High-level synthesis and optimization of video signal processing IP

Tadej Skuber, Andrej Trost, Andrej Žemva

Faculty of Electrical Engineering, University of Ljubljana, Slovenia
E-mail: ts7305@student.uni-lj.si

Abstract. *Experiment described in the paper studies design methodology by implementing image rotation with bilinear interpolation. Algorithm is firstly coded and tested in high level language (C). Code is than converted to RTL design using special commands (directives). To reach desired result latency and small circuit size, code style and directives are experimented with. Effects of different fixed-point decimal places is also analyzed.*

1 Introduction

Advanced designs used in today's increasingly complex electronic products are stretching the boundaries of density and performance. They create a challenge for design teams to hit a target release window within their allocated budget. A productive methodology for addressing these design challenges is one where more time is spent at higher levels of abstraction, where verification time is faster and productivity gain is the greatest [1].

The main goal of this paper is to use High Level Synthesis (HLS) to find algorithm with optimal performance, from the choice of multiple versions of known algorithm for image rotation, evaluating performance and resource use. The usage of HLS enables us a fast way of checking both criteria without much coding effort.

Image rotation and translation are widely used operations in image processing with possible parallelism potential. The only problem is a need for additional resampling algorithm.

In computer vision and image processing, bilinear interpolation is one of the basic resampling techniques. It can be used where perfect image transformation with pixel matching is impossible, so that one can calculate and assign appropriate intensity values to pixels. Unlike other interpolation techniques, such as nearest-neighbor interpolation and bicubic interpolation, bilinear interpolation uses values of only the 4 nearest pixels, located in diagonal directions from a given pixel, in order to find the appropriate color intensity values of chosen pixel [2].

2 Rotation and bilinear interpolation

Used algorithm iterates over resulting image in x' and y' direction. For every pixel (x', y') it looks where does it mirror in original image (x, y) (Fig. 1). Algorithm uses the following formula:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \left(\begin{bmatrix} x' \\ y' \end{bmatrix} - T \right) A + \begin{bmatrix} II_H/2 \\ II_W/2 \end{bmatrix} \quad (1)$$

where

$$A = \begin{bmatrix} a_{00} & a_{10} \\ a_{01} & a_{11} \end{bmatrix} = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix} \quad (2)$$

$$T = \begin{bmatrix} t_1 \\ t_2 \end{bmatrix} = \begin{bmatrix} OI_H/2 - t_{1d} \\ OI_W/2 - t_{2d} \end{bmatrix} \quad (3)$$

Matrix T translates original image for t_{1d} pixels in x direction and for t_{2d} pixels in y direction. Rotation is handled by matrix A, which rotates image for for angle θ in anti-clock wise direction. II_H and II_W are constants that represent original image (Input Image or II) width and height. Constants OI_H and OI_W represent width and height of Output Image (OI).

Getting values from source

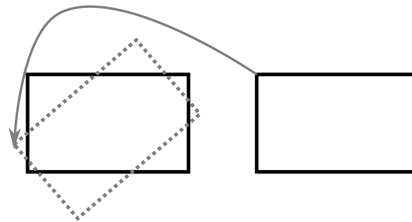

Figure 1: Representation of algorithm operation [3].

Image transformation does not result in integer coordinates values of the pixels. Therefore bilinear interpolation is used to find appropriate pixel values at given coordinates. Pixel color value $f(x, y)$ is calculated using the four neighbor pixels color values (Fig. 2) by the following formula:

$$f(x, y) = \begin{bmatrix} 1-x & x \end{bmatrix} \begin{bmatrix} f(0,0) & f(0,1) \\ f(1,0) & f(1,1) \end{bmatrix} \begin{bmatrix} 1-y \\ y \end{bmatrix} \quad (4)$$

Values $f(0,0), f(1,0), f(0,1), f(1,1)$ are neighbor pixel color values. Variables x and y are distances of calculated (x, y) from the nearest integer coordinate value.

Formula is applied to each of three RGB (Red, Green and Blue) pixel colors, giving three $f(x, y)$ result values that are combined in one piece of data.

