，達到程式加速的效果。

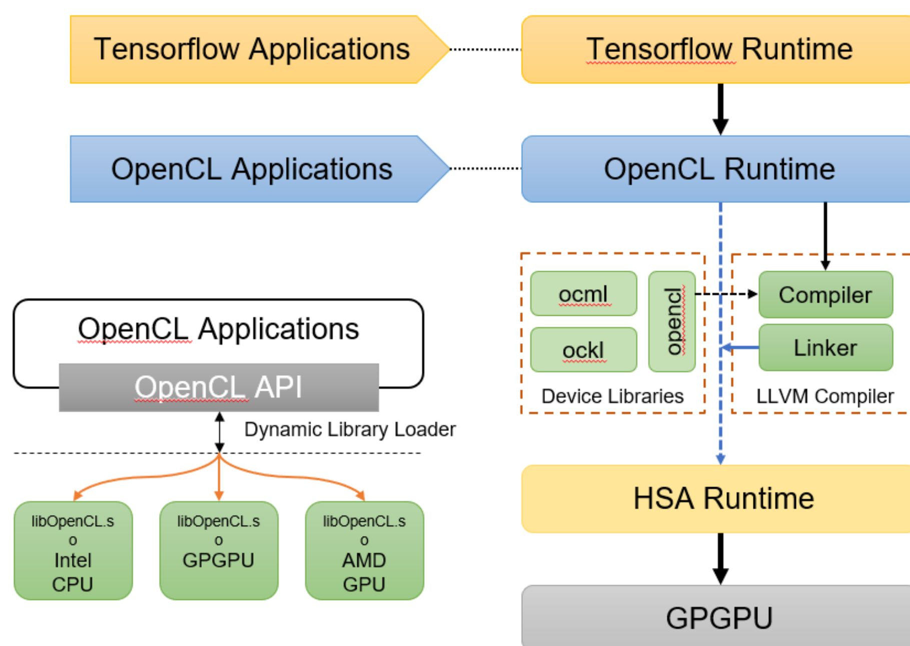


Fig. 1: CASLab GPU Software Stack

- TensorFlow Runtime

要達到 TensorFlow 應用能執行在 OpenCL 架構底下，我們需要了解 TensorFlow Stream Executor 這個由 Google 為 TensorFlow 所定義的 Kernel API 共用介面實作方式，以及 TF-Coriander 這個第三方專案的搭配方案。

以下是對這兩種專案的介紹，首先，介紹何謂 TF-Coriander。原生的 TensorFlow GPU Support 僅支援採用 CUDA Programming Language 的 GPU Device，對於其他平台開發者需自行針對目標平台設計一 Stream Executor。由於 TensorFlow 提供眾多 Kernel Operation 種類，如為該平台提供更完整的支援會需要

花費大量人力成本於此，且未來 TensorFlow 如有更新會難以同步與維護。為了降低新增硬體的複雜度，Perkins, Hugh.於 2017 年提出 CUDA-on-CL 架構，利用一名為 Coriander 之 Source-to-Source Compiler 將原生的 CUDA 應用程式轉譯為 OpenCL Device 可以執行之 Host Code 與 Device Code，藉此將 TensorFlow 原生之 CUDA 程式碼轉為 OpenCL Device Kernel，並為 OpenCL 設計一 Stream Executor 而獨立為一 TensorFlow 分支 TF-Coriander。

TensorFlow Stream Executor 為 Google 為 TensorFlow 定義出 Kernel API 的共用介面，並將與硬體相依的相關實作於各目標之 Stream Executor 中實現，其架構如 Fig. 2 所示。

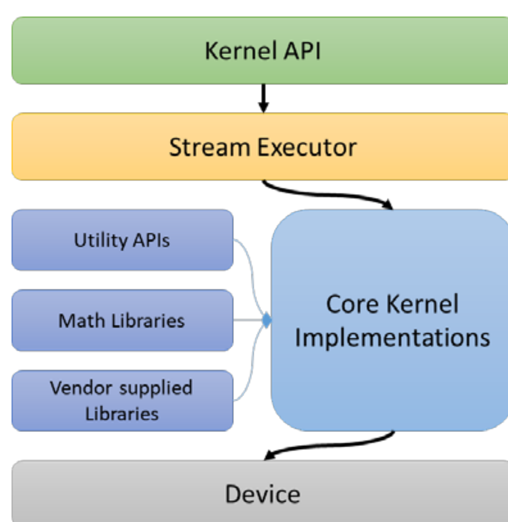


Fig. 2: TensorFlow Stream Executor 基本架構

該架構概念上是以 Stream Executor 作為各目標平台的硬體抽象層，上方應用程式（Kernel API）會透過統一介面對虛擬裝置進行諸如記憶體分配、指令派發與 Kernel Process Monitoring 等等資源管理相關命令；而各平台開發者也可藉此將平台相關優化程式放入 Kernel Implementation 中以優化各 Kernel 於該平台的執行效率。為了讓使用者可以使用 TensorFlow 原有的 API 在 CASLab GPU 上做人工智慧應用，CASLab 修改了 TensorFlow API 的後端實作內容如 Fig. 3 所示，實作了約三十多種 CNN 需求的 Operations 來支援跑在 AI 應用程式上。

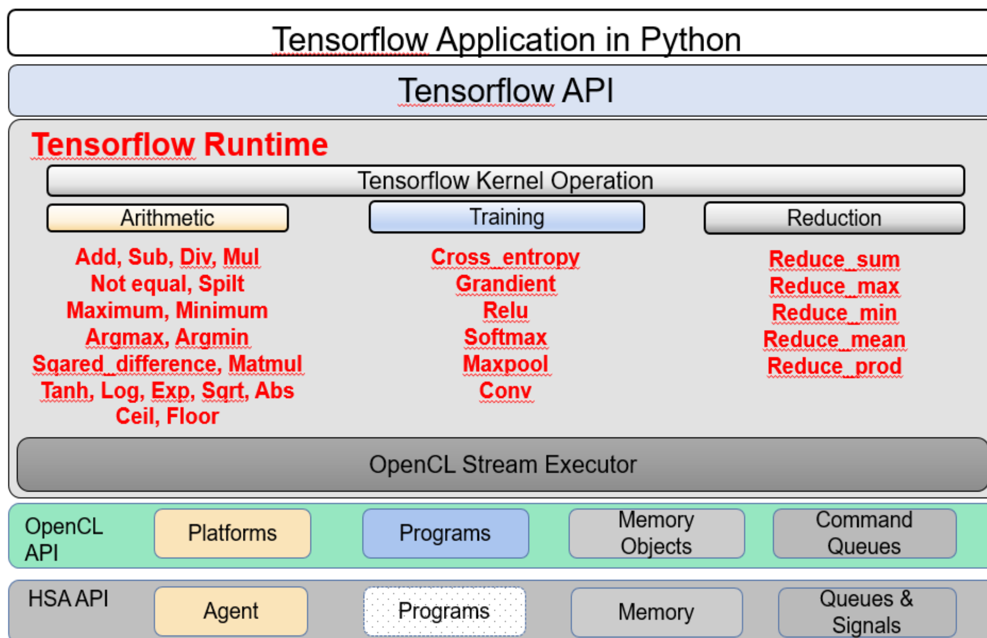


Fig. 3: TensorFlow CNN Runtime

- OpenCL Runtime

為了因應各種不同運算需求，現今的運算平台普遍由 CPU、GPU 或 ASIC 等異質性硬體所組成。而過去不同平台的應用程式開發者經常會需要重新設計應用程式 / 演算法與執行架構以配合 Device Vender 所提供的 Device Model 與 Device Driver，使得設計出的應用程式通常不具備可移植性。為此，Apple 提出一開源語言框架 - Open Computing Language (OpenCL)，並交由非營利組織 Khronos Group 進行管理、維護。OpenCL 為了讓使用者能有效的管理資源，設計了一套軟體架構，讓使用者能由上至下，由大到小的來管理硬體上的資源，如 Fig. 4 所示。CASLab GPU 為了能夠有效支援 OpenCL API，以 C++物件導向，設計一個符合規範的 OpenCL Runtime API，其中每一個物件內部會記錄與硬體相關的資訊以及使用者使用各項資源的情形。

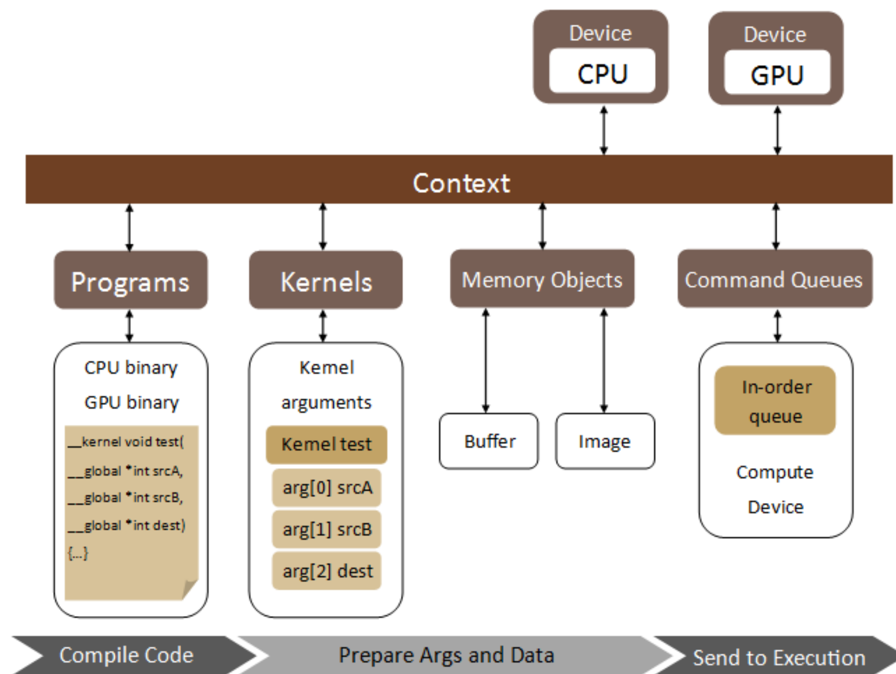


Fig. 4: OpenCL Runtime Architecture

CASLab 實作的 API 有四大類，目前共計 33 個，如 Fig. 5 所示

1. Platforms 相關
   [1] Platform object 用以實作取得整個平台系統相關資訊的 API
   [2] Device object 用以實作取得 Platform 中數個裝置資訊的 API
   [3] Context object 用以管理使用者在一個 Device 需要的硬體資源
2. Programs 相關
   [1] Program object 用來記錄使用者所撰寫的 Device code，內部可以含有多個 kernel。編譯 Device code 的 API (clBuildProgram)，需使用此物件來實作，先呼叫 CASLab 所實作之 Compiler，完成編譯流程後，將結果紀錄再次物件中。。
   [2] Kernel object
   此物件可記錄編譯完成後的 Program 中的其中一個 kernel，並且用以記錄要派發給 GPGPU 的相關資訊。
3. Memory 相關
   [1] Memory object
   用來宣告硬體空間給使用者，並且使用者也能用此類別中的相關 API，來讀取或寫入資料到 GPGPU。
4. Command queues

