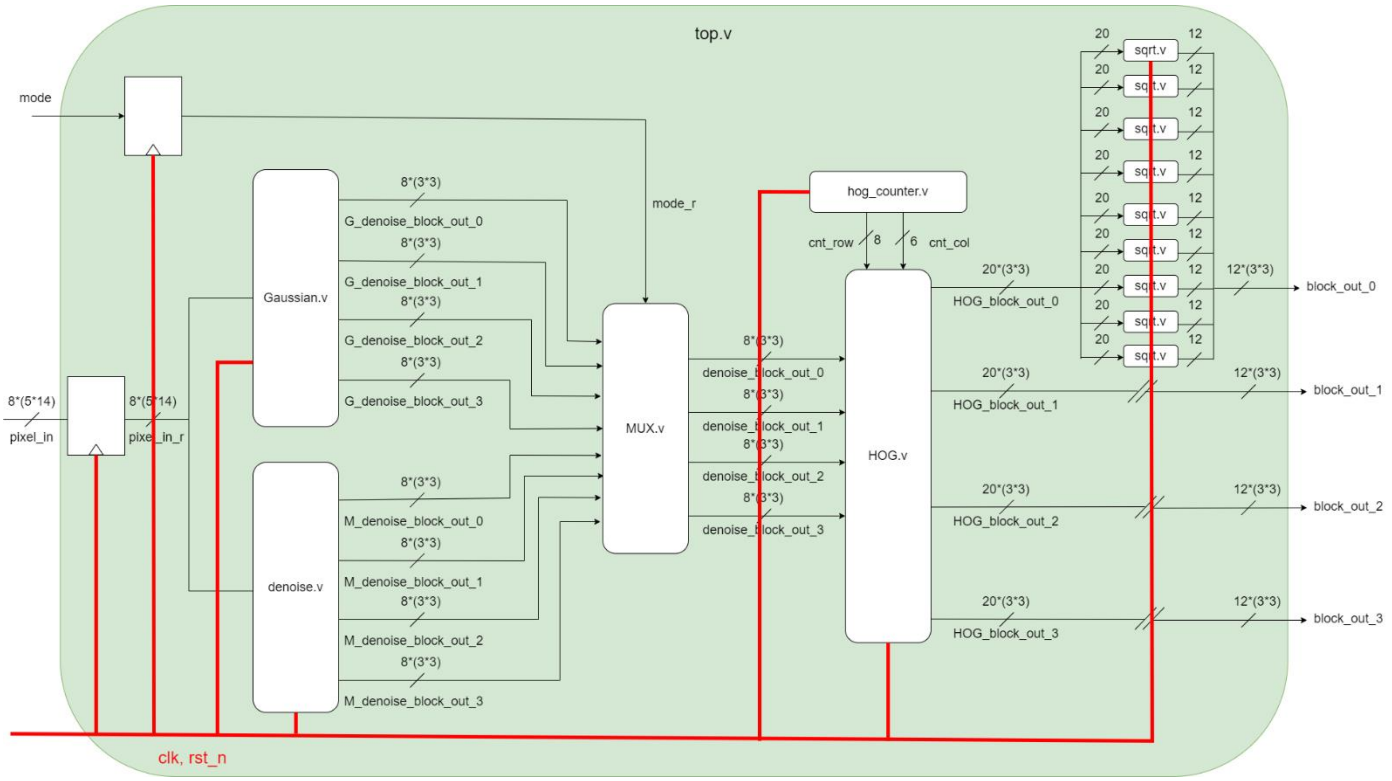
Implement Edge Detection on Images using HOG Algorithm with Some Noise Filters