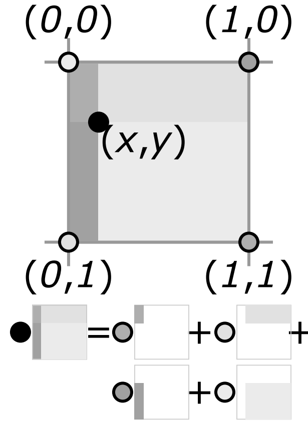

Figure 2: Geometric visualization of bilinear interpolation [2].

2.1 C code

The following code (Lst. 1) was firstly used to test the algorithm and asses performance of HLS (High Level Synthesis).

```
typedef int             cnt_t;
typedef ap_fixed<32, 16> fix_t;
typedef ap_uint<24>      col_t;
...
void bilinear(
col_t input_image[II_H*II_W],
col_t output_image[OI_H*OI_W],
fix_t A[4],
fix_t T[2]){
...
loopi: for (cnt_t i = 0; i < OI_H; i++){
loopj: for (cnt_t j = 0; j < OI_W; j++){

x = ((fix_t)(i) - OI_H_2 - T[1])*(A[0]) +
((fix_t)(j) - OI_W_2 - T[0])*(A[1]) + II_H_2;
y = ((fix_t)(i) - OI_H_2 - T[1])*(A[2]) +
((fix_t)(j) - OI_W_2 - T[0])*(A[3]) + II_W_2;
...
if ((x0 >= 0 && y0 >= 0) &&
(x1 < II_H && y1 < II_W)){
dx = x - (fix_t)(x0);
dy = y - (fix_t)(y0);
dx_dy = dx*dy;

loopk: for (cnt_t k = 0; k < 3; k++){
shift = k << 3;

output_image[i*OI_W + j] |= ((col_t)(
(fix_t)(((input_image[x0 * II_W + y0]) &
(col_t)(0xFF << (shift))) >> (shift))*
(1+dx_dy-dx-dy)+
(fix_t)(((input_image[x1 * II_W + y0]) &
(col_t)(0xFF << (shift))) >> (shift))*(dx-dx_dy) +
(fix_t)(((input_image[x0 * II_W + y1]) &
(col_t)(0xFF << (shift))) >> (shift))*(dy-dx_dy) +
(fix_t)(((input_image[x1 * II_W + y1]) &
(col_t)(0xFF << (shift))) >> (shift))*dx_dy))<<
(shift);
...
```

Listing 1: Basic unmodified C code.

Main algorithm input is 1D array input_image of size II_H*II_W, with each 24 bit element. This array contains 2D image that is transformed into single row. Other inputs are matrix A and T, containing information of image rotation and translation. Output is rotated and translated image output_image with size OI_H*OI_W, each element also being 24 bits in size.

The RGB color values are coded in 24 bit ap_uint type variables with identifier type col_t, giving each color a standard 8 bit value. Variables with identifier fix_t are of type ap_fixed and are used for calculations where decimal places are needed.

Basic algorithm implementation loops through resulting image with two for loops, each pixel assigned (with coordinates (i, j)) a proper pixel value form original image (with coordinates (x, y)). Pixel color calculation is done with bilinear interpolation according to equation (4).

3 Directives

Using Vivado HLS compiler directives, we can assist it in mapping C/C++ code to hardware [6]. In design four directives were used, unroll, pipeline, interface and array reshape. Unroll was used in for loop ”loopk”, for calculation of the individual RGB color value. Loops ”loopi” and ”loopj” are too large for unrolling, so directive was not used here. Directive pipeline was used on all loops, reducing the initiation interval for loops by allowing the concurrent execution of operations [5]. Directive interface with argument ap_memory was used on function array parameters, for implementation of arguments as standard RAM interface. Images were then conveniently stored in RAM. Array reshape was used for parallel access to data in RAM.

Algorithm synthesis settings were set for ZedBoard using Vivado HLS 2017.4. Synthesis clock period was set to default 10 ns, and implementation was done in C and supports various input (II_H being height and II_W width) and output (OI_H and OI_W) image sizes. For implementation, standard test image size of $512*512$ was used.

Results of using different directives are visible in the following table:

DIRECTIVE	LATENCY/INTERVAL
/	8127489
only pipeline and unroll	524299
all directives	524299

Table 1: Latency when using different directives.