[1] Command queues object

用來管理使用者所要派發給 GPGPU 的工作資訊，內部紀錄使用者要派發的順序與內容。

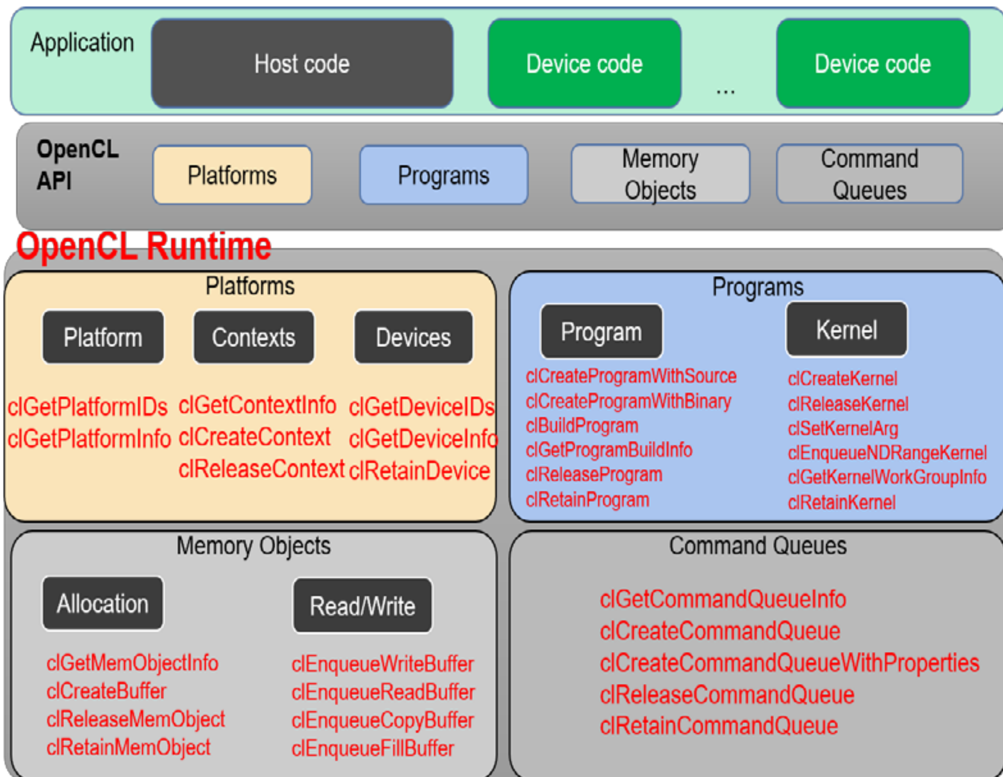


Fig. 5: Implementation of OpenCL API for CASLab GPU

- HSA runtime

類似於 OpenCL 提供一共同的平行運算軟體開發 Framework，HSA 其旨為提供一共同硬體介面。不同於 OpenCL 規範了統一的應用程式開發介面，HSA 規範了統一的硬體操作介面，以簡化上層如 OpenCL 等與底層進行橋接介面之開發複雜度。HSA 主要定訂了 Master Device (E.g. CPU) 與 Slave Device (E.g. GPU) 之間的記憶體空間擺放格式、資料傳輸方式與指令傳送方式等基本溝通協定。並定義一共用中介語言 HSAIL (Heterogeneous System Architecture Intermediate Language) 供軟體層使用，使高階語言如 OpenCL Kernel Code Compiler 設計商不需要針對不同目標硬體額外開發一套編譯器，僅需於 Kernel 派發時由裝置定義之 Finalizer 將 HSAIL 轉譯至目標可執行檔。

舉例來說，HSA 與 OpenCL 進行整合時 OpenCL Kernel Code 轉至 HSAIL 之編譯器可為採用 AMD CLOC，而目標硬體為 CASLab GPU 時再透過我們實驗室先前所開發之 Finalizer 將 HSAIL 進行指令擴充、對應並編譯為 CASLab GPU 所支援之可執

行檔。但這套流程在實際執行上的效能並不佳，因為編譯器採用的是 AMD CLOC 在許多時候並不能針對目標硬體去優化，因此 CASLab GPU 是採用 HSA + LLVM Compiler 來達到編譯執行檔的優化。

CASLab 實作之 HSA runtime 如 Fig. 6 所示，HSA API 可分成四大類，與 OpenCL API 相互對應。

1. Agent 相關，與 OpenCL API 中 Platforms 對應，用以取得硬體相關資訊。

2. Program 相關，此部分直接呼叫 CASLab OpenCL Compiler 來將使用者的 Device code 編譯成 GPU 執行檔。

3. Memory 相關，與 OpenCL API 中的 Memory object 對應，為 OpenCL API 中所需要的 GPU 記憶體做管理。

4. Queues and Signals 相關，與 OpenCL API 的 Command Queue 對應，可將 OpenCL API 中所發出的工作，轉換成信號通知 GPU 來執行。

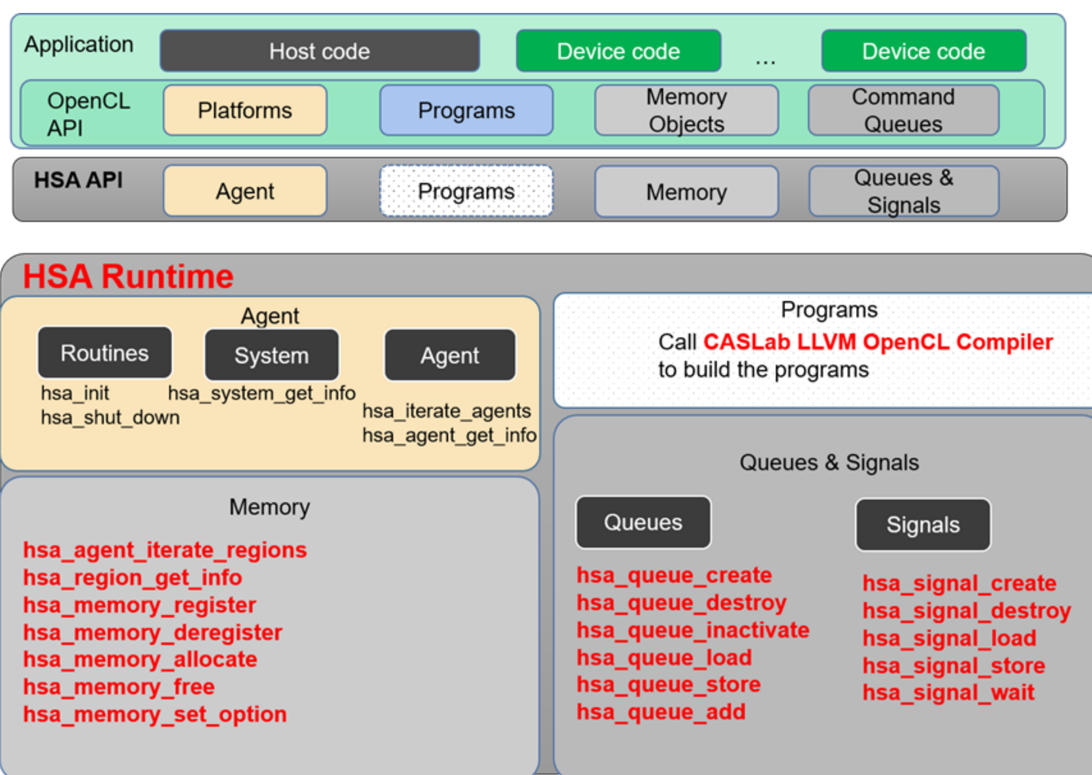


Fig. 6: HSA Runtime

- LLVM Compiler

LLVM 全名為 Low Level Virtual Machine，由 Vikram Adve 與 Chris Lattner 進行開發。LLVM 起源為針對動態與靜態編譯研究所設計的開發環境 / 虛擬機器，提供數種開發與分析工具搭配直譯器 / 虛擬平台以執行目標程式；隨著專案開發越來越豐富，LLVM 發展方向也逐漸由虛擬平台轉變為編譯器開發框架，專案名稱內的 Virtual Machine 也不再代表虛擬機開發，使目前 LLVM 專案名稱僅以代表整體開發、不再具備原先專案意義。也由於 LLVM 逐漸轉向 Compiler 開發框架，開始越來越多公司轉向支持 / 採用 LLVM 作為其硬體 Compiler 之開發環境，如 Apple 於 2005 年開始大力支持 LLVM，且採用 LLVM 平台作為其作業系統之預設編譯環境提供使用者使用。

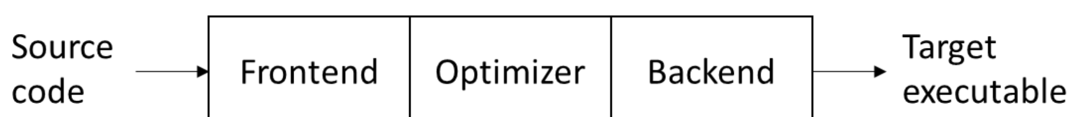


Fig. 7: Compilation flow

普遍編譯器設計上會依工作內容進行分工，將整體編譯流程拆分為 Compiler Front-End、Optimizer 與 Backend 這三部分元件，其編譯流程如 Fig. 7 所示。其個別功能為：Front-End - 負責進行與語言相關的處理；例如，將由 C++所設計之程式碼進行轉譯、轉譯為內部所需的 AST Tree 等語法樹資料結構，並執行語言前處理相關步驟。Optimizer 則為程式碼內容相關的優化步驟；例如，常數前處理、條件式優化等等與語言相依的優化處理。Backend 則將前兩部分所產生的資料結構進行指令統整，並產生出目標可執行的指令、檔案格式。

為了於 LLVM Infrastructure 中新增 CASLab GPU 目標硬體支援，需要 LLVM Project 內的” lib/Target”目錄中新增本實驗室設計之硬體平台 - CASLab GPU，為方便使用，將名稱簡化為 CASGPU。CASLab GPU Target Machine 於 LLVM Project 內的模組架構如 Fig. 8 所示。

我們需要先建構出硬體的命名空間 - CASGPU，並於該命名空間內建立各項子模組與功能等平台內需使用之相關功能。CASGPU 作為 CASLab GPU 硬體於 LLVM 中的 Target Machine，用以表示諸如 X86、ARM 與 RISC-V 等不同目標硬體架構。另外，在 CASGPU Target Machine 下，另外新增了一名為 HSA SubTarget，用以代表在 CASGPU 目標平台下採用 HSAIL-lite 指令格式的目標架構，並依據 HSAIL 之規範於此定義共用之參數與功能；例如，Memory Space 與 Calling Convention 等。最後，定義一目標晶片（C100 - CASLab-GPU ISA V1.0.0），以代號”CA” 作為硬體代表並作為各設計檔案命名前綴，於內部定義暫存器數量、指令支援等硬體相依之功

