electronics

MDPI

Article

The Design of a 2D Graphics Accelerator for Embedded Systems

Hyun Woo Oh, Ji Kwang Kim, Gwan Beom Hwang and Seung Eun Lee *

Department of Electronic Engineering, Seoul National University of Science and Technology, Seoul 01811, Korea; ohhyunwoo@seoultech.ac.kr (H.W.O.); jikwang.kim@seoultech.ac.kr (J.K.K.); hwanggwanbeom@seoultech.ac.kr (G.B.H.)

* Correspondence: seung.lee@seoultech.ac.kr; Tel.: +82-2-970-9021

Abstract: Recently, advances in technology have enabled embedded systems to be adopted for a variety of applications. Some of these applications require real-time 2D graphics processing running on limited design specifications such as low power consumption and a small area. In order to satisfy such conditions, including a specific 2D graphics accelerator in the embedded system is an effective method. This method reduces the workload of the processor in the embedded system by exploiting the accelerator. The accelerator assists the system to perform 2D graphics processing in real-time. Therefore, a variety of applications that require 2D graphics processing can be implemented with an embedded processor. In this paper, we present a 2D graphics accelerator for tiny embedded systems. The accelerator includes an optimized line-drawing operation based on Bresenham's algorithm. The optimized operation enables the accelerator to deal with various kinds of 2D graphics processing and to perform the line-drawing instead of the system processor. Moreover, the accelerator also distributes the workload of the processor core by removing the need for the core to access the frame buffer memory. We measure the performance of the accelerator by implementing the processor, including the accelerator, on a field-programmable gate array (FPGA), and ascertaining the possibility of realization by synthesizing using the 180 nm CMOS process.

Keywords: 2D graphics accelerator; embedded system; line-drawing; Bresenham's algorithm; alpha-blending; anti-aliasing

Citation: Oh, H.W.; Kim, J.K.; Hwang, G.B.; Lee, S.E. The Design of a 2D Graphics Accelerator for Embedded Systems. *Electronics* **2021**, *10*, 469. https://doi.org/10.3390/electronics10040469

Academic Editor: Jorge Portilla

Received: 21 December 2020
Accepted: 10 February 2021
Published: 15 February 2021

1. Introduction

Recently, as advances in computer technology and semiconductor process technology lead a processor to high performance and high integration density, the overall performance of an embedded system, such as computing performance and energy efficiency, has been increased [1,2]. Due to the progress of embedded systems, the demand for adopting embedded systems for a variety of applications is also increasing [3–9]. Some of these applications, such as user-centric applications, require communication with users through 2D graphics [10]. Therefore, an embedded system used in these applications requires the functions to process graphics data and write data on the display device. In order to perform these functions, an embedded system, which includes a general-purpose processor (GPP), generally utilizes the GPP or additional graphics processing units (GPUs) with a graphics library [3]. However, performing a graphics process in real-time using these methods requires a high-performance GPP or GPU due to the execution of a large number of instruction codes in a limited time. For this reason, these methods are not appropriate for applications that have limited design specifications such as low power consumption or a small area [10–12].

In order to solve these issues, 2D graphics accelerators, which perform 2D graphics processing implemented in hardware, were proposed for embedded systems [13,14]. These accelerators are connected to the processor in the embedded system through various kinds of interfaces such as PCI Express and memory bus. Unlike the core of a GPP,

which requires a long execution time because it performs only simple operations with one instruction, a hardware accelerator can perform complex operations relatively fast [15–19]. Moreover, the accelerators have a relatively small area because of the limited and optimized execution logic [20–24]. Therefore, including and exploiting the 2D graphics accelerator allows for a variety of applications that require 2D graphics operations to be implemented with low power and small size. As applying architecture to the system that contains a specific accelerator is an efficient way to satisfy the design specifications of the embedded system, research to design the accelerator for image processing has been performed [25].

Line-drawing is one of the methods to visualize the graphics. As every image is represented as a collection of lines, line-drawing is a basic means of drawing an image [26,27]. Accordingly, the line-drawing operation can deal with various kinds of graphics processing [28,29]. Although this approach is not the most efficient way for all situations, this approach is significantly efficient when the data to be displayed are in the form of points and lines. In this point of view, some research was performed to utilize line-drawing for image processing [27]. Nevertheless, there is not a lot of research using line-drawing as a core algorithm for a graphics accelerator. Our research motivation starts with the idea to apply line-drawing for a graphics accelerator.

