國立清華大學 電機工程學系

實作專題研究成果摘要

FPGA Implementation of Visual Odometry by Using High-Level Synthesis

使用 FPGA 及高階合成技術
應用於視覺里程計

專題領域：資工領域

組別：第 A288 組

指導教授：賴瑾(Jiin Lai) 教授

組員姓名：林奕杰、李承澔、鄧文瑜、林妤諠、郭朝恩

研究期間：112 年 6 月 30 日至 112 年 5 月 8 日止，共 12 個月

一、報告摘要

本專題旨在運用高階合成(High-Level Synthesis，HLS)技術進行硬體加速，以提高開發效率並簡化設計流程。相對於傳統使用 Verilog、VHDL 等硬體描述語言進行開發，HLS 技術能夠將高階程式語言(C、C++、SystemC)快速轉譯為 RTL，並透過不同的優化產生更高效能的硬體架構，讓設計者能夠更快速地開發複雜的硬體系統。

在此專題中，我們使用 Xilinx 的 Vitis 開發系統，將 OpenCV 的演算法改寫，利用 HLS 技術實現視覺里程計（Visual Odometry，VO）演算法。我們將使用電腦視覺、系統設計、硬體設計、HLS 開發流程、HLS 技術等相關知識，以實現 VO 加速器於 FPGA 的高效即時運算。

二、報告內容

1. 背景/動機

視覺里程計（Visual Odometry，簡稱 VO）是透過視覺感測器，如攝影機，來估測相機在空間中的位置和運動軌跡的技術。VO 透過從相機捕捉的一系列圖像中提取特徵點，並通過計算特徵點之間的運動，從而估計相機拍攝的運動軌跡。VO 已廣泛應用於自主車輛、機器人和增強現實等領域，可以幫助車輛或機器人在未知或複雜環境中進行定位和導航，也可以用於增強對場景的三維建模和定位。近年來，研究人員探索了各種硬體加速方法，以增強視覺里程計演算法的運算效率。

FPGA 具有可重構性、低延遲、高平行化等優點，因此被廣泛應用於許多領域，包括數位信號處理、計算機視覺等。而 High Level Synthesis 則是一種將高階程式語言轉換為硬體設計的技術，相對於傳統的硬體描述語言設計，High Level Synthesis 可以大大簡化設計流程並提高硬體設計的效率和品質。

2. 研究目的

本研究旨在運用 High Level Synthesis 技術將 Visual Odometry 演算法實現於 FPGA 上，加快視覺里程計在 FPGA 的硬體開發流程，未來更能提高其運算效率和實時性。

3. 研究方法

3.1 視覺里程計演算法架構：

VO 演算法由四個子算法建構而成，分別為深度估計 (Stereo Matching)、特徵點擷取 (Feature Extraction)、特徵點配對 (Feature Tracking)、以及位移估測 (Motion Estimation)。

Stereo Matching 分析同一時間點左右視覺的圖片(Img_Left0、Img_Right0)及相機參數(P0、P1)，計算出視差(Disparity)，進而得到視覺深度(Depth)。

Feature Extraction 則對左相機位移前後擷取的連續圖片(Img_Left0、Img_Left1)提取特徵點(KP0、KP1)以及其描述子(Des0、Des1)。