能。

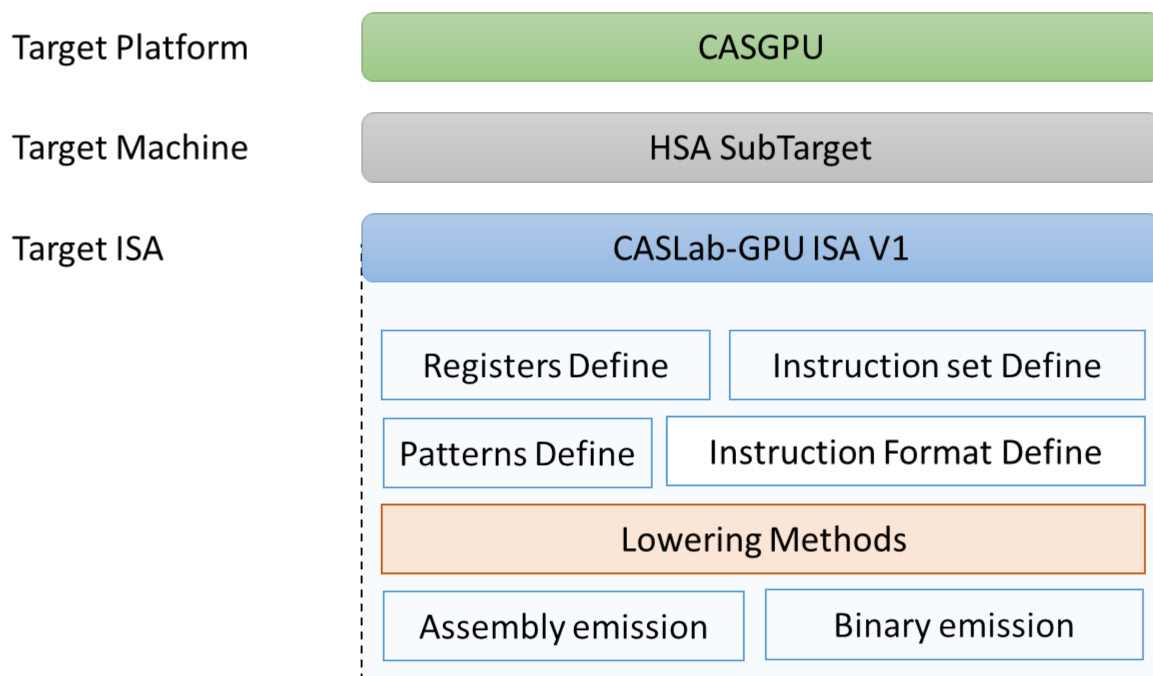


Fig. 8: LLVM-Backend CASLab GPU Structure

在說明完 LLVM Compiler 在 CASLab GPU 是如何規劃後，實作上我們的編譯器目前包含了大約 15000 行的 C code。

Fig. 9 顯示 CASLab GPU 編譯器與 CUDA 的系統對照圖。圖中最上層的是輸入資料的格式， CUDA 要求輸入 CUDA 格式的 .cu 檔案，其他兩者為 CASLab 支援 OpenCL 的 .cl 檔案。而第二層則是三者分別使用的不同編譯器，原版的 CASLab GPU 是採用 AMD CLOC + Finalizer 方案，但因為 AMD CLOC 只提供 Binary 使用，而必須符合其架構設計出對應的 Finalizer 導致許多優化無法達到，新版的 CASLab GPU 設計了自己的 OpenCL LLVM Compiler，使平台更有彈性，也達到效能上的優化。

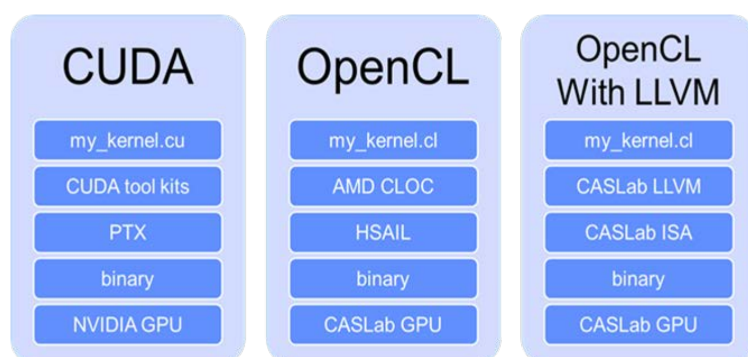


Fig. 9: CASLab GPU 與 CUDA 系統對照圖

二、CASLab GPU 硬體架構與設計

CASLab GPU 為一 SIMT (Single Instruction Multiple Thread)架構，如 Fig. 10 所示，主要由 Interconnection Network (IN)、多個 SM (Streaming Multiprocessor)、WS (Workgroup Scheduler)所組成。IN 負責連接外部 DRAM 和各個元件之間以及資料的搬運。SM 則是運算與執行的主要核心單元，WS 為 GPU 內部之總控制器，主要負責與 CPU 溝通、接收來自 CPU 的工作以及派發 Workgroup。

GPU 執行緒的排程會經由兩層不同的排程單元來派發執行緒，第一排程器 Workgroup Scheduler。當 CPU 發送新的工作時，以 Grid 為單位接收所要執行之程式，再進行切割與排程後，以 Workgroup 為單位派發至每個 SM 去做執行。SM 在收到 Workgroup 後，會根據 SIMD width 分成多個 Warp，並以 Warp 為單位進行運算，同一個 Warp 內的執行緒為平行運算。

第二排程器為位於 SM 內部 Front-end 的 Warp Scheduler，將多個 Warp 進行排程後，交由 Back-end 運算單元處理。

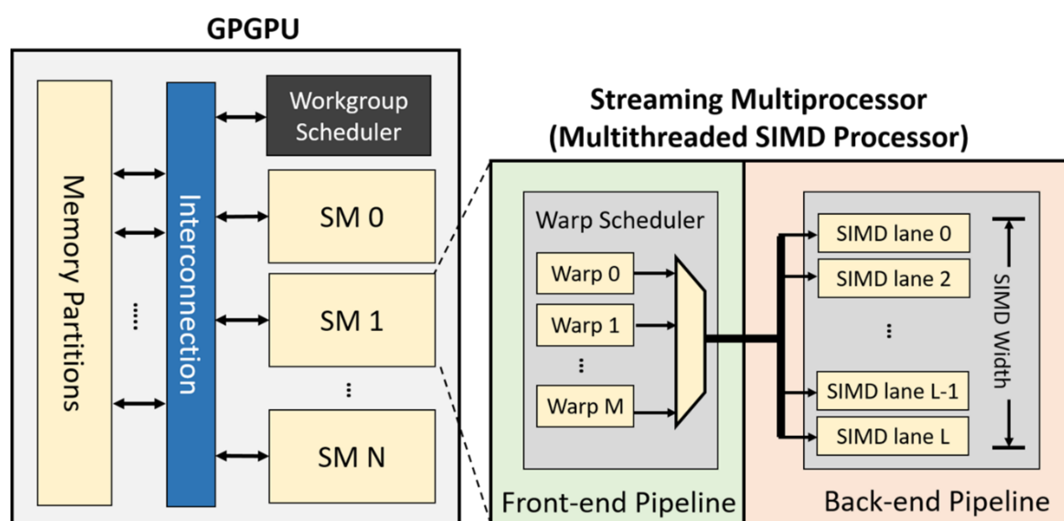


Fig. 10: GPGPU Architecture

CASLab GPU 的 SM 架構如 Fig. 11 所示，可分為 Workgroup Initializer、Front-end Pipeline 和 Back-end Pipeline 三部分。Workgroup Initializer 負責接收 Workgroup 以及其初始化，Front-end 主要為 Warp 之排程以及指令的前處理與 Decode，Back-end 則負責資料的運算以及 Load/Store 之相關。

下面章節會分別對 Workgroup Initializer、Warp Scheduler、Divergence Stack 這三個對 GPGPU 極為重要的模組做介紹。

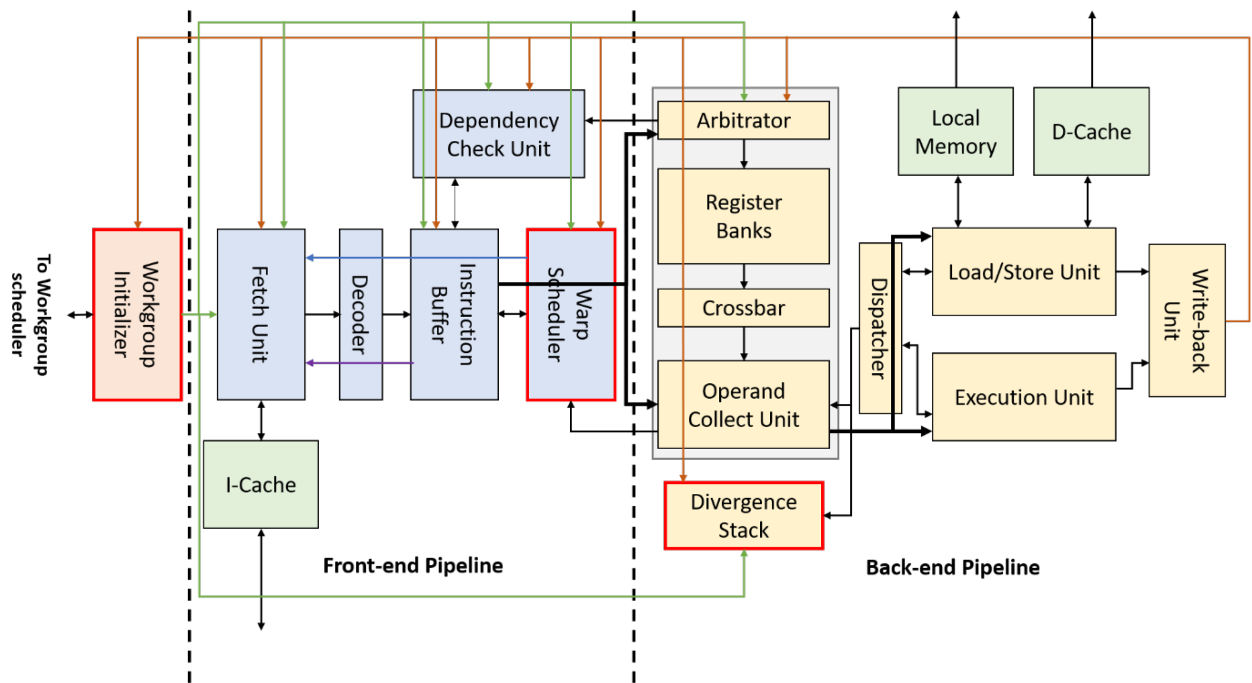


Fig. 11: CASLab SM Architecture

- Workgroup Initializer

Workgroup Initializer 做為接收 Workgroup 的單元。所有 Workgroup Scheduler 發派至 SM 的 Workgroup 都會在這裡進行初始化，並且負責管理整個 Workgroup 的執行進度與狀態，具備以下幾種功能:

1. 從 Workgroup Scheduler 收到軟體派下來 Workgroup 資訊，包含記憶體初始化地址、Register 使用的數目、總共需要執行幾條執行緒....等等。

2. 根據 SIMD width 將接收進來的 Workgroup 切成多個 Warp，同個 Warp 的執行緒以不同資料資源執行相同的指令，來達到 SIMT 執行。

3. 每個 Warp 都會有自己的 Register space 與 item ID，使不同 Warp 雖然執行相同的指令但實際上運算的資料是不同的，用許多 Warps 排序執行，來達到 Latency Hiding 的目的。

4. 初始化每個 Warp 的資料，包含 PC、Special registers、Active mask, Lane ID、Workgroup info address、Kernel argument address ...等，Active mask 裡面則會存放每個 Warp 需要執行的 Lane 數目。以 CASLab GPU 來說，每個 Warp 能夠有 32 條 Lane 來執行，但實際上可能一個 Warp 只需要 16 條 Lane，因此在 Initial mask 就會只給 16 條 Lane active，剩下 16 條 Lane 在運算時並不會有 Commit 的動作。

5. 當 Workgroup 中的所有 Warp 都完成時，釋放佔用的資源，並送出完成訊號。

- Warp Scheduler

經由 Decoder 解碼完成的指令，會先存放到 Instruction Buffer 中，需要先通過相依性的檢查以判斷是否為可以派發的狀態。而從可以被發派的眾多指令當中，選擇一條並發派到後端對應的運算單元執行，就是 Warp Scheduler (WS)主要負責的工作，透過 WS， GPU 能夠達到 Hiding latency 的效果。當一個 Warp 因為 Dependency 或是 Memory access 造成 Stall 時，就能夠切換其他的 Warp 做事情，讓 GPU 一直處於運作的狀態。CASLab GPU 設計了多種 WS 像是 Loose Round Robin(LRR)、Greedy-Then-oldest (GTO)、Two-Level (TL)、Prefetch-Aware (PA)，利用不同機制可以看到不同的效能結果與記憶體搬移次數的差異。

- Divergence Stack

CASLab GPU 在設計上採用 SIMT 架構，在同一個 SIMT Core 內僅會有一組 Instruction Decoder，並派發不同運算資料但相同的指令至該 SIMT Core 內部各 Lane 執行。因為資料的不同，當遇到 Conditional branch 這類的指令時，會因為判斷條件不同造成執行流的分歧(Divergence)發生，亦即 SIMT Core 內各條 Lane 目標 PC 不一致而無法繼續以 SIMT 模式執行。針對此問題 CASLab GPU 採用 Masked Execution，遇到分歧時會使用 Lane Mask 決定有效 Lane，執行時依然採用 SIMT 執行模式，但會根據 Lane Mask 決定執行結果是否須被 Write-Back，待該執行流程結束後，再切換至另一分支。執行順序如 Fig. 12 所示，分為 1, 2 兩個執行流程分別執行。

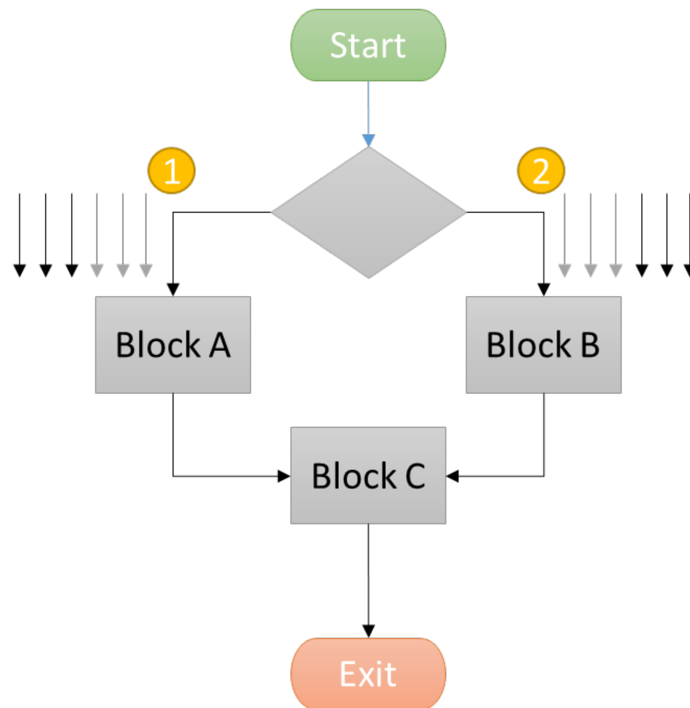


Fig. 12: SIMT diverged execution

Branch Divergence 除了會因採用 Masked Execution 執行而造成 SIMT Core 執行效率降低外，後續如何執行也是需要由 Compiler 與硬體來共同決定，需要決定 Lane Mask 在何時需要被修改、何處決定可以結束 Masked Execution 使 SIMT Core 中的各 Lane 皆具備有效運算。在過去採用 AMD CLOC 與 CASLab Finalizer 的組合下，因為缺少完善的 Control Flow Graph 分析能力，難以得知整體 Kernel Code 的執行流程與 Basic Block 間的相依關係，這導致某些路徑會有不必要的重複執行，而降低整體執行速度。

以 Fig. 12 為例，假設 Block C 內部開發者並無手動增加 Barrier，會使得 Finalizer 在搜尋 Converge point 時僅能採用 Exit node，讓整個執行流程變為 Start → 1 →Block A→Block C→Exit, (Switch Mask), 2 →Block B→Block C→Exit。可以發現到，共同的 Block C 與 Exit node 在兩次 Masked Execution 中被執行了數次，使相同 Basic Block 僅因 Branch path 不同而需分開執行，喪失了 SIMT 所具備之優點，大幅地降低了整體的執行效率。

對於此問題，CASLab Compiler 使用了 IPDOM 以優化 Branch 執行效率，透過建置 Post Dominator Tree，尋找 Immediate Post Dominator 節點作為 Branch Reconverge point，並於 Branch 指令提供 Reconverge BasicBlock 資訊，且於該 Basic Block 進入點新增一特殊指令以通知硬體、改變硬體的執行流程，盡量降低不必

要重複執行之 Basic Block。

以 Fig. 12 所示的執行流程為例，透過 Post Dominator Tree Analysis Pass 可以建置出如 Fig. 13 - A 的 Post Dominator Tree，可以發現 BasicBlock A 與 BasicBlock B 所擁有的 Post Dominator (PDOM)與 Immediate Post Dominator(IPDOM)皆為 BasicBlock C，即可判定 Block C 可作為該 Branch Divergence 之 Reconverge Point，再於 Basic Block C 前端插入 IPDOM 相關指令，即可將整體執行流程更新為如 Fig. 13 - B，使 BasicBlock A 執行完後跳至 BasicBlock C 執行時因執行到 IPDOM 特殊指令，即可更新 Execution Mask 將執行流程轉至 Divergence 另一半部分，開始執行 BasicBlock B 內之指令，執行完成 BasicBlock B 後跳至 BasicBlock C 時將再執行一次 IPDOM 特殊指令，此時即可結束 Branch Divergence，清除 SIMT 的 Execution Mask，使 BasicBlock C 與 Exit Node 皆可於 SIMT Core 內各條 Lane 中一同執行，避免不必要的重複執行。

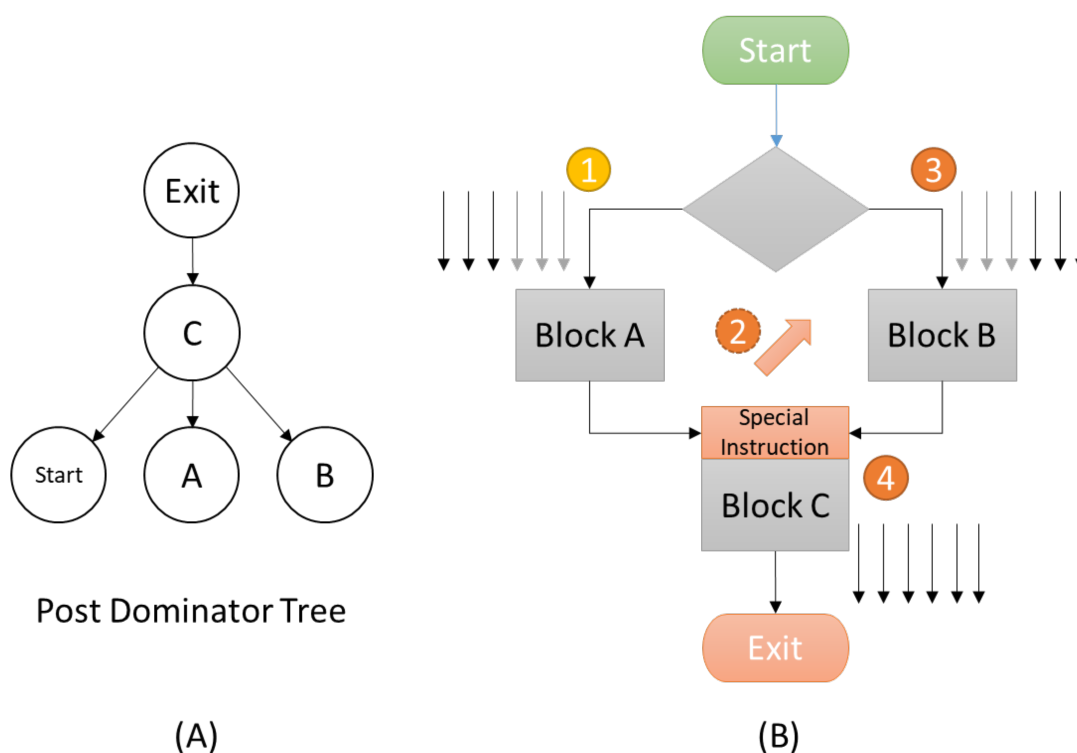


Fig. 13: IPDOM Execution Flow

利用 IPDOM 方法找到分支合併的點，搭配 Divergence Stack 來處理分支與合併問題。Divergence Stack 中以堆疊的結構存放每個 Warp 的遮罩(Mask)與分支相關資訊的遮罩堆疊單元，Divergence Stack 中會存有 48 個 Warp 的遮罩，每層堆疊存放 Mask、Execute PC (EPC)、Reconvergence PC (RPC)。Mask 用於運算時控制 32 條

lane 在 Write back 時是否有效，而 EPC 和 RPC 則會用於分支合併。當 Issue 一筆指令時，會從 TOS 中取出 Mask 用來控制資料是否要寫回 Register 或 Memory；當遇到 end branch、exit、barrier 指令時，會從 TOS 取出 RPC 來判斷目前是否到了匯合點，Next PC 此時就會從 EPC 開始執行，透過軟體與硬體的搭配達到有效控制 SIMT 中分歧的狀況。

