VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF SCIENCE, HO CHI MINH CITY
FACULTY OF ELECTRONICS & TELECOMMUNICATIONS
COMPUTER & EMBEDDED SYSTEM LABORATORY

Gesture Recognition Specification

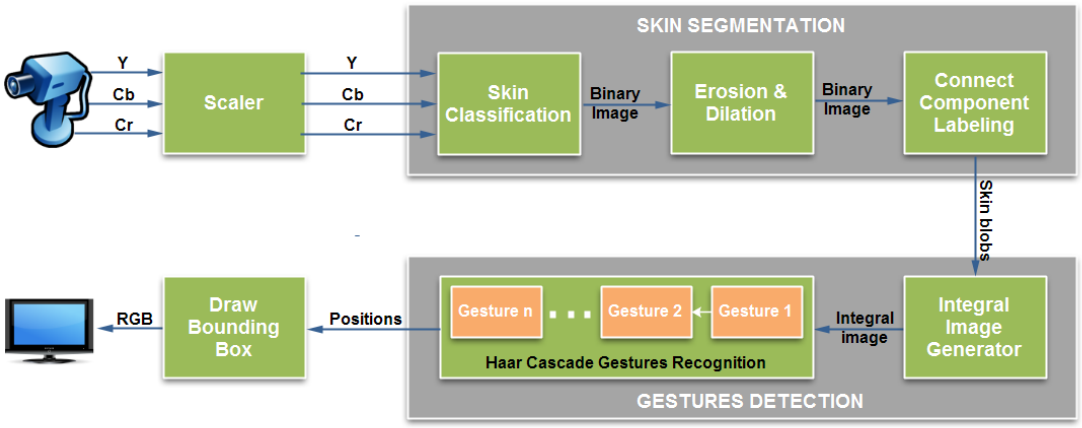
Summary:

Version: 1.0	**Release Date**: Month: May Year: 2012	**Total Size**: ☑ Reports ☑ Images ☑ References ☑ Source Codes ☑ Simulation
Solution Type:	☐ IP Core ☐ Megafunction ☐ SOPC Builder Component ☐ Qsys Component ☑ DSP Design	**External Devices**: ☑ Camera
Supported Device(s):	☑ TMS320DM642	- 720 MHz Clock Rate - 5760 MIPS - 2 Multiplier & 6 ALUs - 32 Mbytes SDRAM & 4 Mbytes Flash
Description:	Gesture Recognition System. C code on Code Compose Studio IDE	

1 TỔNG QUAN

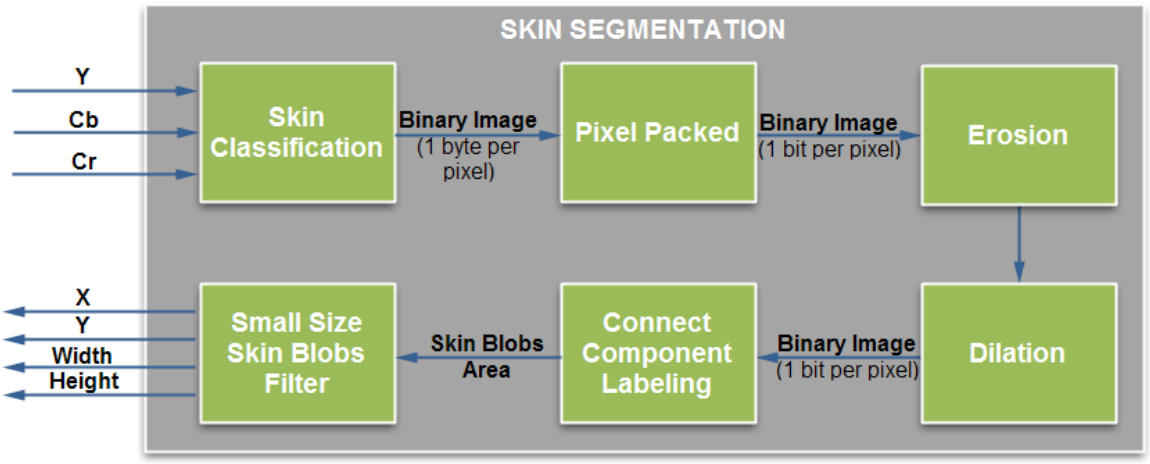
Mục tiêu của đề tài là xây dựng hệ thống nhận dạng cử chỉ trên nền tảng DSP. Hệ thống gồm 3 khối: Tách chọn màu da, Nhận dạng cử chỉ và Theo dõi cử chỉ (*chưa thực hiện*). Nhận dạng cử chỉ là khối quan trọng nhất, sử dụng thuật toán phân loại nhiều tầng (Cascade classifier) với các tập mẫu được huấn luyện bằng thuật toán Adaboost, cùng phương pháp trích xuất đặc tính Haar được giới thiệu bởi Viola & Jones.

Đề tài được thực hiên trên board DM642 Evaluation Module – sử dụng chip DSP TMS320DM642 của Texas Instrument, và sử dụng IDE Code Compose Studio v3.3.