In this paper, we present a 2D graphics accelerator for embedded systems. The accelerator performs a 2D graphics process with a line-drawing operation based on Bresenham's algorithm. Furthermore, the accelerator provides anti-aliasing and alpha-blending features. The accelerator is directly connected to the memory bus to communicate with the core of the processor in the embedded system. Based on this structure, the accelerator can be controlled through reading or writing to certain memory addresses. Moreover, the accelerator is directly connected to the frame buffer, which has the memory to send 2D graphic data to a display device. This architectural characteristic reduces workloads by offloading the burden of the processor to have access to the frame buffer. We analyzed the performance of the accelerator by simulating and implementing the processor including the 2D graphics accelerator on a field-programmable gate array (FPGA). In addition, we ascertained the feasibility of the accelerator by synthesizing the accelerator with the Synopsys design compiler using the 180 nm CMOS process.

The paper consists of the following: Section 2 describes the preliminaries, which are essential to implement the features of the accelerator. The preliminaries are composed of Bresenham's algorithm, alpha-blending, and anti-aliasing. Section 3 explicates the architecture of the 2D graphics accelerator and explains the reasons for adopting the architecture. Section 4 describes the hardware implementation results, the analysis results of the accelerator through a sample application running on implemented hardware, and the synthesis results through the Synopsys design compiler. Section 5 summarizes our entire work and presents future work.

2. Preliminaries

A line-drawing algorithm is an essential element to implement the presented 2D graphics accelerator. As the algorithms vary according to the design architecture and resource usage of the hardware, choosing an appropriate algorithm is important. We chose Bresenham's algorithm and optimized it for the hardware accelerator [30]. Moreover, in order to provide advanced visualization, supporting additional features such as alpha-blending and anti-aliasing are needed.

2.1. Bresenham's Line Algorithm

Bresenham's line algorithm is one of the line-drawing algorithms and is typically used in raster graphics systems [31,32]. The algorithm calculates the position of the pixels to draw the lines. As this process performs only with integer arithmetic calculation, the process has low complexity and a fast calculation speed [33]. In raster graphics, lines are drawn as a way of painting pixels between the start point and end point. Figure 1 represents the various types of lines by Bresenham's algorithm. The two lines in Figure 1a are the type

that the x coordinates of the painting pixels always increment by one while drawing lines, and the two lines in Figure 1b are the type that the y coordinates of the drawing pixels always increment. The type of the line depends on the slope of the line. The slope, marked as letter m, represents the y-coordinate change, marked as dy, compared to the x-coordinate change, marked as dx, of the line, expressed by dividing dy by dx. The expression of the line is as shown in expression (1) because of the slope attribute as m and the line including the start point (x_1, y_1).

$$m = \frac{dy}{dx}, \quad y = m(x - x_1) + y_1 \tag{1}$$

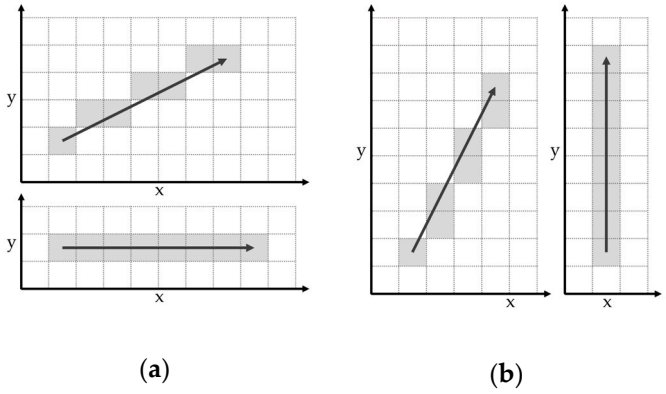

Figure 1. Various lines by Bresenham's algorithm. (**a**) Lines when dx > dy; (**b**) Lines when dx < dy.

Figure 2 presents the fundamentals of the algorithm for drawing each type of line. The algorithm proceeds by selecting the next point to paint based on the current point, marked as (x_i, y_i). Figure 2a shows the case of x coordinates of the points always increment while drawing lines. In this case, choosing the y coordinate of the next point between being changed and not being changed is needed. This job is executed by the following operations. Calculate where the real value y at point $(x_i + 1, y)$ is close to y_i or $y_i + 1$, change the y coordinate when y is close to $y_i + 1$. The algorithm repeats these operations until the current point reaches the end point. In the case of y coordinates of the points always increment, the algorithm proceeds by similar operations as shown in Figure 2b.