利用方向梯度直方圖演算法及雜訊濾波器實現圖像邊緣檢測

Lin Yung Hsieh[1](謝霖泳)
National Tsing Hua University
107061218

I Jieh Liu[1](劉亦傑)
National Tsing Hua University
107061212

Xuan Jie Lin[1](林暄傑)
National Tsing Hua University
107061246



圖一、block diagram

A. Functionality and Block Diagram

本組聲明：本 project 中，濾波器及 HOG 的概念是 survey paper 得到的，其餘所有的 RTL 及 Python script，都是我們為了本堂 IC Lab 的 final project 從無到有一點一滴發想、一行一行慢慢打出來的，**完全沒有**使用任何網路上的開源程式碼，也沒有用到任何之前課堂做過的 project，僅此聲明。

我們最後實現了 proposal 中所提及之大多數功能，共完成了 **HOG (包含 square root 計算)、median filter、Gaussian filter**。

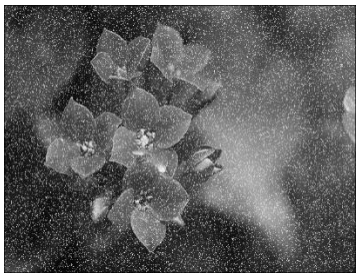
我們每次輸入 5*14 個 pixel，每個 pixel 為 8-bit，因此一次輸入有 5*14*8 = 560-bit，將這些 input pixels 分別輸入 Gaussian.v 及 denoise.v 利用 Gaussian Filter 及 Median Filter 進行雜訊去除，出來的結果送進 MUX.v，利用 input mode 選擇我要哪種 filter 的輸出，並將去除雜訊的 pixel 送進 HOG.v 中進行 HOG 演算法。最後，將每一個 output pixel 送進 sqrt.v 中進行開根號並輸出。

我們將圖片中每個 3*3 的 pixel 稱為一個 block，所以每個 block 有 8*(3*3)個 bits。每個 cycle 中，HOG 都會算出 4 個 block 的 output，也就是會有(3*3)*4 個 pixel 輸出，對於每個 pixel 都會先將 HOG 的值傳入 sqrt.v 中進行開根號再輸出。因此，上圖中右邊可以看到

HOG_block_out_0 的 9 個 pixel 均會被送入 sqrt.v 進行開根號運算後才會進行輸出，而 HOG_block_out_1 ~ HOG_block_out_3 也同理，所以右邊應該會有 36 個 sqrt.v，上圖中僅畫出 HOG_block_out_0 後面的 9 個示意，其他三個 HOG_block_out 則以兩條斜線省略。至於 HOG 運算完的值需進行開根號才能輸出的原因，和原始演算法的設計有關，將在後面 related algorithms 段落說明。

B. Input Data Generation (OpenCV)

我們先將圖片 reshape 成 638*482 的圖片，並利用 random 產生 30000 個隨機變數，作為 noise 的位置，並將這些位置的 pixel value 設為 FF 作為雜訊，也就是下圖中的白點，並將這些 pixel 以 hexadecimal 的形式寫入 **noise_638_482.txt**，作為 input data。



圖二、帶雜訊的圖片輸入

C. Related Algorithms

1. Median Filter 中值濾波器

Median filter 原始的概念是對每個 pixel 值都取以之為中心的 9 個數的中位數，這樣一來，就算這一個位置是雜訊，也有很高的機率被正常的 pixel 取代掉，進而達到去除雜訊的目的。

而在設計上對於「找中位數」做了一些調整，由於找真實中位數會過度浪費資源以及耗費時間複雜度，因此最後決定使用 median of medians 算法，並且設定 3 個值為一個 group。此算法是一種 selection algorithm，並可以找出近似的中位數值，主要概念就是利用分群(e.g. 3 個一組、5 個一組)，比較大小來找出近似的中位數，再把這些中位數們去再跑一次 median of medians 算法，到最後就能獲得一個代表全體的 median。