2 TÁCH CHỌN MÀU DA – SKIN SEGMENTATION

Một trong những đặc điểm quan trọng để nhận biết con người đó là màu da. Do đó, việc sử dụng ảnh màu trong các hệ thống nhận dạng cơ thể người như phát hiện và theo dõi gương mặt, phát hiện và nhận dạng cử chỉ bàn tay dựa vào đặc điểm màu da có nhiều ưu điểm hơn việc chỉ sử dụng ảnh xám (grayscale). Hệ thống sử dụng ảnh màu để phát hiện vùng có màu da trên ảnh, sau đó thực hiện phương pháp phát hiện cử chỉ trên ảnh xám để giảm không gian tìm kiếm, tăng tốc độ.



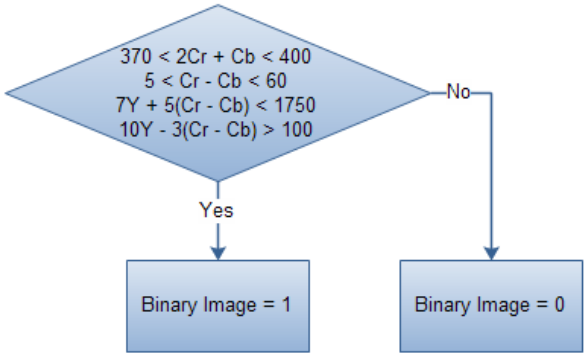
2.1 Image scale

Camera kết nối với board EVM642 qua cổng Composite có thể thu được hình ảnh ở độ phân giải 720 × 576 pixel. Tuy nhiên, việc lưu trữ và xữ lý hình ảnh ở độ phân giải này chiếm quá nhiều tài nguyên hệ thống và nó không cần thiết cho quá trình phát hiện cử chỉ. Do đó, hình ảnh sau khi được camera đưa vào sẽ được thu nhỏ lại ¼ lần xuống độ phân giải 160 × 120 pixel.

2.2 Skin pixel classification

Hệ thống thực hiện phân loại từng pixel có thuộc màu da hay không dựa vào giá trị của pixel đó. Output của module này là 1 hình ảnh binary với pixel có giá trị 1 thể hiện nó thuộc màu da, ngược lại pixel đó là 0.

Việc phân loại màu da có thể thực hiện ở nhiều mô hình màu khác nhau như RGB, YCbCr hay HSL. Tuy nhiên, sau khi thực hiện và đánh giá trên thực tế, kết quả ở từng mô hình không chênh lệch nhau nhiều. Vì thế, việc sử dụng mô hình YCbCr để phân loại là tiện lợi hơn cả do không cần chuyển đổi qua lại giữa các mô hình khác nhau.



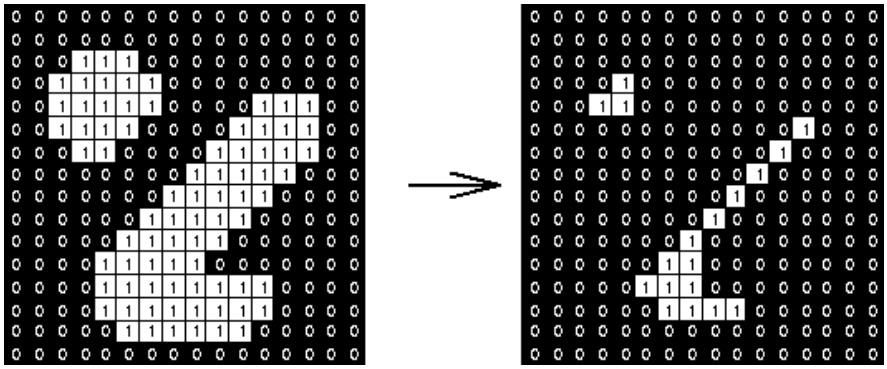
Tuy nhiên , độ chính xác của phương pháp này phụ thuộc nhiều vào camera, ánh sáng môi trường và không thể phân loại những đồ vật có màu giống màu da mặc dù đã thực hiện tinh chỉnh ngưỡng khá nhiều lần.

2.3 Erosion & Dilation

Erosion (ăn mòn) và dilation (giãn) là hai trong số các phép biến đổi hình thái học cở bản được dùng để xóa nhiễu, cách ly các phần tử riêng biệt, và liên kết các phần tử nằm rời nhau trong hình ảnh, đặc biệt là ảnh nhị phân.

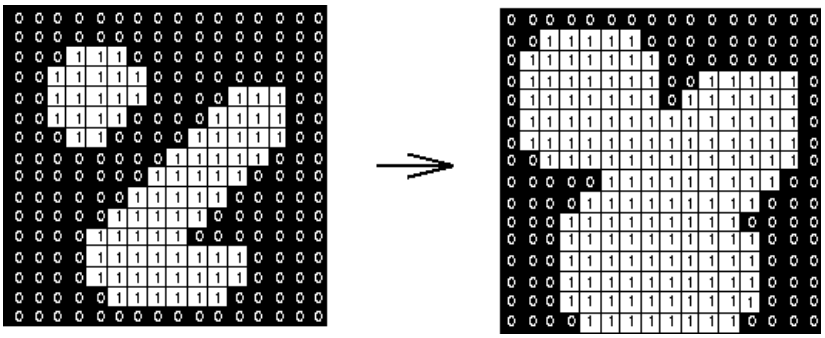
Erosion

Erosion dùng để co rút các binary object trong ảnh, thường dùng để khữ nhiễu trước khi thực hiện các bước xữ lý ảnh khác.Số lượng pixel bị xóa bỏ xung quanh object phụ thuộc vào structuring element.