It can be seen(Tab. 1) that it is possible to achieve latency of about two clock cycles per pixel ($512*512*2 = 524288$), but it can also be noticed that additional directives (interface memory and array reshape) do not help to decrease latency. Problem is that only one RAM array element can be accessed in one clock cycle. Solution would be in loading multiple pixel data in one read cycle, and since input data repeats when iterating true image pixels. This can be used to lower latency to one clock cycle per pixel (Lst. 2).

4 Code modification

Code modification was done so that it does consider repetition of data input [11]. That was done with 48 bit data input instead of 24. Image data was so divided in one array of size II_W/2*II_H. Each element of array therefore includes 2 pixel data. This small change in code gives us desired clock cycles per pixel. For image size $512 * 512$ giving us latency of 262155 ($262155/(512 * 512) \approx 1$).

But this method is only applicable for images of size $2^n * 2^m$. For all other sizes latency is the same. Algorithm with different image size inputs has latency of at least $II_H * II_W * 2$. The test with image size $650 * 975$ has latency of 2535038 ($2535038/(650*975) \approx 4$), or image of size $640 * 540$ has 691235 ($2535038/(640*540) \approx 2$) clock cycles of latency. This modification works only when directive array reshape is applied.

	basic	modified
DSP48E	64/29%	64/29%
FF	2620/2%	3135/2%
LUT	7690/14%	7295/13%

Table 2: Resources used by different algorithms. Both using 32 bit fixed point numbers with 16 bits for calculations.

```
1  ...
2  if((y0 & 0x1) == (cnt_t)(0)){
3  temp_image[0] =
4  (color_t)(input_image[x0 * II_W/2 + (y0 >> 1)]);
5  temp_image[1] =
6  (color_t)(input_image[x0 * II_W/2 + (y0 >> 1)] >>
       24);
7  temp_image[2] =
8  (color_t)(input_image[x1 * II_W/2 + (y0 >> 1)]);
9  temp_image[3] =
10 (color_t)(input_image[x1 * II_W/2 + (y0 >> 1)] >>
       24);
11 }else{
12 temp_image[0] =
13 (color_t)(input_image[x0 * II_W/2 + (y0 >> 1)] >>
       24);
14 temp_image[1] =
15 (color_t)(input_image[x0 * II_W/2 + (y1 >> 1)]);
16 temp_image[2] =
17 (color_t)(input_image[x1 * II_W/2 + (y0 >> 1)] >>
       24);
18 temp_image[3] =
19 (color_t)(input_image[x1 * II_W/2 + (y1 >> 1)]);
20 }
21 ...
22 loopk: for (cnt_t k = 0; k < 3; k++) {
23 shift = k << 3;
24 temp_out[k] = ((color_t)(
25 (fixed_t)(((temp_image[0]) & (color_t)(0xFF << (
       shift)))
26 >> (shift))*(1+dx_dy-dx-dy) +
27 (fixed_t)(((temp_image[2]) & (color_t)(0xFF << (
       shift)))
28 >> (shift))*(dx-dx_dy) +
29 (fixed_t)(((temp_image[1]) & (color_t)(0xFF << (
       shift)))
30 >> (shift))*(dy-dx_dy) +
31 (fixed_t)(((temp_image[3]) & (color_t)(0xFF << (
       shift)))
32 >> (shift))*dx_dy))<<(shift);
33 ...
```

Listing 2: Modified C code.

Use of resources is visible in table 2. It can be seen that there are no major differences in use of resources between basic and modified algorithm for image of size $512 * 512$.

5 Effects of fixed point

Effects of using fixed point, instead of floating point, were also studied. Different number of fixed point decimal bits were used to see the effects on image quality. For assessment of image quality fixed point images were compared to floating point image. Code was only slightly changed (Lst. 3).

Figure 3: Fixed point images with 1, 2, 3 and 16 decimal bits of accuracy.

$$MSE = \frac{1}{m * n} \sum_{i=0}^{n} \sum_{j=0}^{m} [I(i,j) - K(i,j)]^2 \quad (5)$$

Results of comparison are given in Table 4. Image comparison was done with two algorithms, Mean Squared Error (MSE) and Structural Similarity Measure (SSIM) [6]. MSE was simply implemented according equation 5. More complicated image assessment with SSIM [7] was done using existing python library [9]. For simplifying image quality estimation, gray-scale image was used.

```
1 typedef int                cnt_t;
2 //typedef ap_fixed<32, 16> fix_t;
3 typedef float              fix_t;
4 typedef ap_uint<24>        col_t;
5 ...
```

Listing 3: Changing fixed point to floating point to compare images.