2. Gaussian Filter 高斯濾波器

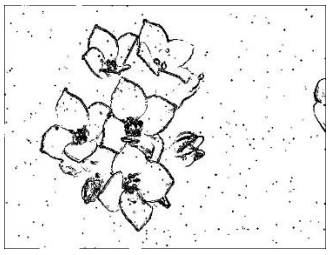
原始 Gaussian filter 會使用 exponential 的運算，而我們經過蒐集資料及討論後，為了硬體上的實作方便，決定以近似 3*3 Gaussian Filter 的 Generalized weighted smoothing filter 矩陣(下稱 3*3 Gaussian filter)如圖三，來作為我們的主要算法，圖像與 3*3 Gaussian Filter 做 convolution 將會達到濾除雜訊、低通、模糊化的效果。

$$\frac{1}{16}\begin{bmatrix}1 & 2 & 1\\ 2 & 4 & 2\\ 1 & 2 & 1\end{bmatrix}$$

圖三、3*3 Gaussian filter

3. HOG 方向梯度直方圖

對於每個 pixel，定義其 x 方向的梯度 g_x 為其上方 pixel 與其下方 pixel 的差值，其 y 方向的梯度 g_y 為其左方 pixel 與其右方 pixel 的差值，並定義這個 pixel 的 total gradient $G = \sqrt{g_x^2 + g_y^2}$。當這個 pixel 的上下顏色差異愈大，或左右顏色差異愈大時，G 值就會愈大，也就意味著這個 pixel 可能是圖中的 edge，藉此達到 edge detection 的效果。圖四是我們將圖一去雜訊後的圖經過 HOG 運算後，G 值大於某個 threshold 的 pixel 描出來，可以看出的確有達成 edge detection 的效果。



圖四、HOG 實際效果

D. Specification

- I/O definition
  - ✓ image: 638*482 pixels
  - ✓ input: 70 pixels / cycle (each pixel is 8-bit)
  - ✓ output: 36 pixels / cycle

- Target performance
  - ✓ timing: 3 ns
  - ✓ area: 750,000 um^2
  - ✓ power: 50 mW
  - ✓ cycle count: 10000
- Synthesis performance
  - ✓ timing: 3 ns
  - ✓ area: 504,438 um^2
  - ✓ power: 31.8mW
  - ✓ cycle count: 8487
- Chip performance (post-layout performance)
  - ✓ timing: 5 ns
  - ✓ area: 562,169 um^2
  - ✓ power: 28.3 mW (ICC)
  - ✓ cycle count: 8487
  - ✓ core utilization: 0.7

E. Implementation

首先，我們會將圖片進行 reshape，並於每個 cycle 將 70-bit (5*14)送入第一階段的 denoise stage，這個 stage 將會有兩種去雜訊的 filter，分別為 median filter 和 Gaussian filter。接著，去除雜訊後的輸出將會送進 HOG.v 進行 edge detection 的運算，而這也是我們本 Final Project 最核心的功能，也就佔據了最多數的運算資源及面積。最後，會將 HOG.v 的輸出送至 sqrt.v 進行開根號的運算，來得到利用梯度進行邊緣檢測的最終輸出結果。

denoise.v 實作了 median filter 的電路，由於我們設計上一次讀進 36 個像素(3*12)，但為了方便處理，在送進 filter 之前會進行 padding 來讓所讀到的 block 大小實際上是 70 個像素(5*14)，如此一來在運算上就可以比較 consistent，而最後就能獲得 36 個像素的輸出(3*12)，因此在下一個 HOG.v 的階段會較好配合。

最核心的運算概念就是利用 comparator 來找到某一個 block 的 median (也就是 9 個像素中的 median)。因此我利用 divide-and-conquer 的概念，設計 median.v 去找出 3 個像素中的 median，並找出三組 median 的值之後，再組合去找出真正對 9 個像素來說的 median。因此，硬體實作上則主要是利用 if-else 去判斷，合成上我們認為會利用到大量的 8-bit comparator 來實現。