Dilation

Dilation dùng mở rộng các binary object trong ảnh, thường được dùng để kết nối các object gần nhau lại trước khi thực hiện gán nhãn (labeling). Số lượng pixel gắn thêm vào xung quanh object phụ thuộc vào structuring element.



2.4 Connect Component Labeling

Sau khi phân loại để chọn ra các pixel có màu da và thực hiện các thao tác xữ lý nhiễu, việc cần thiết tiếp theo là trích ra các vùng có chứa màu da để tiến hành quá trình nhận dạng cử chỉ.

Thuật toán Connect Component Labeling tiến hành kiểm tra trên ảnh binary, nhóm các pixel với các pixel liền kề với nó, và gán nhãn cho các nhóm rời nhau.

Thuật toán như sau:

The specific algorithm is as follow:

(1) Initialization each array, the initial value is 0.

(2) Scan the first row of the image above all, if the current pixel image[1][j] is a skin pixel and m[1][j-1] ≠0, In other words image[1][j] is a skin pixel too. This time, m[1][j]=m[1][j-1],right(m[1][j])=j, size(m[1][j]) = size(m[1][j])+1.

(3)if m[1][j-1]=0, start a new label m[1][j] = k, top(m[1][j])=1 , bottom(m[1][j])=1 , left(m[1][j])=j, right(m[1][j])=j, size(m[1][j])= 1, k=k+1.

(4) Check the first line of each pixel in turn, carry on corresponding label with the array m[1][j],with arrays top[], bottom[], left[], right[] to count the four borders and array size[] to count the number of the current label pixels.

(5) From the second row, scan the pixels by row-major order. If the current pixel is skin, detect current point's neighbors according to the order top-right, top, top-left, left. That is to check m[1][j+1],m[1][j],m[1][j-1],m[2][j-1] whether be zero or not.

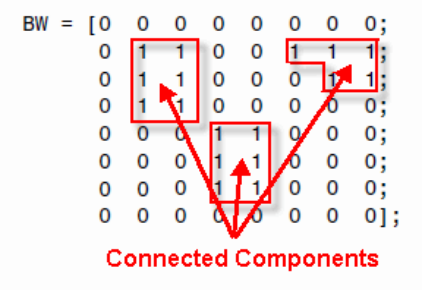
(6) If m[1][j+1],m[1][j],m[1][j-1],m[2][j-1] are all equal to zero, start a new label, carry on processing as steps (3).

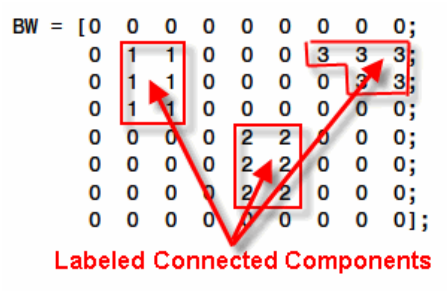
(7)If the top-right is a skin pixel m[1][j+1] ≠0, the current pixel's label is m[2][j]=m[1][j+1], left(m[2][j])=min(left(m[2][j]),j) , size(m[2][j])= size(m[2][j])+1, bottom(m[2][j])=i, 'i' is the number of current row.

(8) If the top-right pixel is not skin, then judge the top pixel.

(9)As the same principle, if the top-right and the top pixels are not skin, then judge the top-left pixel. if the top-left is not, judge the left pixel. the corresponding boundary arrays and the count array make the correspondingly adjust.

(10) After a row operations completely, make m [1][j]=m [2][j], m [2][j] 'data clear and prepare to operate the next row.