三、設計驗證結果與效能評估

- 實驗環境

驗證實驗平台上我們設計 CASLab GPUsim 模擬平台與 CASLab GPU-Lite RTL Design 兩種來做驗證與比較。在結果驗證上，目前 GPUsim 通過多支 PolyBench 應用程式與 CNN 相關 AI 應用 Training 與 Inference python 程式；GPU-Lite RTL 平台上則通過多支 PolyBench 應用程式與 Lenet-5 inference，驗證實際硬體的正確性。

CASLab GPU 在 GPU 硬體配置上，如表 1 所示，在模擬平台上主頻率調整在 1.3GHz，內部使用 8 顆 SM，每個 SM 的 SIMD Width 為 32，總共有 256 個運算核心，將 CASLab GPU 調整規格與 Nvidia TX2 相似來做效能上的評比。另外，我們也納入 Intel i7 CPU 的比較。

表 1: GPU/CPU Configuration

Experiment Platform	Our GPGPUsim	Nvidia TX2	Intel i7 6700
Frequency	1.3 GHz	1.3 GHz	3.4 GHz
SM Number	8 SMs	8 SMs	N/A
DRAM Frequency	2400 MHz	1866 MHz	2133MHz
Workgroup Number	8 Workgroups /SM	N/A	N/A
Warp Number	48 Warps /SM	N/A	N/A
SIMD Width	32	32	N/A
Core Number (SP)	256	256	4(8 threads)
L1 Instruction Cache	16KB (2-way, 64byte line, 128 set)	N/A	128KB
L1 Data Cache	16KB (4-way, 128byte line, 32 set)	N/A	128KB

- 測試程式

測試程式之部分，已通過許多不同功能的測試程式，表 2 列出較有指標性的 12 支，包含了向量的乘法運算、矩陣乘法運算、轉置、類神經網路中常使用的 Convolution 和 Fully Connected 網路，以及 LeNet-5 的 Training 、Inference，以上皆為 GPGPU 應用程式中常見的運算內容，可有效利用 GPGPU 大量平行運算的特

性。

表 2: Poly-bench & CNN Benchmark

Benchmark	Description
3DCONV	3-D Convolution
3MM	2 Matrix Multiplications (E=A.B ; F=C.D ; G=E.F)
ATAX	Matrix Transpose and Vector Multiplication
BICG	BiCG Sub Kernel of BiCGStab Linear Solver
CORR	Correlation Computation
COVAR	Covariance Computation
GEMM	Matrix-multiply C=alpha.A.B+beta.C
MVT	Matrix Vector Product and Transpose
SYRK	Symmetric rank-k operations
Fully Connected	Fully Connected NN
LeNet-5 Training	Training of LeNet-5, a classic CNN architecture
LeNet-5 Inference	Inference of LeNet-5, a classic CNN architecture

- CASLab LLVM Compiler 效能結果

Fig.14 到 Fig.16 比對過去 AMD CLOC & GPU Finalizer 與 CASLab 所設計之 Compiler 對於各 Testbench 之 Kernel Code 之編譯時間、Code Size 與 Register。從 Fig.14 可以看出在編譯時間上，我們的 LLVM Compiler 遠優於原版 AMD CLOC + Finalizer 組合。Fig.15 在 Register 的使用上，LLVM Compiler 達到 41.6%的優化，利用更少的 Register 就能完成程式的運算。Fig.16 則說明單從指令數目來看，LLVM Compiler 也能達到 17.6%的指令縮減。

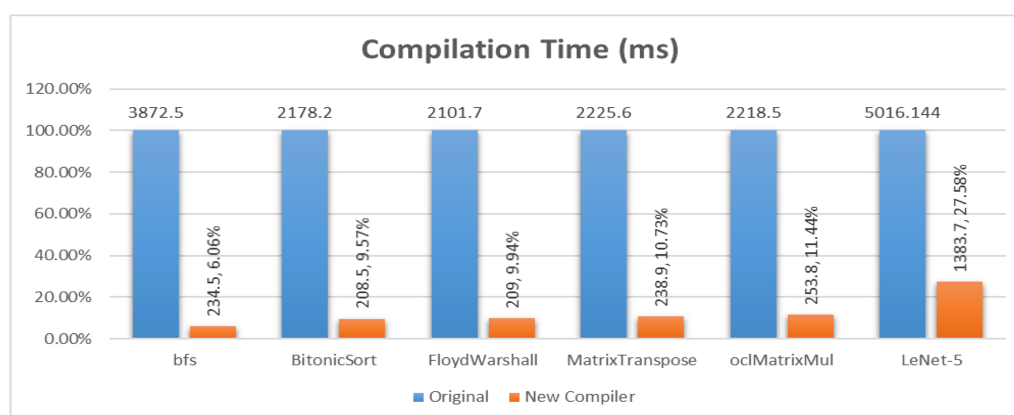


Fig.14: Compilation time comparison of each testbench

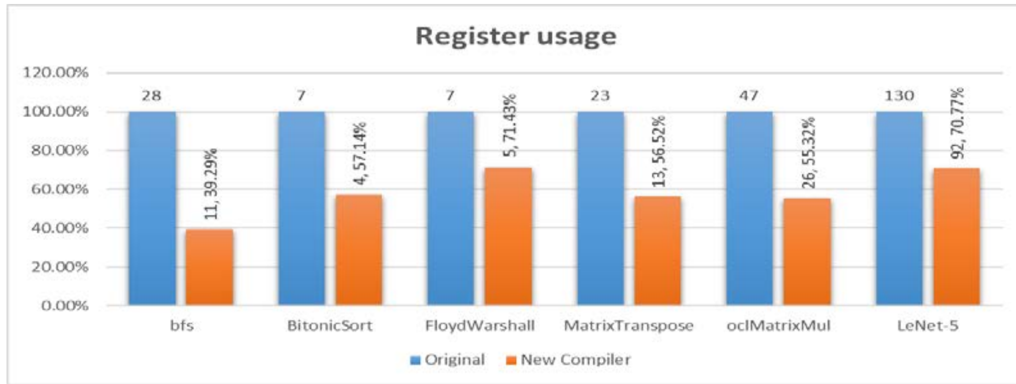


Fig.15: Register usage comparison of each testbench

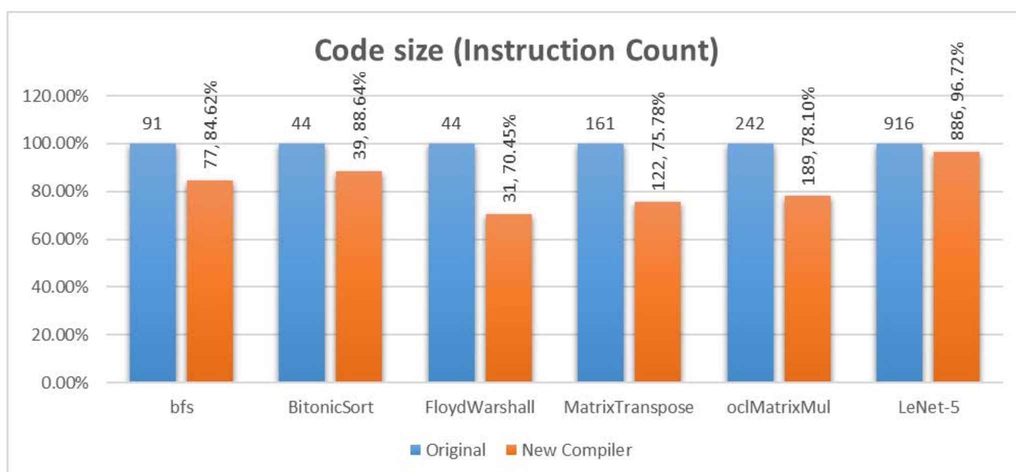


Fig.16: Code size comparison of each testbench

- CASLab GPUsim 效能結果

為了比較 CASLab GPU 在效能上的表現，這裡選用 intel-i7 6700 以及規格與我們相近的 Nvidia TX2 作為對照，測試程式使用 LeNet-5 Inference。在附錄有 GPUsim 平台的實際 Demo 影片之 Youtube 連結，包含[1] LeNet-5 Inference 以及[2] LeNet-5 Training(5000,28*28)。Fig.17 為執行 LeNet-5 inference 時間的柱狀圖，時間測量方式參考附錄[測量 TensorFlow Application Execution Time]，i7-6700 花費 2.76ms，TX2 花費 0.18ms，如果使用舊版 CASLab GPU 利用 AMD CLOC + finalizer 做編譯所執行之時間為 0.249ms 到 0.219ms 取決於 CASLab GPU 不同微架構設計，而使用新版 CASLab 開發的 LLVM Compiler 時只需要 0.175ms 到 0.172ms，在效能上有顯著的改進，執行時間可優於 TX2。附錄[3]為 CASLab GPU 模擬平台的 github 網址，使用者可以在上面執行 CASLab GPU 看整個系統的執行狀況與模擬時間。

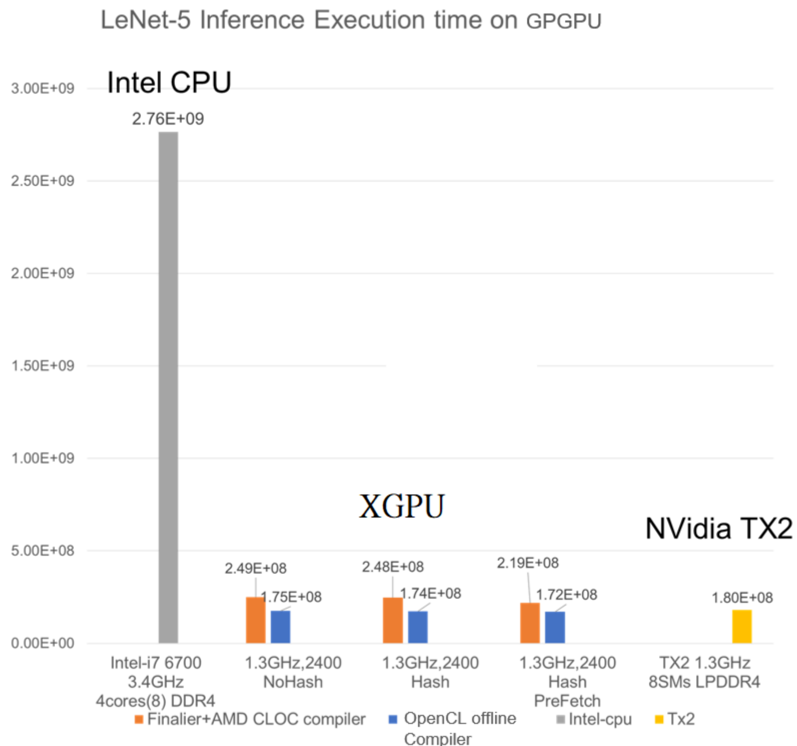


Fig.17: LeNet-5 Inference Execution Time

- FPGA 驗證成果
  - FPGA configuration
    - Board device information
      - Name：HAPS-70(S-12)
      - Device: Xilinx Virtex-7 XC7V2000T FLG1925
      - Logic Cells: 1,954,560
      - Sram size: 12892 x 36Kbits
      - DSP slices: 2160
    - Software support
      - ProtoCompiler：Compile, Pre-map (add constrain), Map (construct actual circuit)
      - Vivado: Place and Route
      - Confpro: program