提取特徵點後，Feature Tracking 以描述子比對特徵點從而形成配對 (Matches)，並以給定閾值(Filter)去除偏差配對。

最後 Motion Estimation 利用深度、配對、相機參數，計算出旋轉矩陣(rmat)以及平移向量(tvec)，並合成最終軌跡轉移矩陣(Tmat)。

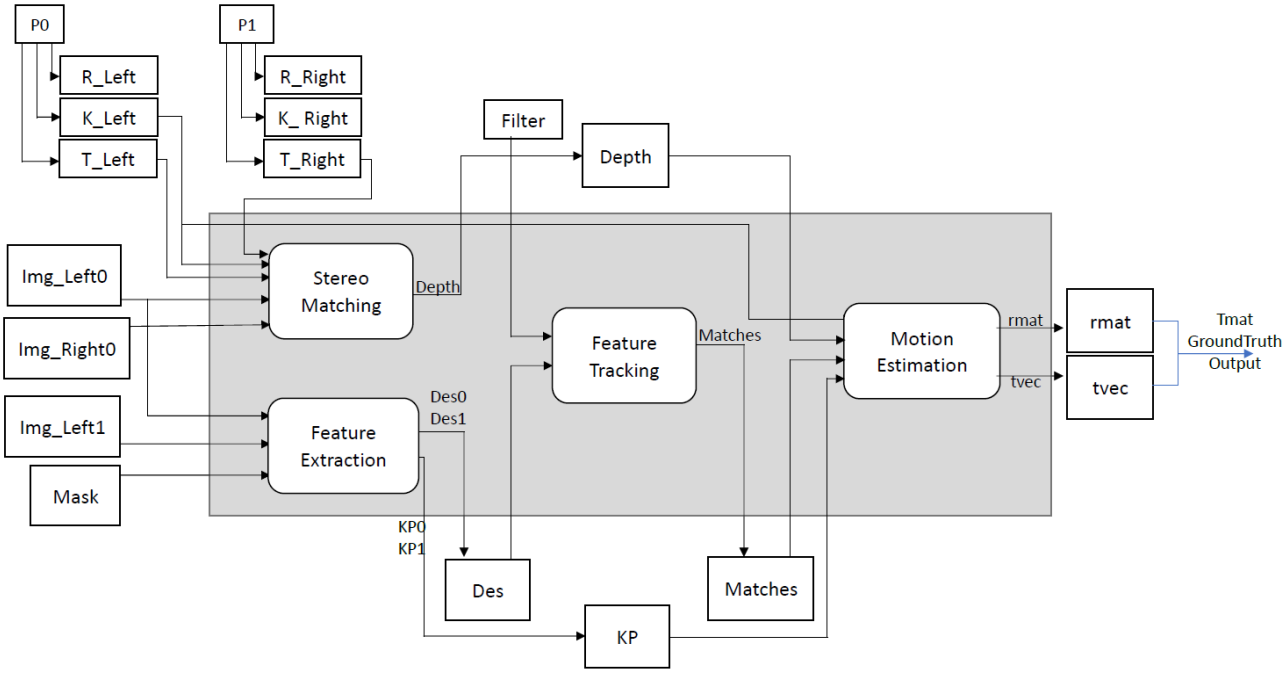

Figure 1 Visual Odometry Block Diagram

3.2 HLS 應用加速程式組成架構：

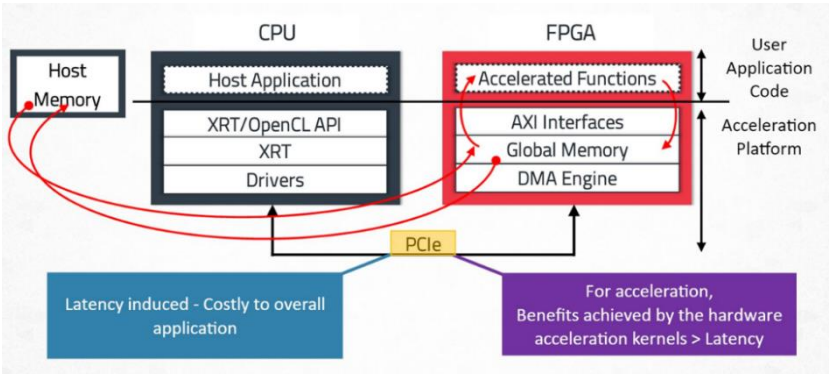

Figure 2 Host-Device Architecture

軟體程式以 C/C++編寫，執行於傳統 CPU（x86 或 ARM）上。其次，軟體程式使用由 Xilinx RunTime（XRT）或 PYNQ 函式庫實現的 user-space API 與 FPGA 中的 kernel 進行資料傳輸。

