2023 7th International Conference on Imaging, Signal Processing and Communications (ICISPC) | 979-8-3503-2507-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICISPC59567.2023.00024

Hardware Development of Sphere Intersection in Ray-casting Using High-Level-Synthesis

Kazuki Kurata
Kyushu Institute Of Technology
Kitakyushu, Japan
kurata.kazuki729@mail.kyutech.jp

Akira Yamawaki
Kyushu Institute Of Technology
Kitakyushu, Japan
yama@ecs.kyutech.ac.jp

***Abstract*—Game applications that require high graphics performance have a large processing load, which greatly affects the running time of battery-driven mobile terminals. Mobile terminals that support dynamic partial reconfiguration of hardware can perform high-performance and low-power graphics processing through power-efficient hardware execution, rather than software. This paper develops hardware for the intersection determination process in ray-casting, which is utilized in the creation of high-graphic game applications. In the development, the HLS, which is an automated software-to-hardware conversion tool is used. We developed the software of intersection determination process so that the HLS can generate small-resource-usage, high-performance, and power-efficient hardware. Intersection determination of ray-casting is carried out by vector and square root calculation. Usually, these calculations are executed in floating-point operation. However, this operation method tends to increase the size of the circuit because it requires a great deal of resources when implemented hardware. It is not desirable due to a huge circuit size means high power consumption. Therefore, this paper shows an optimal method in vector and square root calculations using fixed point, while considering its accuracy. Also, through demonstrations on actual machines, we evaluate its usefulness in terms of performance, circuit size and power efficiency.**

Keywords—High-level Synthesis, FPGA, Ray-casting

I. INTRODUCTION

In various applications for smartphones, game applications field especially have a rapid increase in the scale of development. Game applications which have excellent graphics have been developed to gain a dominant share in the expanding market. The existence of rendering technology, which synthesizes realistic images and videos from data embedded in programs, plays a significant role in this graphic improving trend. Since rending programs can be implemented with simple algorithms, its technology is particularly useful for creating realistic content. The rendering, however, has the disadvantage of high processing load due to the enormous number of calculations. Unlike a game console that is connected to a fixed power supply, a mobile device which runs a game application with a battery must concern this high processing load. Therefore, it is necessary to develop rendering processes that can achieve high performance and battery saving to operate for a long time.

Current mobile terminals run software code of game application on an embedded processor with high clock frequency leading to high power consumption. To achieve high performance and low power consumption simultaneously, hardware implementation instead of software is desired. To handle various kinds of application by their own hardware modules optimally developed, the system must change the hardware configuration in runtime.

To make hardware configurable among many game applications, we think a mobile terminal with a user available field-programmable gate array, FPGA, can reconfigure digital circuits[1,2]. The FPGA has a function which is called dynamic partial reconfiguration, DRP. The DPR can rewrite a specific part of the circuit without stopping the entire system. Using this function, it is possible to switch the hardware module for each application to be executed on the mobile terminal, not affecting other software application running. We are also developing software libraries which can be converted to efficient hardware modules automatically by high-level synthesis, HLS [3,4,5]. Current HLS oriented libraries currently we are developing are about image processing, game programming [6,7,8,9].

This paper develops HLS oriented ray-tracing software, which is one of the rendering algorithms and makes the graphical image more realistic one. The raytracing includes a basic function, ray-casting which traces the ray from the eye to the objective on the screen. In addition, ray-casting consists of several sub-functionalities, such as ambient light, diffusion light, and specular light calculation. This paper focuses on the sphere Intersection in ray-casting and develops its HLS oriented software.

This paper is organized as follows. Section 2 briefly describes the ray-casting algorithm. Section 3, we show the HLS oriented software description for sphere intersection in ray-casting, considering the hardware characteristics. Section 4 discusses the effect of hardware implementation through experiments. Finally, section 5 concludes this paper.