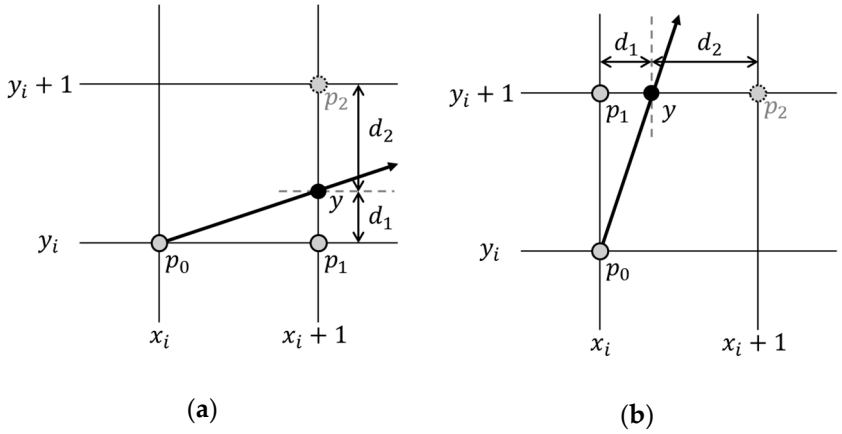

Figure 2. Bresenham's line algorithm according to the line type. (**a**) Lines when dx > dy; (**b**) Lines when dx < dy.

Although the algorithm can be implemented in hardware as it is, optimizing the algorithm for hardware reduces the resource usage. Accordingly, the algorithm should be optimized for hardware implementation by the transformation of the pseudo-code. The following pseudo-code can be obtained through the appropriate transformation of

this process as shown in Algorithm 1. In order to optimize the algorithm, binary division, which has a high cost in hardware implementation, is fully excluded by the transformation. This optimization allows the implemented hardware of the algorithm to achieve the design specifications for embedded systems such as low power consumption and less area.

Algorithm 1: Bresenham's line algorithm pseudo-code

```
    Input  : P_s(x_s, y_s),  P_e(x_e, y_e)
    Output : P = { P_i(x_i, y_i) }
1   i ← 0,  x_i ← x_s,  y_i ← y_s;
2   err ← dx − dy,  e2 ← 2 × err;
3   while x_i ≠ x_e or y_i ≠ y_e do
4       if e2 ≥ −dy then
5           err ← err − dy;
6           x_{i+1} ← x_i + sx;  (sx ← +1 or −1)
7       end
8       if e2 ≤ +dx then
9           err ← err + dx;
10          y_{i+1} ← y_i + sy;  (sy ← +1 or −1)
11      end
12      e2 ← 2 × err,  i ← i++;
13  end
```

2.2. Bresenham's Circle Algorithm

When the width of the line to draw is greater than one pixel's width, drawing the edge of the line to a certain shape increases the quality of the visualization. The circle shape is one of the proper choices. In order to draw circle shapes, we adopt Bresenham's circle algorithm. The algorithm proceedings are similar to Bresenham's line algorithm. Figure 3 shows the rough fundamentals of Bresenham's circle algorithm. Based on the current point $(x_i,\ y_i)$, the algorithm selects the next painting point between $p_1(x_i, +1,\ y_i)$ and $p_2(x_i + 1,\ y_i - 1)$. In order to select the point, calculate the result of the expression (2) by input $(x_i + 1,\ y_i - 0.5)$. The next point is p_2 when the result is lower than 0. Otherwise, the next point is p_1.

$$f = x^2 + y^2 - r^2 \tag{2}$$

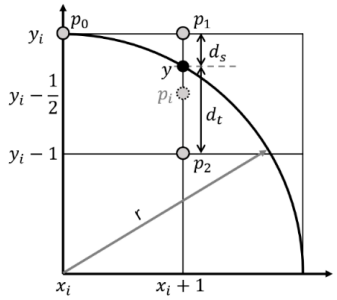

Figure 3. Bresenham's circle algorithm.