硬體加速 kernel 可以用 C/C++或 RTL（Verilog 或 VHDL）撰寫，並在 FPGA 上執行。這些 kernel 使用標準的 AXI interface 與 Vitis 硬體平台集成。通過 AXI-PCIeIP，PCI 加速卡使用 PCIe 總線與 x86 服務器傳輸資料。

在本次的專題中，我們使用 Kitti_visual_odometry 作為 Code Base，並以此為參考將其呼叫的 OpenCV API 轉為 self-contained library，自行改寫內部程式並測試，接著再改

寫為 High-Level Synthesis code，使用 Xilinx Vitis 系列軟體做軟體及硬體驗證，最後在 u50 上執行。

3.3 實作流程：

3.3.1 使用 Visual Studio 運行：

i. 在使用 OpenCV library 的環境下運行 Visual Odometry 演算法，並以 KITTI dataset 作為測資。
ii. 區分 Host 與 Kernel 各自負責的部分。
iii. 從 OpenCV library 中找到所有演算法的 C++ source code，將所有會使用到的 function 與 class 整理出來成為 self-contained code，各 Kernel 即不再使用到 OpenCV library。

3.3.2 使用 Vitis HLS 或 Vitis IDE 運行：

i. 運行 C-simulation，Host 與各 Kernel 各自利用測資驗證正確性，Host 使用 KITTI dataset；各 Kernel 從 OpenCV 環境下輸出自己會用到的測資來使用。
ii. 運行 Synthesis，各 Kernel 改寫成 HLS 可以合成為硬體的形式，並閱讀 Synthesis Report 進行初步優化。
iii. 運行 Co-simulation，確保所有介面可以正常串接且模擬硬體運算結果正確。

3.3.3 Host 採用 OpenCL 架構：

i. Host code 使用 OpenCL 撰寫，並確認 Host 與各 Kernel 間傳輸的資料型態及介面。
ii. Host 程式除了能完整串聯所有 Kernels，也需允許各 Kernel 個別驗證。
iii. Host 程式能使各 Kernel 運行在 C code 模式或 FPGA Kernel 模式。

3.3.4 Host 與 Kernels 進行驗證：

i. Host 先個別驗證各 Kernel，且一次僅有其中一 Kernel 跑在 FPGA 上，其餘部分皆先跑在 CPU 上。
ii. Host 整合驗證各 Kernel，將所有 Kernel 串聯且都跑在 FPGA 上。
iii. 依序執行 SW-Emulation -> HW-Emulation -> Hardware，驗證每個步驟皆正確運行。

3.3.5 Host 與 Kernels 進行優化：

i. 確認資源使用量及比較時間(FPGA time v.s. CPU time)，並進行優化。

3.4 開發與驗證：

3.4.1 Host Program：

為了方便驗證我們制定並以 define macro 方式選擇以下 6 種編譯模式：

```
_PURE_C_                    // Run all function in pure C code
_ONLY_K_StereoMatching_     // Run StereoMatching kernel in PL side
_ONLY_K_FeatureExtraction_  // Run FeatureExtraction kernel in PL side
_ONLY_K_FeatureTracking_    // Run FeatureTracking kernel in PL side
_ONLY_K_MotionEstimation_   // Run MotionEstimation kernel in PL side
_ALL_KERNELS_               // Run all function in HLS code
```

˙_PURE_C_:

Host 的行為與 pure C Host program 相同，所有 function 都運行在 Host PS side。

˙**_ONLY_K_<Kernel_Name>_:**

以 Xilinx v++ compiler 編譯、OpenCL 溝通該 kernel，於 FPGA PL side 運行。其餘的 kernel function 仍是在 Host PS side 以 pure C program 運算。

˙**_ALL_KERNELS_:**

以 Xilinx v++ compiler 編譯、OpenCL 溝通 4 個 kernel，於 FPGA side 運行。

在先前單個 kernel 驗證完畢後，只要在 Host 端正確連接前後級對應的輸入輸出，即可在不修改 kernel code 的情況下同時運行 4 個 kernel，達成所有演算法都透過 FPGA 加速的目標。且 kernel 開發者不必擔心在不同的運行模式下介面定義不相同。