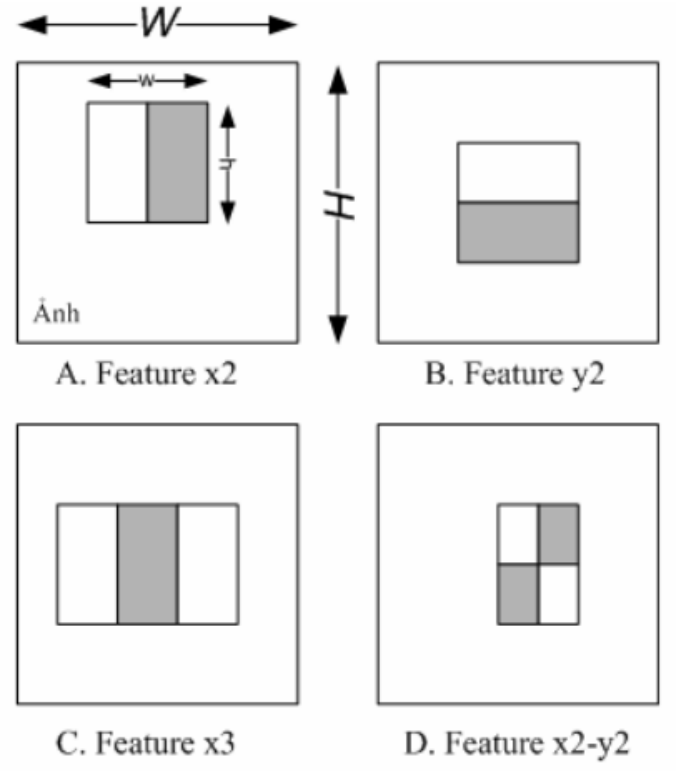
3 PHÁT HIỆN VÀ NHẬN DẠNG CỬ CHỈ

3.1 Tổng Quan

Trước khi đi vào thuật toán phát hiện và nhận dạng cử chỉ, việc hiểu một số khái niệm quan trọng cũng như những thành phần nhỏ hơn được sử dụng trong thuật toán là thật sự cần thiết. Vì thế phần này sẽ dành trình bày về những định nghĩa quan trọng cũng như những thành phần cốt lõi được sử dụng trong thuật toán phát hiện và nhận dạng cử chỉ.

3.1.1 Haar Feature

Haar Feature là một loại đặc trưng thường được dùng cho bài toán nhận dạng trên ảnh. Haar Feature được xây dựng từ các hình chữ nhật có kích thước bằng nhau, dùng để tính độ chênh lệch giữa các giá trị điểm ảnh trong các vùng kề nhau. Trong hình a và b bên dưới, giá trị của feature cho bởi một ảnh bằng hiểu số giữa tổng các điểm ảnh thuộc hai vùng hình chữ nhật sáng và tối. Trong hình c thì giá trị feature bằng tổng các điểm ảnh trong hai vùng chữ nhật bên ngoài trừ tổng các điểm anh trong hình chữ nhật ở giữa. Trong hình d, giá trị feature bằng tổng các điểm ảnh nằm trong hai hình chữ nhật màu tối trừ tổng các điểm ảnh nằm trong hai hình chữ nhật màu sáng.



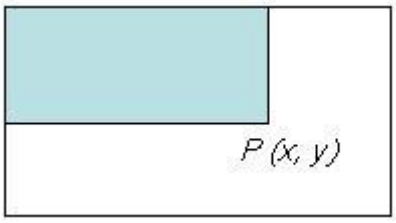
Tác dụng của Haar Feature là nó thể hiện được thông tin về các đối tượng trong ảnh (vì nó biểu diễn mối liên hệ giữa các bộ phận của đối tượng), điều mà bản thân từng điểm ảnh không thể hiện được.

Như vậy có thể thấy rằng, để tính các giá trị Haar Feature, ta phải tính tổng các điểm ảnh nằm trong các vùng trên ảnh. Nhưng để tính toán các giá trị của các Haar Feature cho tất cả các vị trí trên ảnh đòi hỏi chi phí tính toán khá lớn, không đáp ứng được yêu cầu thời gian thực. Do đó Viola và Jones đưa ra một khái niệm gọi là Integral Image.

3.1.2 Integral Image

Integral Image là một mảng hai chiều với kích thước bằng với kích thước của ảnh cần tính các Haar Feature, với mỗi phần tử của mảng này được tính bằng cách tính tổng của điểm ảnh phía trên và bên trái của nó. Bắt đầu từ vị trí phía trên - bên trái đến vị

trí phía dưới - bên phải của ảnh, việc tính toán này đơn thuần chỉ đựa trên phép cộng số nguyên đơn giản, do đó tốc độ thực hiện rất nhanh.



$$P(x,y) = \sum_{x' \le x, y' \le y} i(x', y')$$

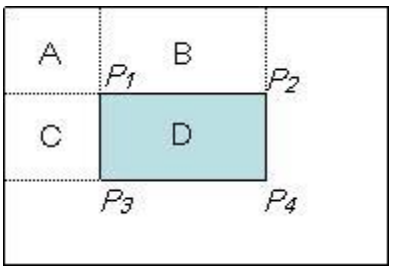
Sau khi đã tính được Integral Image, việc tính tổng các giá trị điểm ảnh của một vùng bất kỳ nào đó trên ảnh thực hiện rất đơn giản theo cách sau:

- Giả sử ta cần tính tổng các giá trị mức xám của vùng D như trong hình dưới, ta có thể tính như sau:

  D = A + B + C + D – (A + B) – (A + C) + A

- Với A + B + C + D chính là giá trị tại điểm P4 trên Integral Image, tương tự như vậy A + B là giá trị tại điểm P2, A + C là giá trị tại điểm P3, và A là giá trị tại điểm P1. Vậy ta có thể viết lại biểu thức tính D ở trên như sau:

$$D = \underbrace{(x4,y4)}_{A+B+C+D} - \underbrace{(x2,y2)}_{(A+B)} - \underbrace{(x3,y3)}_{(A+C)} + \underbrace{(x1,y1)}_{A}$$