II. SPHERE INTERSECTION

A. Overview of Sphere Intersection in Ray-casting

When we usually perceive an object as visual information, light goes through the following process. That is, light is emitted from a light source, reflected by an object, and then comes into our sight. This process needs to be handled if we should produce a program which can calculate the light flow faithfully.

979-8-3503-2507-2/23/$31.00 ©2023 IEEE
DOI 10.1109/ICISPC59567.2023.00024

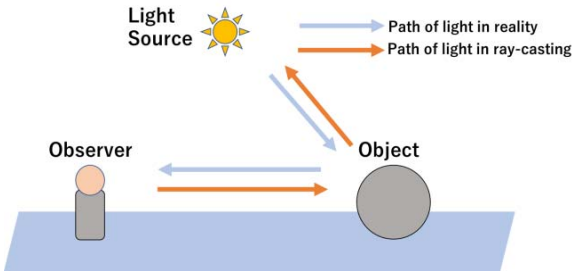

Fig. 1 Light Flow in the Real Case and Ray-casting Case

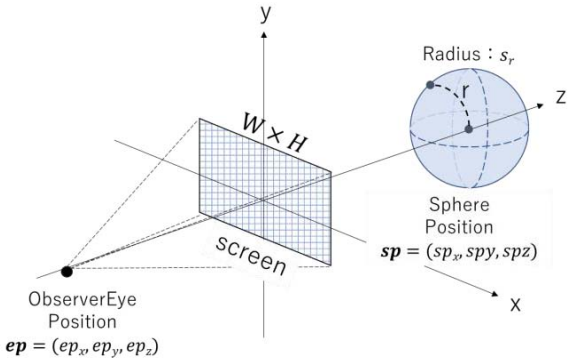

Fig.2 Screen Setup

In actual cases, however, it is not realistic to calculate all the light emitted from the light source due to the number of calculations required. Therefore, ray-casting starts tracking from the observer, as shown in Fig. 1.

In this method, rays which represent a line of sight are generated from the observer. Rays are used to locate objects. By examining the positional relationship between the ray and the object, it is possible to determine whether there is a "sphere intersection", the point at which the sphere and the ray intersect.

B. Screen Setting

First of all, ray-casting need to define a screen as shown in Fig. 2. In the previous section, we explained that the ray is released by the observer. However, our vision is not a line, but has certain areas. The screen is used to provide a pseudo this field. Rays emitted from the observer pass through each point on this screen and sphere intersection determination is done for all rays. The result of each determination is stored as color information on the screen. Finally, when all the information is assembled, it becomes output image.

The screen on the computer has the size indicated by unsigned integer ranging from 0 to a maximum width or height. However, the world coordinate of ray-casting is generally expressed by the real number ranging from -1.0 to 1.0. Thus, the integer coordinate is transformed to such real coordinate. The transformation from integer coordinates $\boldsymbol{i} = (i_x, i_y)$ to world coordinates of screen $\boldsymbol{w} = (w_x, w_y, w_z)$ is as follows.

$$\begin{cases} w_x = 2i_x / (W-1) - 1.0 \\ w_y = -2i_y / (H-1) + 1.0 \\ w_z = 0 \end{cases} \quad (1)$$