3.4.2 Kernels：

以下為 Kernels 的四個開發流程，每個步驟皆須確保運算結果正確：

˙Pure (self-contained) C/C++ code simulation

˙Synthesis

˙Co-simulation

˙on FPGA

對於四個 kernel 的設計流程，從 OpenCV 實作四種演算法所呼叫的 function 著手，首先將內部使用的無論數學運算、資料儲存結構，節錄/實作成完全沒有呼叫外部函式庫的 self-contained code。順利實作完成後，HLS 編譯器會出現不支援部分程式碼的情形，kernel 介面也沒有硬體定義，因此要針對可合成性修改程式，成為 synthesizable code。

接著，儘管程式已符合硬體合成需求，還是可能出現資源使用過多、運行時間過長等等的問題，此時便要研讀各項 report，針對問題做優化。當優化完成並通過模擬測試後，最後即可輸出成 binary file，交付 host program 完成應用端測試。

4. 研究結果

4.1 測試資料集：

在我們的實驗中使用 KITTI dataset，KITTI dataset 是一個被廣泛使用的計算機視覺資料集，其中包含了從一個行駛中的車輛上拍攝的影像序列、雷達數據、慣性測量單元 (IMU) 等資料。它主要用於研究和評估各種計算機視覺任務，如視覺里程計 (visual odometry)、立體視覺 (stereo vision)、物體偵測與辨識 (object detection and recognition) 等。

4.2 實驗環境：

CPU	11th Gen Intel(R) Core(TM) i7-11700 @ 2.50GHz
RAM	49112592 kB
OS	Ubuntu 20.04.4 LTS
FPGA	Xilinx u50

4.3 成果比較：

4.2.1 測試方法

由 Pure C program 及 HLS program 分別輸出 2000 張圖像的軌跡(trajectory)，各自與真實路況軌跡(ground truth)比對，並比對 Pure C program 和 HLS program 中演算法的實際行為是否相符，或是釐清誤差的原因。

4.2.2 10 組圖片的平均運行時間比較

Kernel	FPGA time (Baseline Version)	FPGA time (Optimized Version in 23/5/7)	CPU time
Stereo Matching	8290.4ms (216.3Mhz)	7880ms (190.3Mhz)	439.3ms
Feature Extraction	910.3ms (300Mhz)	321.6ms (104.7Mhz)	58.6 ms
Feature Tracking	11.6ms (300Mhz)	5.1ms (300 Mhz)	4 ms
Motion Estimation	60.6ms (300Mhz)	408.6ms (258.1 Mhz)	2.5 ms

4.2.3 2000 組圖片對應的真實路徑(Ground truth)

在 KITTI Dataset 中，XZ 平面解釋為生活當中的 2D 水平面(長度、寬度/前後、左右)；Y 軸則是解釋為海拔高度。由於此資料集是在相對平整的平面路段採集而成，無過多不規則的高低起伏，因此在後續的結果分析應主要聚焦於 XZ-plane。

4.2.4 結果

以下分別為 **Ground truth (**Blue**)** v.s **Pure C program** (Orange) v.s **HLS program (**Green**)** 跑出來的路徑圖：