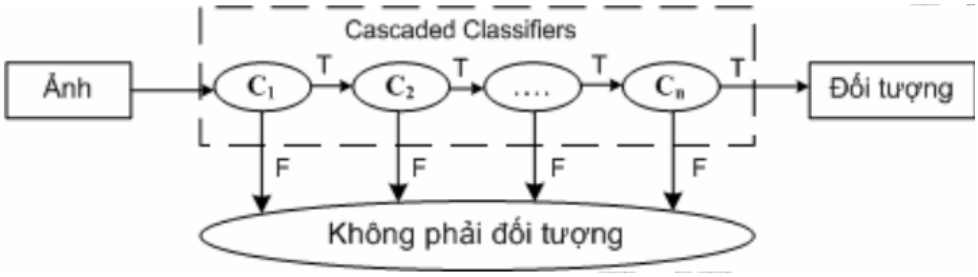


3.1.3 Cascade Of Classifiers

Các bộ phân loại tốt thường tốn rất nhiều thời gian để cho ra kết quả phân loại bởi vì nó phải xét rất nhiều đặc trưng của mẫu. Tuy nhiên, trong các mẫu đưa vào, không phải mẫu nào cũng thuộc loại khó nhận dạng, có những mẫu rất dễ nhận ra. Đối với những mẫu này, ta chỉ cần xét một hay vài đặc trưng đơn giản là có thể nhận diện mà không cần xết tất cả các đặc trưng. Nhưng đối với các bộ phân loại thông thường thì cho dù mẫu cần nhận dạng là dễ hay khó thì nó vẫn sẽ xét tất cả các đặc trưng mà nó

rút ra được trong quá trình huấn luyện. Việc này làm tốn thời gian xử lý một cách không cần thiết.

Cascade of Classifiers được xây dựng để rút ngắn thời gian xử lý. Cascade Tree gồm nhiều stage, mỗi stage của cây sẽ là một stage classifier. Một mẫu để được phân loại là đối tượng thì nó cần phải qua hết tất cả các stage của cây. Các stage classifiers ở stage sau được huấn luyện bằng những mẫu negative mà stage classifier trước nó nhận dạng sai, tức là nó sẽ tập trung học từ các mẫu background khó hơn, do đó sự kết hợp các stage classifier này lại sẽ giúp bộ phân loại có tỉ lệ phân loại sai thấp. Với cấu trúc này, những mẫu background dễ nhận diện sẽ bị loại ngay từ stage đầu tiên, giúp đáp ứng tốt nhất với sự gia tăng độ phức tạp của các mẫu đưa vào, đồng thời rút ngắn thời gian xử lý.



3.1.4 Nhận Dạng Cử Chỉ Với Cascade Of Boosted Classifiers

Từ Integral Image, ta có thể tính nhanh tổng các điểm ảnh nằm trong một vùng hình chữ nhật bất kỳ trên ảnh. Các vùng ảnh này sẽ được đưa qua các hàm Haar Feature cơ bản để tính toán đặc trưng, kết quả tính toán sẽ được đưa qua bộ điều chỉnh AdaBoost để loại bỏ nhanh các đặc trưng không có khả năng là đặc trưng của cử chỉ bàn tay. Chỉ có một tập nhỏ các đặc trưng mà bộ điều chỉnh AdaBoost cho là có khả năng là đặc trưng của cử chỉ bàn tay mới được chuyển sang cho bộ quyết định kết quả (là tập các bộ phân loại yếu có cấu trúc như trong trên). Bộ quyết định sẽ tổng hợp kết quả là cử chỉ bàn tay nếu kết quả của các bộ phân loại yếu trả về là cử chỉ bàn tay.

Mỗi bộ phân loại yếu sẽ quyết định kết quả cho một Haar Feature, được xác định ngưỡng đủ nhỏ sao cho có thể vượt được tất cả các bộ dữ liệu mẫu trong tập dữ liệu huấn luyện (số lượng ảnh trong tập huấn luyện có thể rất lớn). Trong quá trình xác định cử chỉ bàn tay, mỗi vùng ảnh con sẽ được kiểm tra với các đặc trưng trong chuỗi các Haar Feature, nếu có một Haar Feature nào cho ra kết quả là cử chỉ bàn tay thì các đặc trưng khác không cần xét nữa. Thứ tự xét các đặc trưng trong chuỗi các Haar Feature sẽ được dựa vào trọng số (weight) của đặc trưng đó do AdaBoost quyết định dựa vào số lần và thứ tự xuất hiện của các đặc trưng Haar-like.