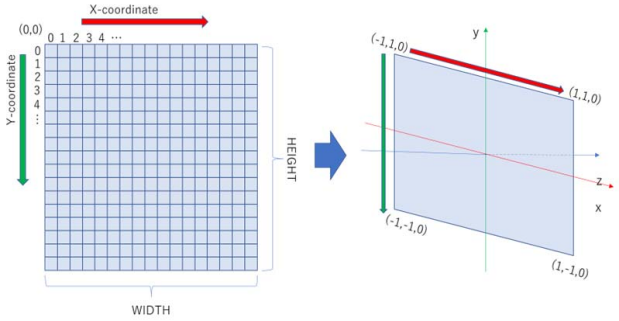

Fig. 3 Coordinate Transformation

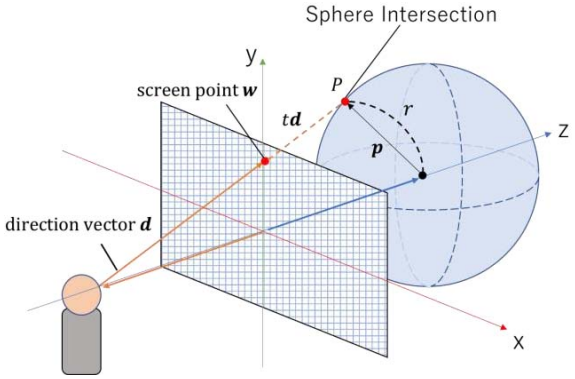

Fig. 4 Relations of vectors used in ray-casting.

C. How to determines the Sphere Intersection

The sphere intersection is calculated by an equation of the second degree which indicate the relationships between observer and object, sphere. So, the ray equation must be defined to build such quadratic equation. The ray equation is represented by a vector of half-lines which have a starting point and extend from it. This can express the ray emitted from the observer. The ray equation is defined by the sum of the starting point vector $\boldsymbol{s} = (s_x, s_y, s_z)$, which determines the observer position, and the direction vector $\boldsymbol{d} = (d_x, d_y, d_z)$,which determines where the observer is facing. And it can be expressed using the parameter $t(t \geq 0)$ as follows.

$$\begin{cases} x = s_s + td_x \\ y = s_y + td_y \\ z = s_z + td_z \end{cases} \quad (2)$$

By using $\boldsymbol{p} = (x, y, z)$ which represent an arbitrary point, we can transform Eq. (2) as Eq. (3).

$$\boldsymbol{p} = \boldsymbol{s} + t\boldsymbol{d} \quad (3)$$

Here, t means the distance from the observer to an arbitrary point $\boldsymbol{p}$. Therefore, in the determination of sphere intersection, it is necessary to determine this t to check whether a sphere exists in the extension of the ray direction. Also, the equation of the sphere must be defined because it is required to determine t. Eq. (4) is the equation of a sphere which is centered at the origin.

$$x^2 + y^2 + z^2 = r^2 \quad (4)$$

```
typedef uint32_t U32
typedef uint16_t U16
⋮
void Sphere_Intersection(……){
  U16 x, y;
  U32 wx, wy, wz;
  wz = 0;
  for(y=0;y<HEIGHT;y++){
    for(x=0;x<WIDTH;x++){
    wx =  (2 * (x << 10)) / (WIDTH  - 1) + (1 << 10);
    wy = -(2 * (y << 10)) / (HEIGHT - 1) - (1 << 10);
           ⋮
           ⋮
    }
  }
}
```

Fig. 5 Pseudo Code of World Coordinate Transformation for HLS

Eq. (4) can be rephrased that the norm up to point $\boldsymbol{p} = (x, y, z)$ on the surface of the sphere is r. Expressing this, we obtain Eq. (5).

$$|\boldsymbol{p}|^2 = r^2 \tag{5}$$

The relationship of the ray and sphere equation are shown in Fig. 4.

Based on the above, we can calculate the value of t. Substituting Eq. (5) into Eq. (3) gives Eq. (6).

$$|\boldsymbol{s} + t\boldsymbol{d}|^2 = r^2 \tag{5}$$

Expanding and rearranging the above equation, then we obtain Eq. (7).

$$|\boldsymbol{d}|^2 t^2 + 2(\boldsymbol{s} \cdot \boldsymbol{d})t + |\boldsymbol{s}|^2 - r^2 = 0 \tag{7}$$

For Eq. (7), Let us express $|\boldsymbol{d}|^2$ is A, $2(\boldsymbol{s} \cdot \boldsymbol{d})$ is B, and $|\boldsymbol{s}|^2 - r^2$ is C as shown in Eq. (8).

$$At^2 + Bt + C = 0 \tag{8}$$

Solving the quadratic equation shown in Eq. (8), we can derive an equation for t as in Eq. (9).

$$t = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A} \tag{9}$$

From Eq. (9), we can realize a sphere intersection exists in front of observer when t is zero or more. However, it should also be noted that when calculating Eq. (9), there may be one or two results for t. When there is one t, it means that the ray emitted from the observer intersects the sphere at just one point. On the other hand, when there are two t, it means that the ray penetrates the sphere and there are two intersections. In this case, the only intersection of near side is visible from the observer, thus the smaller t is employed.