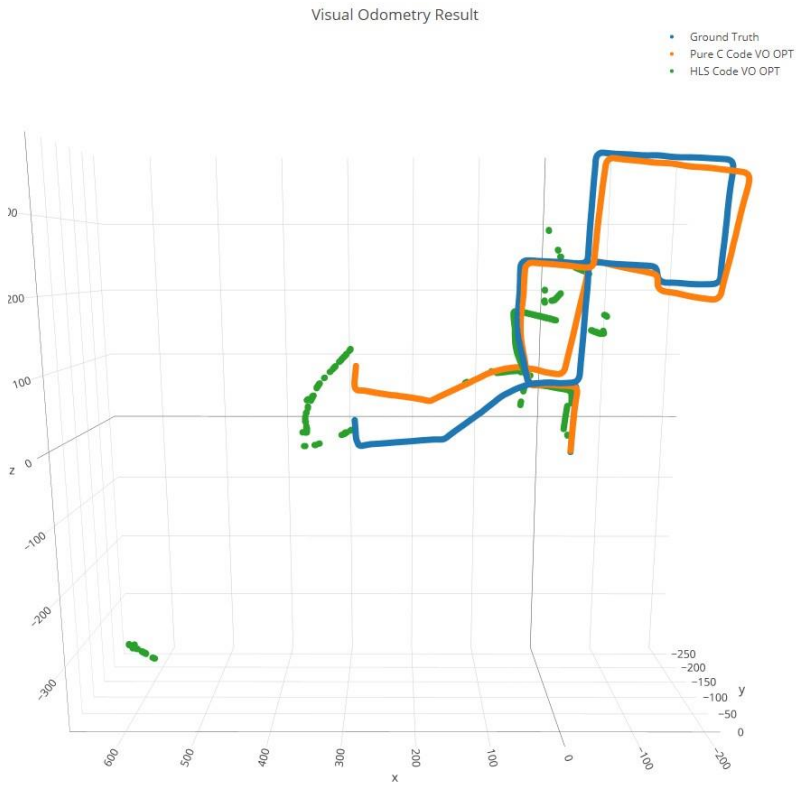

Figure 3 Trajectory Comparison

4.4 結果分析：

4.4.1 Ground truth 與 Pure C program 的結果誤差

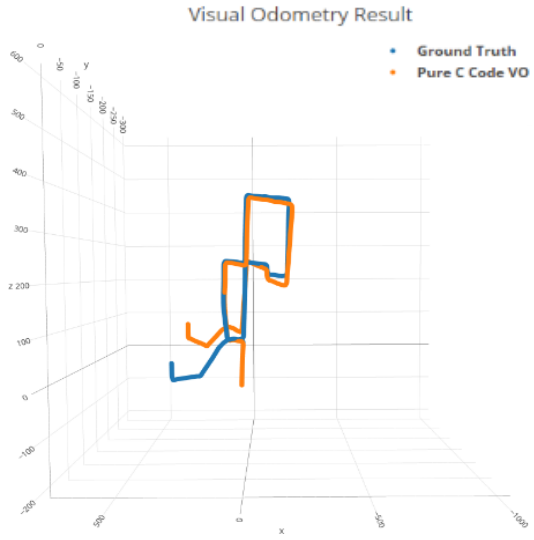

Figure 4 Track discontinuity and matching segment between ground truth and pure c program

Figure 4 中 Pure C program 計算出的軌跡與真實路徑(Ground Truth)間存在些微偏轉，此為 VO 演算法中已知難以避免的問題，因此後續應著重比較 Kernel 與 Pure C program 計算出的軌跡。