3.2 Chi Tiết Khối Nhận Dạng Cử Chỉ

3.2.1 Calculate Integral Image

Theo phương pháp do Viola & Jones đề xuất, để tính được Integral Image $I_{m,n}$, ta cần phải tính $R_{m,n}$ ($R_{m,n}$ là tổng các điểm ảnh từ hàng đầu tiên đến hàng thứ m) trước, sau đó tính $I_{m,n}$ theo hai biểu thức đệ quy sau:

$$\begin{aligned} R_{m,n} &= R_{m,n-1} + A_{m,n} \\ I_{m,n} &= I_{m-1,n} + R_{m,n} \end{aligned}$$

Khi thực hiện phương pháp này trên board DSP, cách đơn giản nhất là lưu trữ tất cả dữ liệu cần tính toán trên bộ nhớ ngoài – SDRAM, vì bộ nhớ ngoài thường có dung lượng lớn và chi phí thấp. Tuy nhiên, việc truy xuất đến bộ nhớ ngoài chậm hơn rất nhiều so với truy xuất bộ nhớ on-chip.

Việc tính Integral Image của dữ liệu chứa ở bộ nhớ ngoài thường được thực hiện theo các bước sau:

- *(a)* Chuyển một phần dữ liệu ảnh vào bộ nhớ on-chip.
- *(b)* Dữ liệu được lấy ra từ on-chip để thực hiện tính toán, sau đó lưu kết quả vào on-chip.
- *(c)* Chuyển kết quả từ bộ nhớ on-chip vào biến output chứa trên bộ nhớ ngoài.

Các bước được lặp lại tuần tự theo *(a)* – *(b)* – *(c)* cho đến khi đã xữ lý hết dữ liệu.

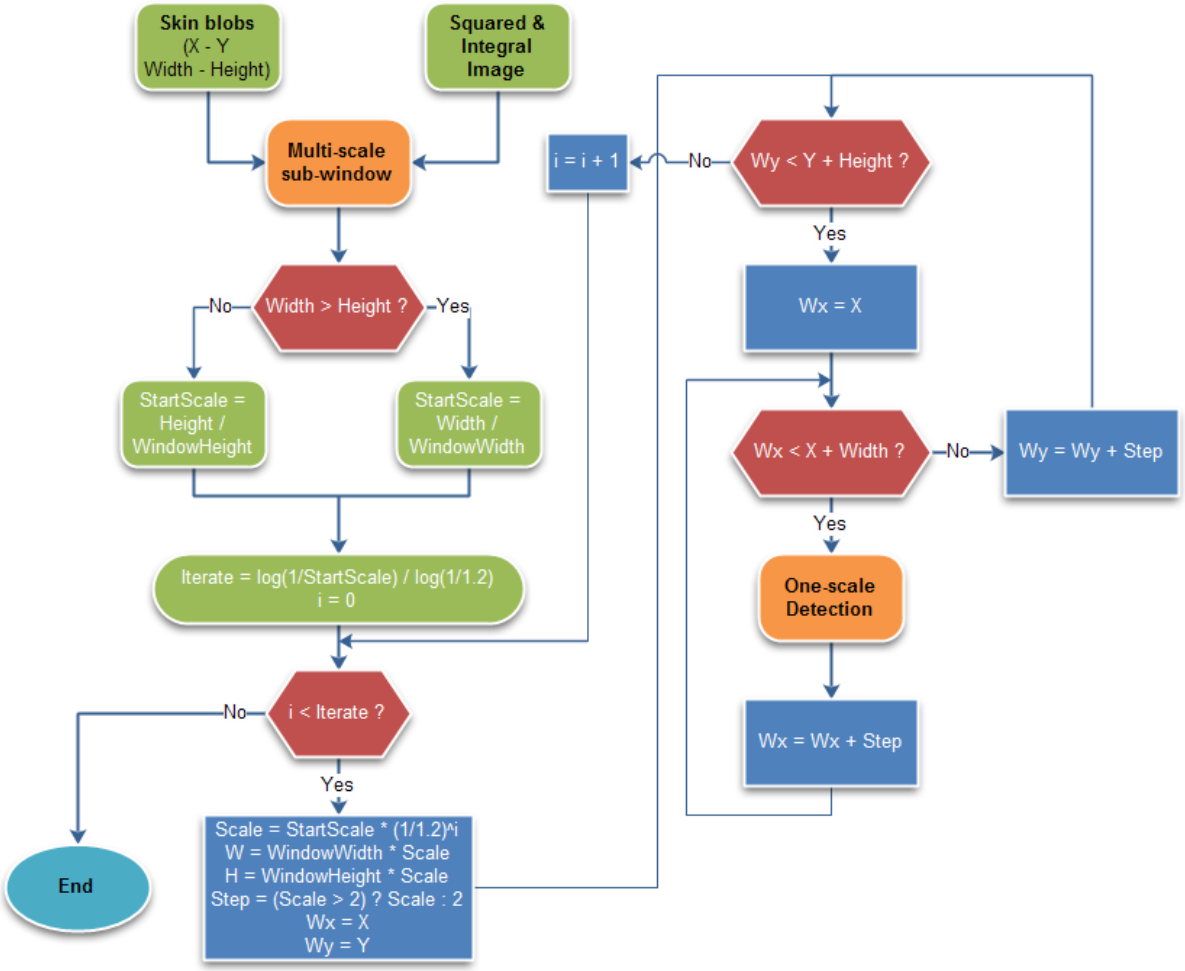
$$a_1 \; b_1 \; c_1 \; a_2 \; b_2 \; c_2 \; a_3 \; b_3 \; c_3 \; \ldots \; a_r \; b_r \; c_r$$