III. HLS-ORIENTERD SOFTWARE DESCRIPTION

A. Fixed-point Version of World Cordinate and Intersection

Ray-casting uses vector calculations. Vector calculations are usually done by using floating point operations. However, hardware implementation by floating point is very costly due to the complexity of the circuitry. Therefore, the HLS-oriented code must be converted to fixed-points for less circuit resources.

Briefly, conversion to fixed-point can be achieved by making the operation integer-only by adjusting the digits with a shift operation, while ensuring that no digit's overflow and that

```
typedef int32_t I32
⋮
void Sphere_Intersection(…, epx, …, spx, …, sr){
  ⋮
  for(y=0;y<HEIGHT;y++){
    for(x=0;x<WIDTH;x++){
      ⋮
      I32 dx,dy,dz,tx,ty,tz,A,B,C
      dx = wx - ex; //2^10 <= 2^10 - 2^10
      dy = wy - ey;
      dz = wz - ez;
      tx = epx - spx;
      ty = epy - spy;
      tz = epz - spz;

      // 2^10 <= 2^20 <= 2^10 * 2^10
      A = ((dx * dx) + (dy * dy) + (dz * dz)) >> 10;
      B = ((tx * dx) + (ty * dy) + (tz * dz)) >> 10;
      C = ((tx * tx) + (ty + ty) + (tz * tz) - (sr * sr)) >> 10;
      ⋮
    }
  }
}
```

Fig. 6 Pseudo Code Calculating A, B and C of Eq. (7).

accuracy is ensured. This adjusting is done in three steps. The first is to eliminate decimal operations.

Specifically, we need to shift the number to the left by n bits. Increasing the numerical value allows us to ignore the decimal point that may arise in the calculation. This paper confirms that the system works normally when n=10 from experiments. Using this result, Eq. (1) becomes below.

$$\begin{cases} w_x = 2(i_x \times 2^{10}) \,/\, (W-1) - (1 \times 2^{10}) \\ w_y = -(2i_y \times 2^{10}) \,/\, (H-1) + (1 \times 2^{10}) \\ w_z = 0 \end{cases} \tag{10}$$

To implement Eq. (10), we described an HLS-oriented pseudo code in C language as shown in Fig. 5.

Also, it must be considered that this change will affect other formulae. There is an error in the calculation results due to vector $\boldsymbol{w}$ becomes the order of 2^{10}. For example, the variables placed as A, B and C in Eq. (8) directly affected. To avoid this impact, a correction needs to be carried out in the second step. That is, the parts not related to decimal point, such as observer position vector $\boldsymbol{ep}$, sphere position vector $\boldsymbol{sp}$ and sphere radius r are shifted to the left to match the order of vector $\boldsymbol{w}$.

Then, in the last step, the bits that had been shifted to the left are shifted back to the right. This right shift operation prevents numerical overflow in the calculation process that is done subsequently. The HLS-oriented pseudo code in C language for executing above method is shown in Fig. 6.

In Fig. 6, the observer position of ep_x, ep_y and ep_z, the sphere position of sp_x, sp_y and sp_z, and the sphere radius of sr are parameterized via the arguments of function, Sphere_Intersection. They are fixed-point numbers whose fractional part has 10 bits. That is, their floating-point numbers have been multiplied by 2^{10} previously. Also, A, B and C calculated using these arguments are adjusted so that they do not overflow by shifting 10 bits to the right.