4.4.2 Pure C program and HLS program 的結果誤差

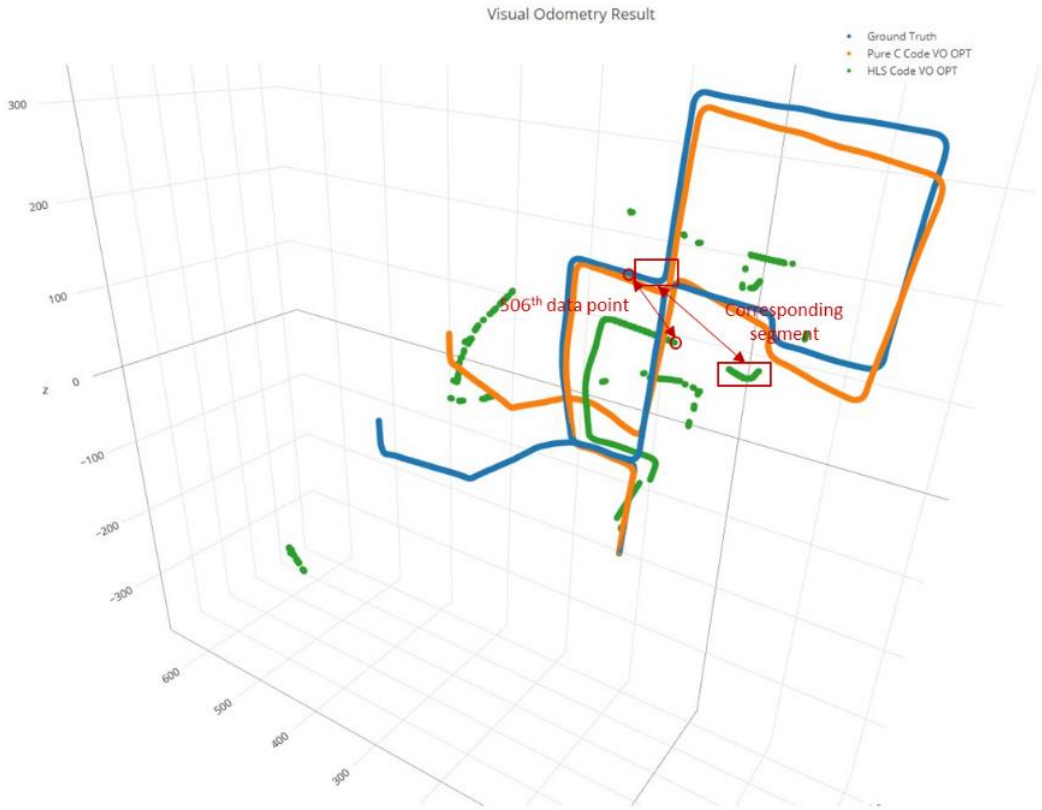

Figure 5 Track discontinuity between HLS program and pure c program

在本階段 kernel 開發階段時重新加入部分演算法以因應轉彎的路況變化。

由 Figure 5 可看出，optimized kernel 在約前 506 個數據點軌跡與 Pure C program 的軌跡相似。接著即因誤差累積過大，後續數據點開始出現不可辨識的軌跡與斷點。可估計的路徑長度較前一階段 baseline kernel 多出近 400 個 iteraions。

4.5 結果討論：

4.5.1 誤差的原因

(1) 誤差累積

由於 VO 具有誤差累積性:

$$e_i = {}^{i}T_{i+1} \cdot {}^{i}\hat{T}_{i+1}^{-1}$$

意即一旦誤差發生，只要計算的路徑越長，最後的誤差也會因累積而隨之增大。

(2) 斷點形成的原因

從圖中可見路徑出現斷點的原因主要來自於 Motion Estimation kernel 中使用的 Levenberg-Marquardt (LM)迭代優化演算法。這種演算法面臨著陷入 local minima 的問題，其原因與迭代的起始點有關。在優化過的 Motion Estimation kernel 中，LM 的起始點來自三種不同的組合，但這些結果仍有可能使結果陷入 local minima，尤其是在動作變動劇烈時，例如轉角等情況。接著便可預見，計算出的結果超出預期許多，這便是出現斷點的主因。

(3) 未來優化方式

我們觀察到 ground truth 一開始並不是走座標軸 z 直線的方向，因此未來可以在系統的最後加上校正的實作，我們覺得可以採用 Bundle Adjustment 的技術修正，達到更精確的三維重建效果，以最小化在所有影像中的重投影誤差。

4.5.2 FPGA 速度較 CPU 慢的原因

(1) Common Reason:

˙**未做到最高頻率運算：**

目前頻率未能做到最快的 300MHz，觀察 report 可以發現還有許多造成 timing violation 的運算。

Future work: 將所有無法達到高頻的運算修改以達成 300MHz。

˙**讀取資料未完整做到 burst：**

在 kernel interface 部分，並未對 m_axi 做 max burst length、number of outstanding 的測試，使得僅僅在讀取資料的表現，kernel 就輸給有做完整 caching 的 CPU。

˙**未優化完全及未高度平行化運算：**

目前尚無法使整個演算法以 dataflow 的形式計算，且有些運算還可以再修改成能夠做到 pipeline 及 array partition 的形式，使運算高度平行化。