2.3. Alpha-Blending

In order to provide drawing graphics with transparency and blending with the original image, alpha-blending is needed. Figure 4 shows the description of alpha-blending. Each pixel's data in the image to draw has an alpha value α to express the transparency. Alpha-blending blends the graphics to draw and the original image by reading the color value of each pixel of the original image and graphics to draw, calculating the new pixel value of

the image frame by expression (3). As the color of the digital image is composed of three color elements—red, green, and blue—the calculation of the new color of pixel p requires calculating each three-color axis.

$$p_{new} = \alpha p_{draw} + (1 - \alpha) p_{original} \quad (3)$$

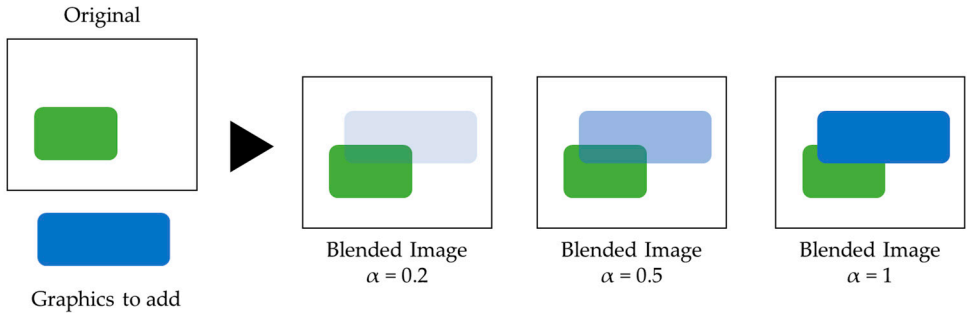

Figure 4. Description of alpha-blending.

2.4. Anti-Aliasing

When expressing a graphical object that has a higher pixel density than the target graphics system, aliasing can be generated because the raster graphics system has limited pixel density. As the line to draw is an ideal graphical object that has unlimited pixel density, the generation rate of aliasing is very high. Anti-aliasing is a technique to deal with this problem. Figure 5 shows the description of anti-aliasing. Anti-aliasing improves visualization of the aliasing-generated lines, such as the line shown in Figure 5a, by blurring the rough edges at the borders of the line. Blurring can be done by decrementing the alpha value of the rough edges sequentially as shown in Figure 5b.

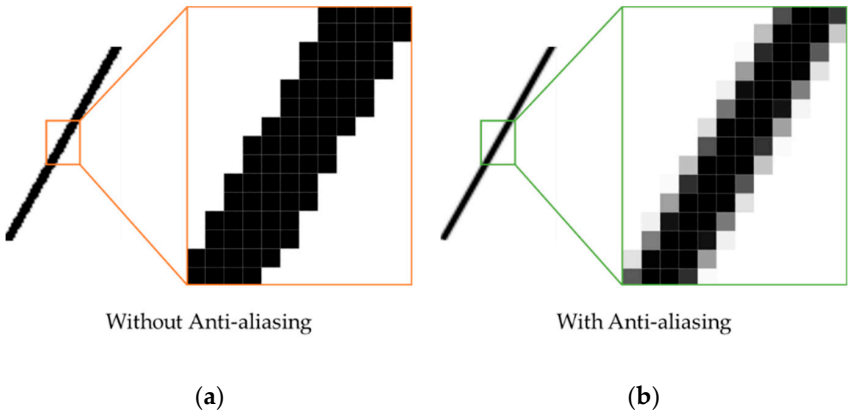

Figure 5. Description of anti-aliasing. (**a**) Line without anti-aliasing; (**b**) Line with anti-aliasing.

The anti-aliasing process starts with detecting the borders of the line. Akin to Bresenham's line algorithm, the anti-aliasing has two types of lines to process, which are related to the slope value. Figure 6 shows the progression of the anti-aliasing process. The anti-aliasing starts with detecting the start point and end point of each border segment. The detection is executed while drawing a line with Bresenham's line algorithm by checking the generated coordinates. Next, as the start point ends and the end point of the border segment is clarified, the process applies the decremental alpha value to each point of the border segment. The following pseudo-code presents the process to apply the alpha value when the slope is lower than or equal to one. The alpha value of the pixel is quantified by three bits, maximum of seven, to reduce the area of the circuit by minimizing the arithmetic calculation.