Fig. 7 depicts the calculation of t in Eq. (9). According to the value of t, the screen is colored to the red which means the eye intersects the sphere.

```
⋮
void Sphere_Intersection(*screen, ex, …, sx, …, sr){
  ⋮
  for(y=0;y<HEIGHT;y++){
    for(x=0;x<WIDTH;x++){
      ⋮
      I32  D = ((B * B) - 4 * (A * C)); //2^20
      UINT SqD;
      INT  t = - 1, tt[3];
      SqD = FixSqrt( D );                  //2^10  <= sqrt(2^20)
      tt[0] = - (B << 8) / (2 * A);        //2^8 <= (2^18 <= 2^10) / 2^10
      tt[1] = ((-B + SqD) << 8) / (2 * A); //2^8 <= (2^18 <= 2^10) / 2^10
      tt[2] = ((-B - SqD) << 8) / (2 * A); //2^8 <= (2^18 <= 2^10) / 2^10
      if( D == 0 ){
        t = tt[0];
      }else if( D > 0 ){
        if( tt[1] > 0             ) t = tt[1];
        if( tt[2] > 0 && tt[2] < t ) t = tt[2];
      }
      if( t > 0 ) screen[i] = RED;
      else screen[i] = Blue;
      ⋮
    }
  }
}
```

Fig. 7 Pseudo Code of Determining Sphere Intersection

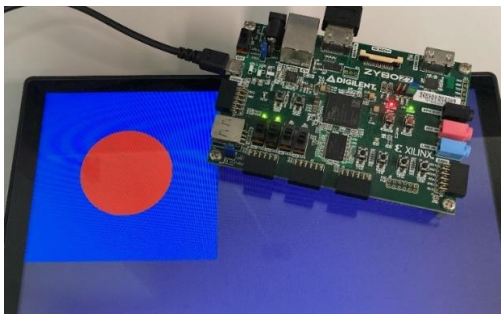

Fig. 9 Experimental Setup

B. *Fixed-point Implementation of Square Root*

Ray-casting uses square root calculation in Eq. (9). To implement a high-level synthesizable fixed point square root, we employed square root extraction [10]. The square root extraction can realize the square root with the fixed number of loop iteration. Thus, the HLS can generate a well-organized hardware module with ideal pipelined structure. The size of fixed-point number of the square root extraction may be selectable through the tradeoff among the hardware size and accuracy. We have developed the HLS-oriented square root extraction that can select the fixed-point number width as shown in Fig. 8.

IV. Experiment

To conduct demonstration of high-level synthesis hardware, we developed a prototype using a commercial FPGA board, as shown in Fig. 9. The commercial FPGA board is Digilent ZYBO Z7-10, and the FPGA on board is Xilinx ZYNQ. This prototype can be connected to a display to visually confirm the results of the Sphere Intersection The display shown in Fig.9 indicates the results of the sphere intersection. The results of the display are colored red for the sphere part and blue for the rest, as shown in the program in Fig 8. The high-level synthesis tool used in the experiments was Xilinx Vivado HLS 2022.2, and the FPGA implementation tool was Xilinx Vivado 2022.2.

```
#if BITWIDTH == 16
typedef uint16_t UINT
#else
typedef uint32_t UINT
#end if

UINT FixSqrt(UINT x){
  if( x==0 ) return 1;
  int shift = BITWIDTH/2;
  UINT root, sub, rem;
  for(i = 0; i < shift; i++){
    rem=(rem<<2)|(x>>(BITWIDTH-2));
    x <<= 2; root <<= 1; sub <<= 1;
    UINT chk=rem-sub-1;
    if(chk>=0)
      root += 1, sub += 2, rem = chk;
  }
  return root;
}
```

Fig. 8 Pseudo Code of Square Root Extraction for HLS

A. *Amount of Hardware*

We generated three types of hardware of sphere intersection, which are (1) Floating point hardware (HW_{float}), (2) Fixed point hardware with 32bit square root (HW_{32bit}), and (3) Fixed point hardware with 16bit square root (HW_{16bit}).