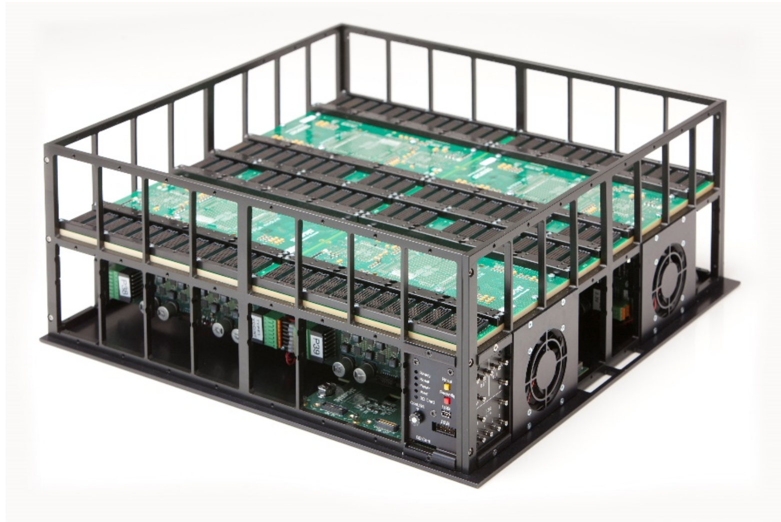


Fig.18: HAPS-70

Fig.18 為 HAPS-70 FPGA 圖，採用 TSMC 40 奈米製程來合成 CASLab GPU-Lite 版本，規格如表 3 所示，整顆 GPU 在 FPGA 上使用了 1,054,569 個 LUT(look up table)及 898 顆 36kBits block RAM，合成後面積約為 5,627,088.775μm2 速度為 100MHz。

本平台在 GPU 硬體配置上因為硬體限制的原因，目前 FPGA 上的規格都是精簡版本，主頻率為 100MHz，內部使用 1 顆 SM，每個 SM 的 SIMD Width 為 4 總共只有 4 個運算核心，在 DRAM 的部分用 FPGA 內部提供的 SRAM 來做替換。

表 3 GPU FPGA configuration

Experiment Platform	Our GPGPU in FPGA
Frequency	100 MHz
SM Number	1 SM
DRAM Frequency	N/A (使用 FPGA 內的 SRAM)
Workgroup Number	8 Workgroups /SM
Warp Number	32 Warps /SM
SIMD Width	4
Core Number (SP)	4
L1 Instruction Cache	16KB (2-way, 64byte line)
L1 Data Cache	16KB (2-way, 128byte line)

Fig.19 顯示 CASLab GPU-Lite RTL 平台與 CASLab GPUsim 平台上針對幾支 polybench 的執行時間比較，可以發現時間上兩者是非常相近的，證明 CASLab 所開發的 GPU 符合時序精確性與實際硬體運作的行為。目前因為 FPGA 硬體資源有限，我們無法類似於 Nvidia TX2 的規格在實際 FPGA 上驗證比較，但在縮減版

CASLab GPU-Lite 已合成並驗證其結果，展望未來將持續優化硬體部分的限制，附錄[4]影片說明 CASLab GPU-Lite 於 FPGA 上的驗證流程。

CASLab 採用 ESL full system 設計，從應用程式端打造 TensorFlow-OpenCL 執行環境，根據 OpenCL Spec 實作自己的軟體堆疊，基於開源的 HSAIL 設計出 HSAIL-Lite 版本的 GPU ISA，更開發了 LLVM Compiler 來支援程式的編譯以優化整個 GPU 的執行時間。硬體端開發 SIMT 架構 GPU，達到多執行緒平行運算，在 Full system 底下驗證 CNN model、OpenCL testbench 的正確性，將 ESL GPU 設計移植到實際 RTL 層級開發，並在 RTL 層級上驗證結果的正確性以及符合 ESL 開發下的執行時間，打造屬於全台灣第一顆運算型 GPU。

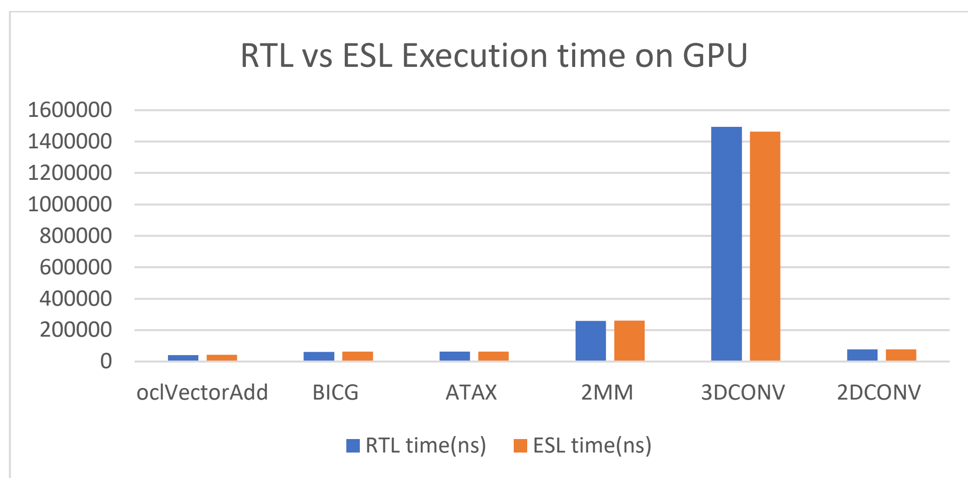


Fig.19: RTL vs ESL Polybench Execution Time

附錄

[1] 展示 LeNet-5 Inference on XGPUsim，左半部為本團隊 XGPUsim 硬體，右半部為要執行之 TensorFlow LeNet-5 inference，可以看到當程式在執行時，我們能觀察硬體相對應的行為。以影片這個案例，我們輸入資料為 9，最後 inference 資料也是 9 的機率最高。
https://www.youtube.com/watch?v=A7KXUoQ2RZ4

[2] 展示 LeNet-5 Training(5000,28*28) on XGPUsim，訓練 5000 張 28*28 pixel 圖片，Batch 為 50，Iteration 為 100 下去做 LeNet-5 Training，可以看到花費了 16 天將準確率 tune 到 0.30，可以看到準確率慢慢提升，完全符合以其他商用大廠 GPU 訓練時的訓練準確率進程。從這 16 天中我們可以驗證，CASLab GPU 在硬體微架構與軟體層上都有緊密的配合與互動，才能在這大量的運算過程中正常運作，維持訓練準確度的進展，驗證本團隊 GPU 的正確性。https://www.youtube.com/watch?v=a3E3s7TgYTA

[3] CASLab GPUsim github 網址，使用者能在上面跑 CASLab GPUsim，看到 OpenCL 程式在 GPU 上執行狀況。https://github.com/caslab-NCKU/CASLab-GPU-SIM

[4] 此為 CASLab GPU-Lite 於 FPGA 上的驗證流程展示 包含了 pre-sim, post-sim 的模擬結果，以及利用 ProtoCompiler 在 Haps-70(FPGA) 上面執行的 FPGA 驗證流程。

https://www.youtube.com/watch?v=a3E3s7TgYTA&list=PLc8R0ZIxpIdM5_yG1HdYv3D-W0fISE7sW&index=2&fbclid=IwAR2dm9ObZiOD0BjearkJyCm0gOA3nrc-5BREs0BthqAhIDu4jN43_QHeqro

[測量 TensorFlow Application Execution Time]

在 Python Inference code 中加入 TensorFlow timeline API，如 Fig. 20，此處是以測量 TX2 的 Execution Time 為例子。

```
from tensorflow.python.client import timeline
```

創造 tf.RunMetadata 格式的 buffer

將收集到的 run_metadata 轉換成 timeline 格式並且寫入到 json 檔案

```
if __name__ == '__main__':
    # Choose test image in mnist data set
    IMAGE_NUM = 2
    run_metadata = tf.RunMetadata()
    mnist = input_data.read_data_sets('/tmp/tensorflow/mnist/input_data', one_hot=True)
    test_images = np.array(mnist.test.images)[ IMAGE_NUM-1):IMAGE_NUM+test_amount, :]

    infer = inference()
    result = infer.predict(test_images)
    trace = timeline.Timeline(step_stats=run_metadata.step_stats)
    with open('timeline_tx2.json','w') as trace_file:
        trace_file.write(trace.generate_chrome_trace_format())
```

test amount 設定要 inference 的圖的數量

```
def predict(self,image):
    with tf.device('/cpu:0'):
        self.saver.restore(self.sess,self.parameter_path)

    with tf.device('/gpu:0'):
        predition = self.sess.run(
                        self.lenet.prediction,
                        feed_dict={self.lenet.raw_input_image: image},
                        options=tf.RunOptions(trace_level=tf.RunOptions.FULL_TRACE)
                        run_metadata = run_metadata
                        )
```

將 session 內的 trace_level 設定為 FULL_TRACE
將 session 內的資料記錄到 run_metadata 這個 buffer 內

Fig.20: 修改 Inference Code 流程

當 Inference 一張圖時，時間為 1.74 秒，但是所得到的時間是 Execution time = Compile time + Compute time，若要消除 Compile time，需要將整個 prediction 跑兩次，因為第二次的執行會使用到剛剛已經編譯好的 kernel，Fig.21 為記錄第二次的 Execution time。

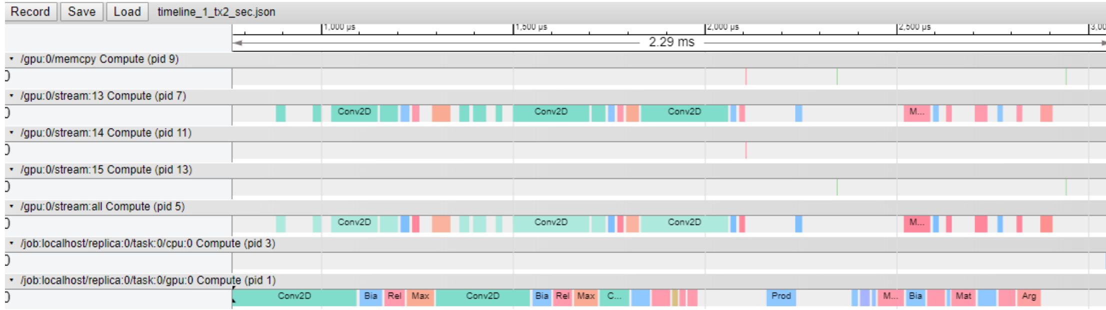


Fig.21 Inference 第二次 Execution Time

第二次 inference 一張圖時間為 2.29 ms，但有可能還有其他 runtime 因素影響，為求準確將 inference 多張圖的時間平均，得到 Fig.22 斜率為 0.186 (ms / per image)，所以平均 inference 一張圖的時間約為 0.186 ms。