至於 Gaussian.v 本質上的運算則是類似於先前作業中的 convolution，也就是利用大量的乘法、加法器來實現 Gaussian filter。而由於 HOG.v 已經耗費了主要的運算資源，我們希望盡量的節省資源，於是我們去觀察到近似 Gaussian filter 其數學形式皆為 2 的冪次方，也就意味著可以利用 Verilog 中的 concatenate operator 來避免去使用到乘法、除法，也就理論上可以節省資源的耗費，最終本 module 的面積只占總面積的不到 5%，也就達到節省資源的目的。

在撰寫 HOG 電路時，需要一定的運算資源來做減法與乘法，考慮到我們的 spec 規定的 cycle count 要在 10000 以內，我們選擇運用 72 個乘法器與 72 個減法器來實作 HOG 以達到 spec。在 timing 方面每經過一個乘法或減法就會用 DFF 擋一次，來達到最佳 timing，最後單獨合成 HOG 的 timing 在 2.5ns 左右。撰寫 HOG.v 的困難點是在邊界的點需要特別處理，邊界值在當個 cycle 沒法直接運算，必須等到下個 cycle data 進來時才能做運算，所以在電路內部必須準備 636+6 組 DFF(邊界的 row and col)來儲存一些必要資訊，所以 HOG 運用的資源佔整個電路比例最多。

sqrt.v 的部分，有許多方式可以實作，我們希望盡量降低 cycle 數以及 HOG.v 所表示的梯度輸出是 20-bit，而且我們已知 sqrt.v 的輸出為 12-bit，因此在 trade-off 的考量下最終以查表方式實作，如此一來就能以面積和數值輸出本身的精度去換取在 1 個 cycle 內快速產生運算結果。

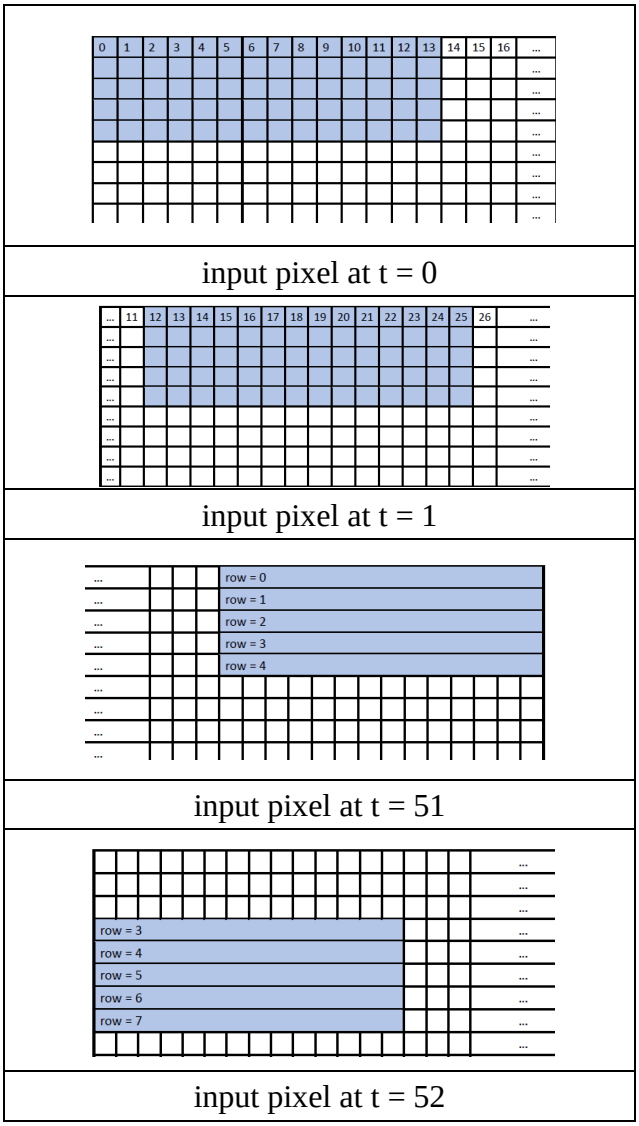
F. Verification

在我們的設計中，每個 cycle 均會輸出 4 個 block，也就是 36 個 pixel，但因為 HOG 及兩種 filter 都會運用到每個 pixel 相鄰的 pixel，為了運算方便，我們在 input pattern 的時候每