Fig. 10 shows that the number of lookup tables (LUT) and flip-flop (FF) that would be required when this hardware are implemented in FPGA with the HLS tool. From Fig. 10, converting to the fixed-point version can reduce the circuit size by 92% at the maximum. Also, since the limited resources on the FPGA did not allow implementation of the floating-point hardware, only the fixed-point version was implemented on the actual machine, and the performance of these two versions was checked in subsequently section.

B. *Execution Time*

This section evaluates the performance of the implemented hardware from the point of execution time. The software programs were executed on the embedded CPU (650 MHz) in ZYNQ and on the CPU (3.6 GHz) of the PC.

Fig. 11 shows the measured execution time. The results shows that our hardware achieved the performance improvements of about 85% and 94% compared to PC-CPU and embedded CPU.

Comparing HW_{32bit} and HW_{16bit}, the 16bit implementation of square root extraction was only slightly faster, but we can recognize the difference is negligible. In other words, it was found that the bit width of square root extraction has no effect on the operating speed. This is because the effect of a pipelining. In the square root extraction, the number of loops in the function is only eight more implemented with 32 bits than implemented with 16 bits. When executing this in software, the number of loops is directly related to the function run time because of sequential processing. However, parallel processing by pipelining is available in hardware. Thus, it is considered that the minor difference in the number of loops did not contribute to the execution time.

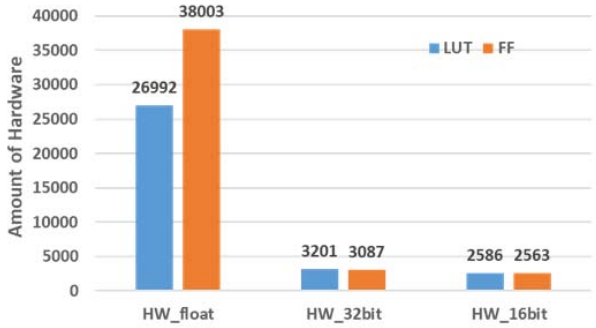

Fig. 10 Amount of Hardware

C. *Power Efficiency Estimation*

Eq.(11) gives the formula which estimates the power efficiency, with taking into account the hardware circuit size.

$$Power\ Efficiency\ of\ HW_i = \frac{Power\ Consumption\ of\ SW}{Power\ Consumption\ of\ HW_i} = \frac{Execution\ Time\ of\ SW \times f_{CPU}}{Execution\ Time\ of\ HW_i \times f_{FPGA} \times \frac{Amount(HW_i)}{Amount(HW_{ref})}} \quad (11)$$

In Eq. (11), the "Execution Time of SW" is the execution time of the floating-point program on PC-CPU or Embedded CPU. The "Execution Time of HW" is the time when execute fixed-point program in hardware. Also, f_{CPU} is 3.6 GHz or 650 MHz and f_{FPGA} is 100 MHz.

The value of HW_{float} in Fig. 10 was used for the reference hardware circuit scale as Amount(HW_{ref}). Calculating the power efficiency based on the above, Fig. 12 was obtained. As shown in Fig. 12, both developed hardware achieved high power efficiency. Also, from the differences of power efficiency between the 32bit and 16bit, we can realize implemented by 16bits is better. If accuracy is required in the calculation, 32 bits may be suitable, but in this "sphere intersection detection", it is not required to such level. Therefore, implementing in 16 bits is considered desirable for hardware.

V. Conclusion

This paper has developed the sphere intersection hardware by HLS. The experimental results show that our HLS-oriented description is effective in the hardware implementation for sphere intersection determination process. As future work, we plan to develop remaining function of ray casting, ambient light calculation, diffuse light calculation and specular light calculation.