Tuy nhiên, có thể thấy rằng không phải tất cả các bước đều phụ thuộc vào bước xữ lý trước đó. Ví dụ, nếu sử dụng 2 buffer, *(a_2)* có thể bắt đầu chuyển dữ liệu mới vào buffer thứ 2 ngay khi *(a_1)* kết thúc, tức thực hiện song song với *(b_1)*. Trình tự các bước giống như sau:

$$\begin{array}{lccccccccc} \text{CPU} & & b_1 & & b_2 & & \ldots & b_{r-1} & & b_r \\ \text{DMA} & a_1 & a_2 & c_1 & a_3 & c_2 & \ldots & a_r & c_{r-1} & & c_r \end{array}$$

Bằng cách này, thời gian xữ lý khi thực hiện song song *(a)* và *(b)* bằng với thời gian thực hiện của *(a)* hoặc *(b)*.

3.2.2 Multi-scale Sub-window



3.2.3 One-scale Detection

Flowchart comming soon

Source Code Down-scale Image

```
/*Create buffer for image scale*/
unsigned char y_buf[100800]; // 100800 = ((640 * 480) * 21) / 64
unsigned char cr_buf[50400]; // 50400 = 100800 / 2
unsigned char cb_buf[50400]; // 50400 = 100800 / 2

/*Excution image scale*/
VLIB_imagePyramid8((unsigned char*)capFrameBuf->frame.iFrm.y1,
numPixels, numLines, y_buf);
VLIB_imagePyramid8((unsigned char*)capFrameBuf->frame.iFrm.cb1,
(numPixels >> 1), numLines, cb_buf);
VLIB_imagePyramid8((unsigned char*)capFrameBuf->frame.iFrm.cr1,
(numPixels >> 1), numLines, cr_buf);

/*Create pointer to image scale 160 x 120*/
unsigned char *y_scaled = y_buf + 76800; // 76800 is offset of 320 * 240
unsigned char *cr_scaled = cb_buf + 38400; // 38400 = 76800 / 2
unsigned char *cb_scaled = cr_buf + 38400; // 38400 = 76800 / 2
```

Source Code Skin Classification

```
void skin_ycbcr
(
    const unsigned char     *y_data,    /* Luminence data(Y')*/
    const unsigned char     *cb_data,   /* Blue color-difference */
    const unsigned char     *cr_data,   /* Red color-difference */
    unsigned char *restrict bin_data,   /* Binary pixel output */
    unsigned                num_pixels  /* #of luma pixels to process */
)
{
    unsigned short     i;  /* Loop counter */
    unsigned char     y0;  /* Individual Y components*/
    unsigned char     cb, cr;/* Color difference components*/

/*  Iterate for num_pixels/2 iters, since we process pixels in pairs*/
    i = num_pixels >> 1;

    while (i-->0)
    {
        y0 = *y_data++;
        cb = *cb_data++;
        cr = *cr_data++;
        if((2*cr + cb) > 378 && (2*cr + cb) < 403
            && (cr - cb) > 1 && (cr - cb) < 20
            && (7*y0 + 5*(cr - cb)) < 1400
            && (10*y0 - 3*(cr - cb)) > 850
            )
        {
            *bin_data++ = 1;
            *bin_data++ = 1;
            *y_data++;
        }
        else
        {
            *bin_data++ = 0;
            *bin_data++ = 0;
            *y_data++;
        }
    }

    return;
}
```

Source Code Erosion & Dilation

```
/* Packed binary image 1 pixel per byte to 1 pixel per bit */
VLIB_packMask32(binary_img, binary_img_packed, 19200);

/* Excute Erosion and Dilation */
VLIB_erode_bin_cross((const unsigned char *)binary_img_packed, (unsigned
char *)erode_dilate, 19200, 160);
VLIB_dilate_bin_square((const unsigned char *)erode_dilate, (unsigned
char *)binary_img_packed, 19200, 160);
VLIB_dilate_bin_square((const unsigned char *)binary_img_packed,
(unsigned char *)erode_dilate, 19200, 160);
```

Source Code Connect Component Labeling

```
// This function returns the bytes required to process the worst-case,
pathological
// binary image, which is nearly impossible to occur in practice. This
estimate
// considers image dimensions as well the number and configuration of
binary pixels
VLIB_calcConnectedComponentsMaxBufferSize(IMAGEWIDTH,IMAGEHEIGHT,MINBLOB
AREA,&maxBytesRequired);
bytesRecommended = maxBytesRequired;

sizeOfCCHandle =  VLIB_GetSizeOfCCHandle();
handle = (VLIB_CCHandle *) MEM_alloc(DDR2HEAP, sizeOfCCHandle, 8);

// Set-up Memory Buffers
pBuf    = (void *)  MEM_alloc(DDR2HEAP, bytesRecommended, 8);
status  = VLIB_initConnectedComponentsList(handle, pBuf,
bytesRecommended);
status = VLIB_createConnectedComponentsList(handle,
    IMAGEWIDTH,
    IMAGEHEIGHT,
    (int*)erode_dilate,
    MINBLOBAREA,
    1);
VLIB_getNumCCs(handle, &numCCs);

for (i=0; i < numCCs; i++)
{
    VLIB_getCCFeatures(handle, &vlibBlob, i);

    CCFeatures[i][0]   = vlibBlob.xmin;
    CCFeatures[i][1]   = vlibBlob.xmax;
    CCFeatures[i][2]   = vlibBlob.ymin;
    CCFeatures[i][3]   = vlibBlob.ymax;

}

MEM_free(DDR2HEAP, pBuf, maxBytesRequired);
MEM_free(DDR2HEAP, handle, sizeOfCCHandle);
```