NDB	DSP48E	FF	LUT
8	17	2423	2423
9	17	2467	5982
10	17	2696	6452
11	17	2724	6826
12	17	2778	7014
13	17	2836	7188
14	23	2894	7324
15	43	2820	7351
16	51	3092	7334
17	51	2916	7332
18	51	2964	7311

Table 3: Resources used when changing number of fixed point bits. Using 10 bits for integer part and NDB for decimal part.

As can be seen in Fig. 3 and Tab. 4, image quality quickly becomes sufficient with the use of more than 8

decimal bits. Image distortion is visible only if a few decimal bits are being used. Assessment methods used enable insight of image quality, revealing that use of more bits does not significant add to image quality. But increasing the number of fixed point bits has an effect on resource use (Tab. 3), mainly on DSP48E blocks (FFs and LUTs are not greatly effected).

NDB	MSE	SSIM
1	4782	0.32853
2	4556	0.32055
3	2202	0.47513
4	951	0.60306
5	590	0.68887
6	230	0.81405
7	98	0.90796
8	43	0.96283
9	9.77	0.98872
10	1.56	0.99701
11	0.41	0.99887
12	0.13	0.99948
13	0.06	0.99967
14	0.029	0.99988
15	0.014	0.99993
16	0.011	0.99994
17	0.005	0.99997
18	0.002	0.99999

Table 4: Results with different Number of Decimal Bits(NDB). Using 10 bits for integer part.

6 CONCLUSIONS

As shown in this paper, HLS is useful at development of digital hardware design. As demonstrated in this experiment, algorithm that would take bigger effort to code in HDL (Hardware Description Language) language such as VHDL, can easily be implemented. Doing so with use of only a few lines of simple C code and some directives.

Important thing to consider when using HLS tool is that C code synthesis output does not always meet our idea of hardware. Change of hardware design can be made by exploiting directives. Use of directives like unroll or pipeline can for instance increase algorithm parallelism and shorten latency. More drastic and optimized change in hardware design usually requires slightly different coding style. For instance, Vivado HLS only managed to reduce latency after image array elements were grouped together.

One disadvantage of HLS is definitely unreadable HDL language code at the end of synthesis. Regardless of representation, the best algorithms for hardware realization are not necessarily the same as those used in software. While HLS tools will produce hardware for realizing the algorithm, there may be better or more appropriate algorithms [10].

References

[1] UltraFast High-Level Productivity Design Methodology Guide, Xilinx, UG1197 (v2018.2) June 6, 2018.

[2] Bilinear interpolation, Wikipedia, https://en.wikipedia.org/wiki/Bilinear_interpolation

[3] Image Rotation With Bilinear Interpolation, http://polymathprogrammer.com/2008/10/06/image-rotation-with-bilinear-interpolation/

[4] SDAccel Development Environment Help, HLS Pragmas, https://www.xilinx.com/html_docs/xilinx2018_1/sdaccel_doc/ijd1517252127848.html/

[5] Vivado Design Suite User Guide, High-Level Synthesis, Xilinx, UG902 (v2014.1) May 30, 2014.

[6] Xilinx Vivado HLS Beginners Tutorial, https://medium.com/@chathura.abeyrathne.lk/xilinx-vivado-hls-beginners-tutorial-custom-ip-core-design-for-fpga-59876d5a4119

[7] How-To: Python Compare Two Images, https://www.pyimagesearch.com/2014/09/15/python-compare-two-images/

[8] Zhou Wang, A.C. Bovik, H.R. Sheikh, E.P. Simoncelli, Image Quality Assessment: From Error Visibility to Structural Similarity, IEEE Transactions on Image Processing, Volume: 13, Issue: 4, April 2004.

[9] Image processing in Python, http://scikit-image.org/

[10] The Advantages and Limitations of High Level Synthesis for FPGA Based Image Processing.

[11] Ryan Kastner, Janarbek Matai, and Stephen Neuendorffer, Parallel Programming for FPGAs, May 11, 2018.