Future work: 改寫變數型態與運算方式以達成更有效的運算。

(2) Stereo Matching

˙**尚無實作出 cache 的分配資源寫法：**

由於內部演算法計算複雜，且有許多層迴圈的結構，在 cache 的實作上有一定的困難度；且在有時間壓力的情況下，目前並未採取此解法改寫。

Future work: 使用聰明的資源分配技巧將資料 cache 起來，以解決資源不足所導致的成果誤差並縮短運行時間。

˙**切圖運算需呼叫多次：**

將圖片切成三等分後，在內部 kernel 呼叫了五次該演算法。因此在時間上比起原先只呼叫一次演算法、且不用進行後面疊圖計算的時間還慢上許多。

Future work: 優化演算法使 buffer 使用量縮減以達到不需切圖與疊圖的運算。

(3) Feature Extraction

˙**未符合硬體適合的運算形式：**

有許多運算可以做 Divide and Conquer 並修改成能發揮硬體優勢的運算方式，而我目前所做的修改未能達成最好的硬體運算效率。

Future work: 重新改寫成硬體能高效運算的程式。

˙**FPGA 資源限制：**

原先應可以在偵測 keypoint 時同時做 GaussBlur 的運算，然而存放圖片的 BRAM 有限，導致無法同時進行運算。

Future work: 優化資源使用量並改寫成並行運算。

(4) Motion Estimation

˙**新增演算法：**

除了實作原始 OpenCV 中演算法，也額外增加演算法以優化運算結果，例如：比較多組先前算的結果做誤差計算，因此新增的計算步驟，導致整體的運行時間增加。

Future work: 優化演算法，以降低運行時間。

5. 總結

本次實作成功將 VO 演算法應用於 FPGA 上進行運算。在不斷迭代優化後，除 Motion Estimation 為增加路徑估計精準度而增加用時，已大幅縮短整體 FPGA 的運行時間。儘管其運行時間仍不及 CPU，但與最初版的運行時間相比已有顯著進步。

由於 self-contained C code 計算出的軌跡與真實路徑(Ground Truth)間的偏差實屬 VO 演算法已知難以避免的問題。因此我們著重比較 Kernel 與 self-contained C Program 計算出的軌跡，可發現前 506 個 iterations 中，兩者具有相似的軌跡，說明我們的系統在較短的路徑內可以做出相對準確的軌跡估計。

在此次專題實作中，即便硬體資源與 CPU 平台相比較為貧乏許多，我們仍成功在 FPGA 上開發出整套系統。我們學習到如何應用 HLS 技術於系統的設計開發、驗證硬體行為，並找到可能導致結果不如預期的原因，未來將進一步持續平行優化。

三、心得感想

為了這兩學期的實作專題，我們從去年暑假開始學習相關知識，從開放式課程與教授的上課錄影、研讀論文，到最後自行實作。一路上我們小組克服了許多障礙，首先將 OpenCV library 的 project 轉為 self-contained 的 code 就是一大困難，接著無法將 C code 做 Synthesis、硬體資源限制無法成功合出硬體都修改了好久。最後測試路徑時，一開始誤差也極大，修改了好久才縮小誤差！經過了一年的努力，我們從一開始如何寫 High-Level Synthesis Code 都不會，到最終讓整個演算法可以放到 FPGA 上並成功運行，當看到成功運行的那一刻是很欣慰的，覺得努力總算有了成果，也讓我們得到許多成就感。

這一年來，我們學到很多，也進步很多，因為要將許多上課知識實際應用到專題上，包括 C++程式、計算機結構、邏輯設計實驗等知識的結合，不單單只是上課學習理論而已，也讓我們發現自己還有許多能力不足之處。

每周末都和教授與助教討論、與組員們合作，一步一步更新進度，儘管過程常常碰到困難，以為自己要做不出來了，但最後也都一一克服！雖然成果還有很多進步空間，但這一年來是很充實的，也學到視覺里程計算的演算法、整套 HLS 開發流程、HLS 的技術、團隊合作的能力，希望未來能夠帶著所學持續進步！

最後非常謝謝教授、助教及組員們一年來的指教及協助！