Source Code Calculate Integral Image

```
/*******************************************************/
/* Calculate Integral Image                            */
/*******************************************************/
for(i = 0; i < numLines; i++)
    for(j = 0; j < numPixels; j++)
        *(squared_img + i * numPixels + j) = (*((unsigned
char*)capFrameBuf->frame.iFrm.y1 + i * numPixels + j))*(*((unsigned
char*)capFrameBuf->frame.iFrm.y1 + i * numPixels + j));
memset(pLastLine, 0, 352 * sizeof(unsigned int*));
VLIB_integralImage8((unsigned char*)capFrameBuf->frame.iFrm.y1,
image_width, image_height, pLastLine, ii_m);
memset(pLastLine, 0, 352 * sizeof(unsigned int*));
VLIB_integralImage16(squared_img, image_width, image_height, pLastLine,
sii_m);
```

Source Code Multi-scale Sub-window Scaning

```
int Multi_Scale_Subwindow(double *ii_m, double *sii_m)
{
    int count = 0;
    double ScaleWidth = ((double)image_width)/window_w;
    double ScaleHeight = ((double)image_height)/window_h;
    double StartScale = 0;
    if(ScaleHeight < ScaleWidth)
        StartScale = ScaleHeight;
    else
        StartScale = ScaleWidth;
    unsigned char itt = (unsigned char)ceil(log(1/StartScale) /
log(1/1.2));
    for(int i = 0; i < itt; i++)
    {
        double Scale = StartScale * pow(1/1.2, i);
        printf("Scale: %f \n", Scale);

        //Scale window size
        int w = floor(window_w * Scale);
        int h = floor(window_h * Scale);

        // Spacing between search coordinate of the image
        int step = (Scale > 2) ? floor(Scale) : 2;
        int wy;
        int wx;
        // Find hands in the image for the current scale
        for(wy = 1; wy < image_height - h; wy += step)
        {
            for(wx = 1; wx < image_width - w; wx += step)
            {
                if(HaarCascade_Detection(ii_m, sii_m, wx, wy, w, h,
Scale))
                {
                    Hands[count][0] = wx;
                    Hands[count][1] = wy;
                    Hands[count][2] = w;
                    Hands[count][3] = h;
                    count++;
                }
            }
        }
    }
    return count;
}
```

Source Code One-scale Detection

```
int Onescale_Detection( double *ii_m,
                        double *sii_m,
                        int wx, int wy,
                        int w, int h,
                        double Scale
                        )
{
    int curr_tree_index = 0;
    int curr_threshold_index = 0;
    int curr_feature_index = 0;
    double sum_stage = 0;
    double mean = GetSumRect(ii_m, wx, wy, w, h) / (w * h);
    double variance = GetSumRect(sii_m, wx, wy, w, h) / (w * h) -
pow(mean,2);
    variance = (variance < 0) ? 1 : variance;
    double standard_deviation = sqrt(variance);
    // Loop through a classifier stage
    for(int i_stage = 0; i_stage < (sizeof(stage)/sizeof(stage[0]));
i_stage++)
    {
        sum_stage = 0;
        for(int i_trees = 0; i_trees < stage[i_stage]; i_trees++)
        {
            double Rectangle_sum = 0;
            int i_feature;
            for(i_feature = 0; i_feature < 3; i_feature++)
            {
                int RectX =
(int)floor(feature[1][curr_feature_index]*Scale + wx);
                int RectY =
(int)floor(feature[2][curr_feature_index]*Scale + wy);
                int RectW =
(int)floor(feature[3][curr_feature_index]*Scale);
                int RectH =
(int)floor(feature[4][curr_feature_index]*Scale);
                int RectWeight = feature[0][curr_feature_index];
                Rectangle_sum += (GetSumRect(ii_m, RectX, RectY, RectW,
RectH) * RectWeight);
                curr_feature_index++;
            }
            Rectangle_sum = Rectangle_sum / (w * h);

            // Get the values of the current haar-classifiers
            if(Rectangle_sum < threshold[curr_threshold_index] *
standard_deviation)
                sum_stage += left_val[curr_threshold_index];
            else
                sum_stage += right_val[curr_threshold_index];
            curr_threshold_index++;
        }
        if(sum_stage > stage_threshold[i_stage])
            continue;
        else
            return 0;
    }
    return 1;
}
```