次會給(3+2)*(12+2)個 pixel，且每個 cycle 的 input 會有兩個 column 或兩個 row 的重疊，如表一所示，以確保 filter 運算之正確性。

也是因為這個原因，我們才決定將 input image 的大小設為 638*482，以確保在 input data 有 overlap 的情況下可以剛好被 cycle 數整除。換言之，每給完一整個橫排的資料需要恰 52 個 cycle，而且恰需 159 次上述 iteration 始可給完整張圖片的資料。

在表一中，可以看到 t=0 和 t=1 的 input data 有重複了 column 12 及 column 13，而當這一橫排給到最後(t=51)和下一個 cycle (t=52)相比，則重複了 row 3 及 row 4，這麼做都是為了確保 filter 運算之正確性。



表一、輸入資料重疊示意圖

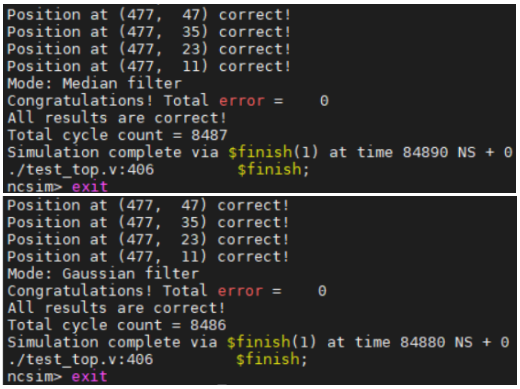
而 pre-sim、gate-sim、post-sim 我們都有設計可以測試兩種 filter 的指令，為 ncverilog -f {*simulation file*} +define+{*filter selection*}，simulation file 有 sim.f、gatesim.f 以及 postsim.f 可以選，其中 sim.f 就是 pre-sim，後方的 filter selection 可以用 MED 或 GAUSS 去測試兩種 filter，預設為 median filter。

至於對答案的部分，則是按照助教之前的方法，去讀 golden file 中相應的位置與 output 比較，看看 block_out_0 ~ block_out_3 是否均相同。然而，圖片邊界的部分因為 HOG 的值本來就無法定義，所以遇到圖片的邊界會略過邊界區的檢驗。

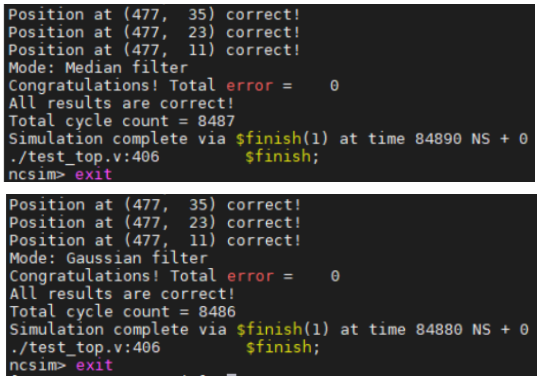
G. Result

	median filter	Gaussian filter
pre-sim	All pass!	All pass!
gate-sim	All pass!	All pass!
DRC/LVS	Clear! (error=0)	
post-sim	All pass!	All pass!