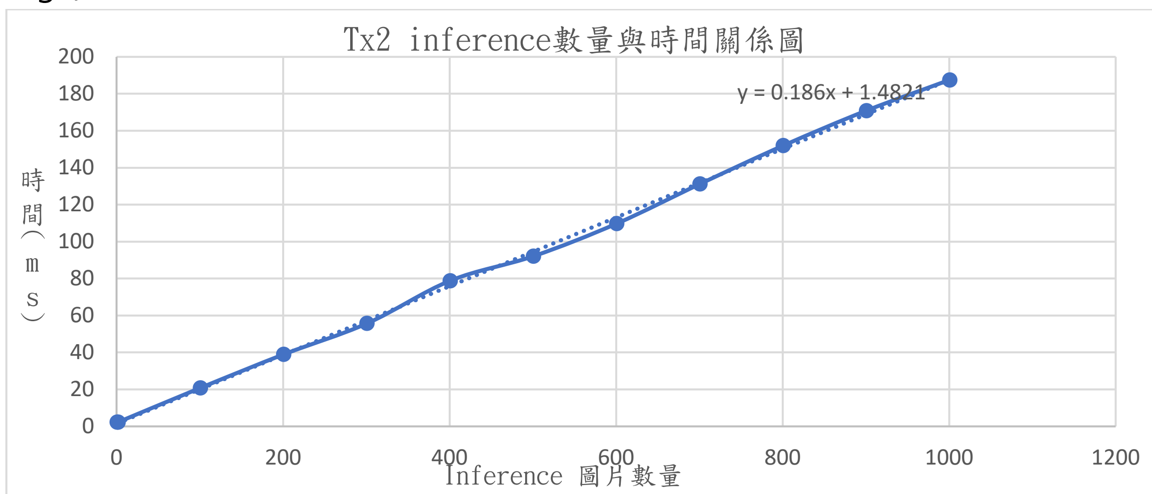


Fig.22: Tx2 inference 圖片數量與時間關係圖

Micro-Darknet for Inference for AI ASIC Design

國立成功大學電機工程系 Computer Architecture and System Laboratory

IC 設計業者關心 AI 晶片本身的開發，尤其「AI 晶片最大瓶頸在功耗及記憶體頻寬」的問題。神經網路推論程式 Micro-Darknet for Inference (MDFI)為 C-code based DNN 運算程序，加入功耗模型後，即可驗證各式加速器設計的耗能結果，如 Input data reuse, weight reuse, output reuse 不同設計方案的成果。更可支援業者在 Digital MAC, or Analog MAC, Compute-in-Memory 等前瞻議題的研究分析，協助業者降低研發 AI 晶片的費用與縮短開發時程。

一、技術使用現況及產生效益

運用於邊緣計算(Edge Computing)的 AI-on-Chip ASIC 設計需要考慮 DNN 應用模型的運算要求，提供晶片性能表現的系統層級分析，如 Fig. 1 所示:

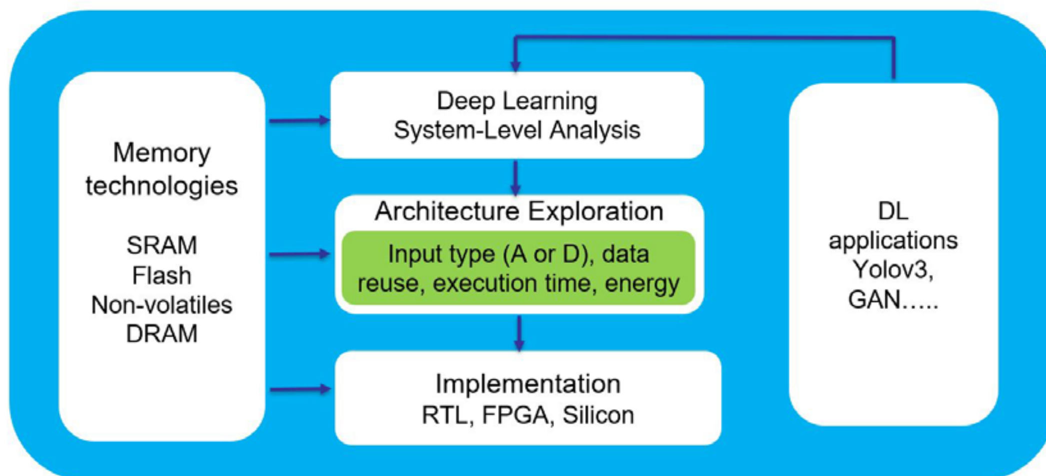


Fig. 1: Deep Learning Accelerator ASIC Design System Framework

AI Accelerator 運算電路不論是數位或類比型態，需結合可能的記憶體技術進行整體性能的 Architecture Design，這個工作更需結合各式的 DNN 應用如 Yolov3, Yolov4 等，以進行 AI 加速器的設計與印證。

本技術「Micro-Darknet for Inference」: MDFI 為一神經網路推論程式，是純 C code 的 Neural Net Inference Framework，支援如上圖所示的 AI-on-Chip ASIC 開發流程，業者可以使用 MDFI 來發展 AI-on-Chip ASIC 的軟硬體合併設計方案和驗證平台，設計與評量人工智慧推論運算硬體的執行效能。

開發者可以根據 MDFI ESL (Electronic System Level) 範例， 如 Fig. 2，開發任一種 AI-on-Chip 晶片，運作 Darknet 所支援的各式神經網路 Model 或自行開發設計的 CNN 神經網路模型，進行其軟硬體協同模擬設計與效能評估。

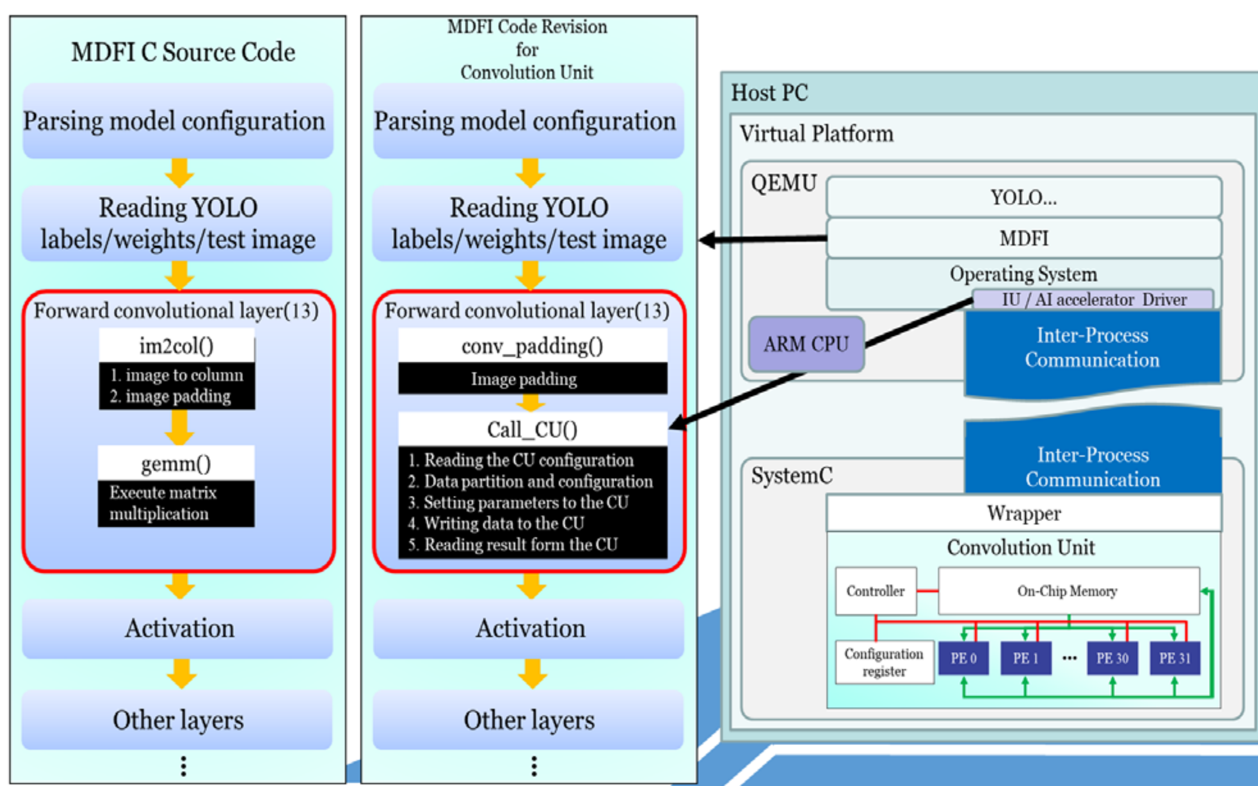


Fig. 2: MDFI 技術使用範例

MDFI 所支援的運算格式包含浮點與 int-8-16-8 的資料格式，而支援的 DNN model 包含

1: 業者自行設計的 CNN (Convolutional Neural Net) 神經網路

2: Darknet 所支援的 Image Classification DNNs 或 Object Detection Models 如 Fig. 3 所示。

Model	MDFI	Darknet
YOLOv2 608x608	X	O
Tiny YOLO	X	O
YOLOv3-320	O	O
YOLOv3-416	O	O
YOLOv3-608	O	O
YOLOv3-tiny	O	O
YOLOv3-spp	O	O
Go	X	O
Alexnet	O	O
Darknet reference	O	O
VGG-16	O	O
Extraction	O	O
Darknet19	O	O
Darknet19_448	O	O
Darknet53	O	O
Darknet53_448	O	O
Resnet 18	O	O
Resnet 34	O	O
Resnet 50	O	O
Resnet 101	O	O
Resnet 152	O	O
Densenet 201	O	O
ResNeXt50	O	O
ResNeXt101	O	O
ResNeXt152	O	O

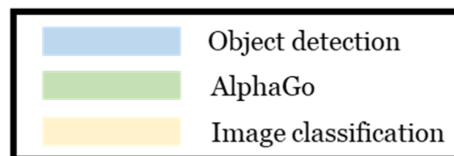


Fig. 3: MDFI 所支援的 DNN models

二、技術應用廣度及市場規模

AI 晶片應用領域廣泛，國內外軟硬體科技大廠皆大量投入資源掌握發展先機。根據 Gartner 預估，2018 年 AI 晶片市場產值已成長至 10 億美元，至 2022 年將達 132.5 億美元。根據 Allied Market Research 發布的一份報告，2017 年全球機器學習晶片市場規模約 24 億美元，預計到 2025 年這一市場規模將達到約 378 億美元，複合年增長率為 40.8%。就收入而言，目前主流的 4 種 AI 晶片類型有 CPU、GPU、FPGA、ASIC，報告預計 AI ASIC 的收入未來將超越 GPU。目前加速器運算 MAC 以數位設計為主，預計未來與現有前瞻設計可配合各式不同的 Non-volatile 記憶體進入類比方式，如 Fig. 4 所示:

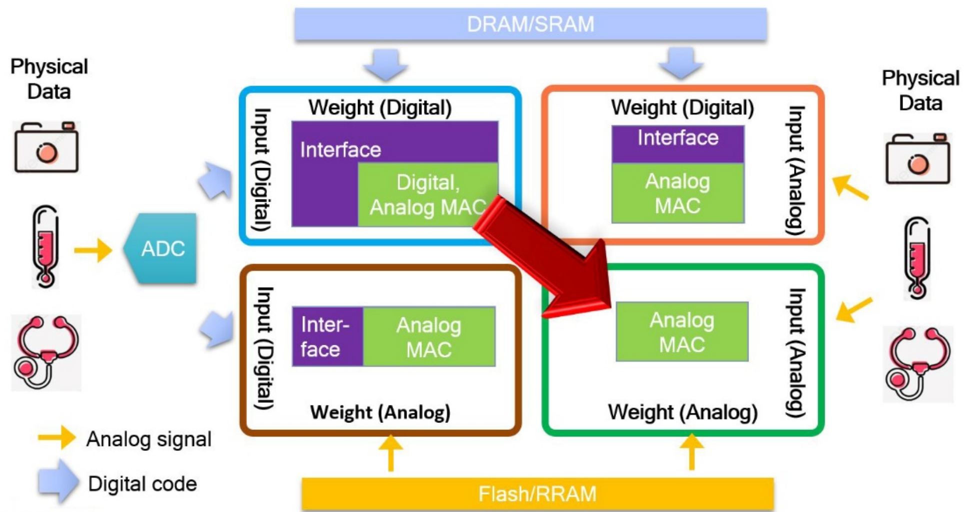


Fig. 4: MDFI for AI ASIC 晶片設計技術應用廣度

三、與其他競爭性或替代性技術之強度比較

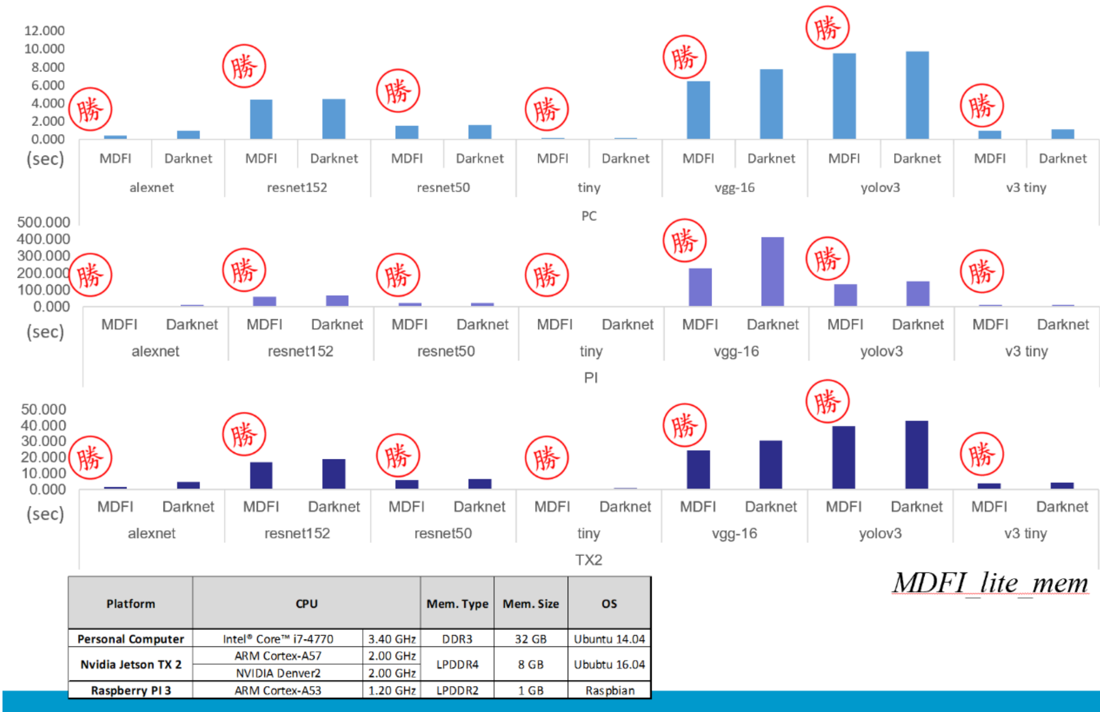
MDFI 支援 AI-on-Chip 推論運算，直接適合支援 ASIC Design，其 Code size, Compilation time 均遠優於使用其他 DNN framework 用於協助 AI Accelerator 設計， 如 Fig. 5 所示。

Framework	Feature	Code size	Compilation time(sec)
Darknet	Train/Inference	822k	13
Caffe	Train/Inference	48M	302
Tenserflow (static lib C)	Train/Inference	221M	3630
Tflite (static lib)	Inference	1885k	231
* MDFI -Os	Inference	155k	3
勝 * MDFI -Ofast	Inference	278k	4
* MDFI_lite_mem	Inference	279k	4

* MDFI -Os : “optimize for size”
* MDFI -Ofast : “Disregard strict standards compliance”
* MDFI_lite_mem : “Layer-wise memory management version, and compile with -Ofast”

Fig. 5: MDFI 為一 light weight inference C code 直接用於 AI 加速器硬體設計

MDFI 為一經過優化的 Light weight inference C code，其執行效能勝過原始的 Darknet Framework 如 Fig. 6 執行時間之比較。



Platform	CPU		Mem. Type	Mem. Size	OS
Personal Computer	Intel® Core™ i7-4770	3.40 GHz	DDR3	32 GB	Ubuntu 14.04
Nvidia Jetson TX 2	ARM Cortex-A57	2.00 GHz	LPDDR4	8 GB	Ububtu 16.04
	NVIDIA Denver2	2.00 GHz			
Raspberry PI 3	ARM Cortex-A53	1.20 GHz	LPDDR2	1 GB	Raspbian

Fig 6: MDFI 與原始 Darknet 執行 Inference 的程式執行 CPU 時間比較

MDFI 技術與其他 Open source DNN tool kits 具互補性。 MDFI C code 讀入神經網路的 Configuration 與 Weights 檔， 其讀入格式是 Darknet 格式，直接以 Sequential Layer 方式描述一 DNN，因此 MDFI 可與其他 DNN Design Tool Kits 銜接如 Fig. 7 所示，透過一個 IR to Darknet Converter 將 Intel OpenVINO 的 Frontend 銜接到 MDFI C code，直接支援經過 OpenVINO Model Optimizer 優化過的神經網路。

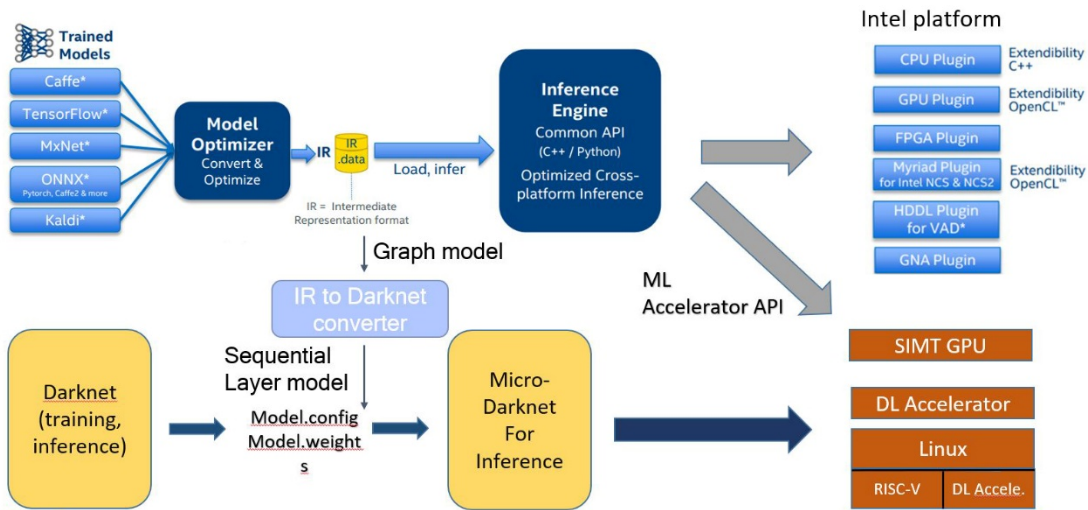
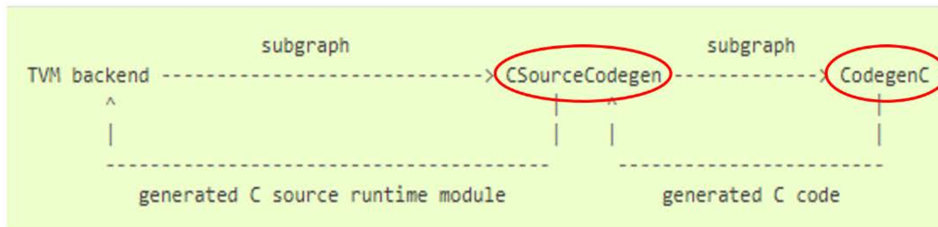


Fig. 7: 透過一個 IR to Darknet converter 將 OpenVINO 的 Frontend 銜接

而透過 MDFI 設計的加速器可在 TVM 的 Backend 設計其 C Codegen 功能, Fig. 8，直接與 TVM 前端工具鏈銜接。



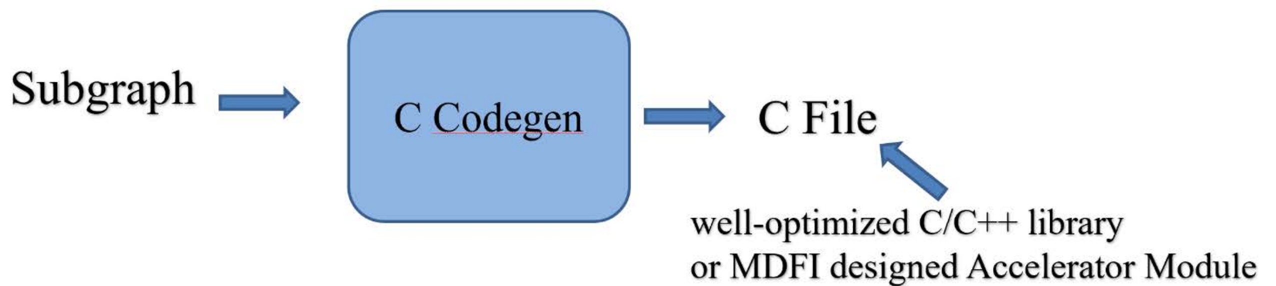


Fig. 8: Add a Hardware Backend in Tensor Virtual Machine (TVM).

MDFI 支援成大 AI-on-Chip 團隊在 Digital MAC, Analog MAC, Compute-in-Memory, Design Framework 等前瞻議題的研究分析，歡迎業界洽詢合作與使用。