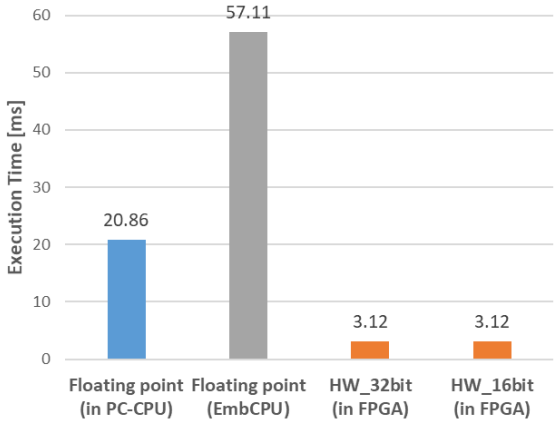

Fig. 11 Execution Time

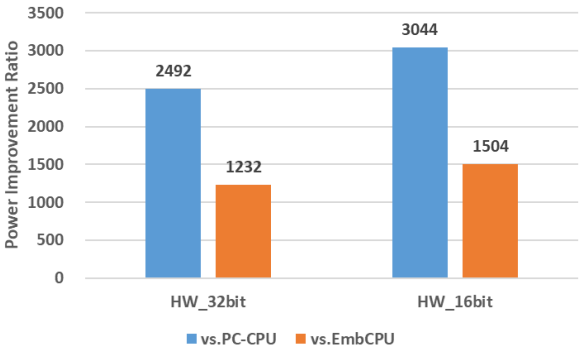

Fig. 12 Power Efficiency

References

[1] Yuki Yamagata and Akira Yamawaki: "A Consideration to Develop a High-level Synthesizable Software Game Library", Proc. of IIAE Int'l Conf. on Industrial Application Engineering 2018, pp.202-205 (2018).

[2] Kazuki Hashimoto and Akira Yamawaki: "Performance Effect of I/O Peripheral Dynamic Reconfiguration on FPGA", Proc. of IIAE Int'l Conf. on Industrial Application Engineering 2021, pp.208-211 (2021).

[3] F. Ferrandi et al., "Bambu: An Open-Source Research Framework for the High-Level Synthesis of Complex Applications," Proc. of ACM/IEEE Design Automation Conference 2021, pp.1327-1330 (2021).

[4] M. A. Özkan et al., "AnyHLS: High-Level Synthesis with Partial Evaluation," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 39, no. 11, pp. 3202-3214 (2020).

[5] Panagiotis G. Mousouliotis and Loukas P. Petrou, " CNN-Grinder: From Algorithmic to High-Level Synthesis descriptions of CNNs for Low-end-low-cost FPGA SoCs," Micro-processors and Microsystems, vol. 73, https://doi.org/10.1016/j.micpro.2020.102990 (2020).

[6] Akira Yamawaki and Seiichi Serikawa: "A Describing Method of an Image Processing Software in C for a High-level Synthesis Considering a Function Chaining", IEICE Transactions on Information and Systems, Vol. E101-D, No. 2, pp.324-334 (2018)

[7] Moena Yamasaki and Akira Yamawaki: "Description Method for High-level Synthesis of Histogram Generation and Their Evaluation", IEIE Transactions on Smart Processing & Computing, vol.8, no.3, DOI:10.5573/IEIESPC.2019.8.3.178, pp.178-185 (2019)

[8] Yuki Yamagata and Akira Yamawaki: "A performance evaluation of read/write burst transfer by high-level synthesizable software for the alpha blending processing", Artificial Life and Robotics, Vol.25, No.2, pp.253-257, DOI:10.1007/s10015-020-00590-x (2020)

[9] Kilryong Lee and Akira Yamawaki: "Background scrolling in high-level synthesis oriented game programing library," Artificial Life Robotics 27, pp.455–460, https://doi.org/10.1007/s10015-022-00758-7 (2022)

[10] Israel Koren, "Computer Arithmetic Algorithms, 2nd ed.", pp. 48-50 (2011)