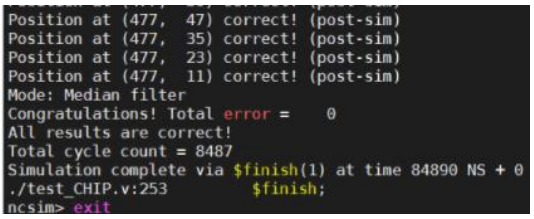
表二、完成度整理

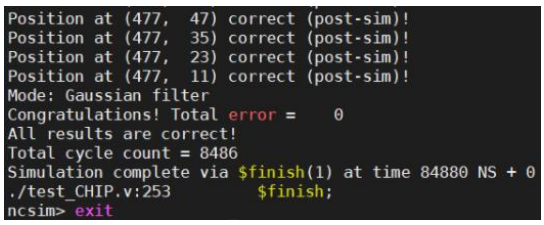


圖五、pre-sim 結果(上圖為 median filter，下圖為 Gaussian filter)

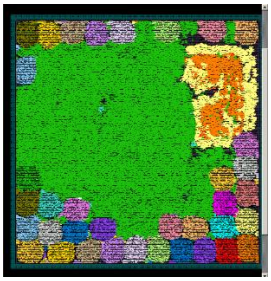


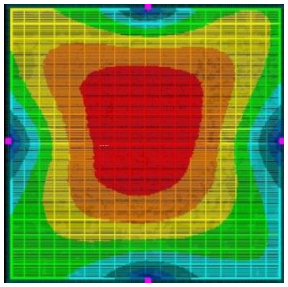
圖六、gate-sim 結果(上圖為 median filter，下圖為 Gaussian filter)



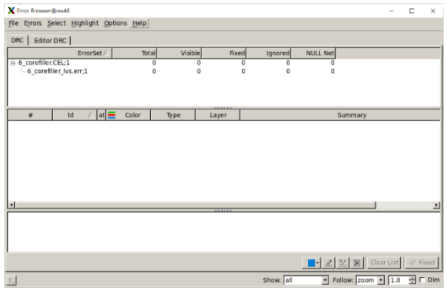


圖七、post-sim 結果(上圖為 median filter，下圖為 Gaussian filter)





圖八、routing 結果及 PNA voltage drop (左圖綠色部分為 HOG，佔電路最大部分面積)



圖九、DRC/LVS 錯誤已清空

	Absolute area (um^2)	Percentage (%)
Total cell area	303,623	100
HOG	159,038	52.38
sqrt	108,000	35.57
median filter	18,803	6.19
Gaussian filter	12,372	4.07
Others	~5,000	-

表三、各 module 佔的實際面積

Core utilization=0.7	Total cell area	Total core area
Absolute area	303,623	434,940

表四、面積及 core utilization

單位為 mW	ICC	Primetime (pre-sim)	Primetime (post-sim)
Total Power	28.3	5.375	14.9
		5.42	19.5
Cell Leakage Power	5.565	5.32	5.398
		5.32	5.564
Total Dynamic Power	22.7404	0.055	9.502
		0.1	13.936

表五、prime time 結果 (綠色為 median filter 橘色為 Gaussian filter)

H. Contribution of Each Member

●林暄傑

HOG (HOG.v), Gaussian filter (Gaussian.v), MUX.v, synthesis, APR

●劉亦傑

median filter (denoise.v, median.v), synthesis, APR

●謝霖泳

generate test patterns and golden file (Python), top.v , hog_counter.v, sqrt.v, testbench (test*, *.f), synthesis, APR

●報告為三人合力完成

I. Conclusion

最終順利處理一張 638*482 大小的圖片，進行 HOG 運算跟兩種主流 filter 的處理。效能在 area、power、cycle count 都比一開始設的目標還要好上不少，並在 core utilization 達到 70% 的水平。經過這次 Final Project 的洗禮，讓我們更體會到數位 IC 的設計流程及箇中奧秘。

J. Reference

[1] Navneet Dalal and Bill Triggs. Histograms of Oriented Gradients for Human Detection. *CVPR, 2005.*

[2] J. Yang et al. Image super-resolution as sparse representation of raw image patches. *CVPR, 2008.*

[3] Antoni Buades, Bartomeu Coll and J.M. Morel. A non-local algorithm for image denoising. *CVPR, 2005.*

[4] https://en.wikipedia.org/wiki/Median_filter

[5] https://en.wikipedia.org/wiki/Gaussian_filter