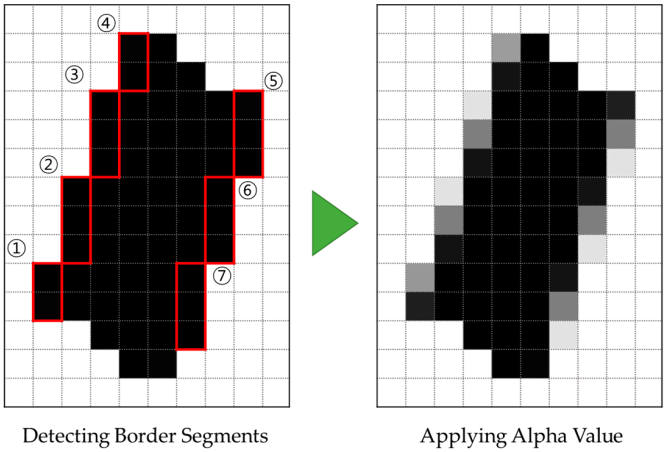

Figure 6. Progression of the anti-aliasing process.

3. 2D Graphics Accelerator

The 2D graphics accelerator provides the 2D graphic processing features including line-drawing, alpha-blending, and anti-aliasing. In order to perform the execution with those features, the accelerator receives setup data, such as start point, end point, the width of the line, bit per pixel (BPP), other configurations, and start flag, from the core of the processor. After the setup data are received and the start instruction is sent, the accelerator operates independently to the core during execution. When the line-drawing process is completed, the accelerator sends the interrupt signal to the interrupt handler of the processor, letting the core recognize the line-drawing process is completed. Based on this characteristic, the workload of the processor is reduced by making it unnecessary for the processor to continuously check what the accelerator completed.

3.1. Line-Drawing Process

Figure 7 presents the progression of the line-drawing process. The setup first receives the line configuration from the core, such as start point, end point, and line width. The module generates the aligned coordinate, slope, line width, and point of the edges from the line configuration and transfers to edge builder. The edge builder sets up the borders of the line by generating the coordinates. The accelerator has three cap modes called perpendicular, vertical, and circle for drawing line caps. Line caps are created by submodules in edge builder. The submodules transfer the minimum and maximum value of x and y coordinates to the line detector module. The line detector starts to process line-drawing by determining what coordinates are borders. The painter generates the coordinates to paint, which are inside the borders, and executes the anti-aliasing process when the anti-aliasing option is set. Finally, the blender paints the pixels with alpha-blending through options transferred from the setup and coordinates from the painter by writing the color to the frame buffer.

3.2. Optimized Architecture

Figure 8 shows the architecture of the processor including the proposed 2D graphics accelerator. As shown in Figure 8a, the accelerator is connected to the core through the memory bus of the processor. For this reason, the core controls the accelerator through memory access instructions. Moreover, the frame buffer is directly connected to the accelerator and connected to the memory bus. Based on this architecture, the core can deal with the conditions that line-drawing is inefficient to process 2D graphics, such as loading a bitmap image to the frame buffer. This characteristic enables the processor to respond flexibly and efficiently to various conditions. Figure 8b presents the architecture of the 2D graphics accelerator.

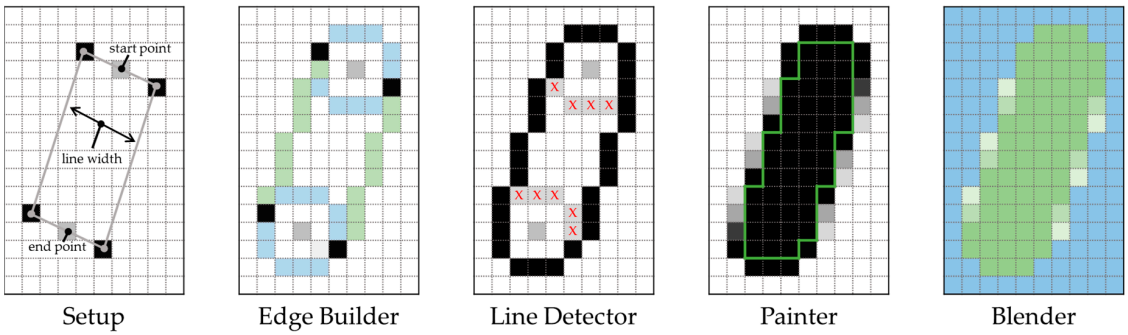

Figure 7. Progression of the line-drawing process.

The accelerator contains the following six modules, called config register, setup, edge builder, line detector, painter, and blender. Config register is a module to save the line configuration and options, such as anti-aliasing and cap mode, from the memory bus. The other modules perform the line-drawing process with options saved in the config register. The five modules, which perform the line-drawing process, operate as a pipelined architecture. Therefore, the accelerator provides high throughput.

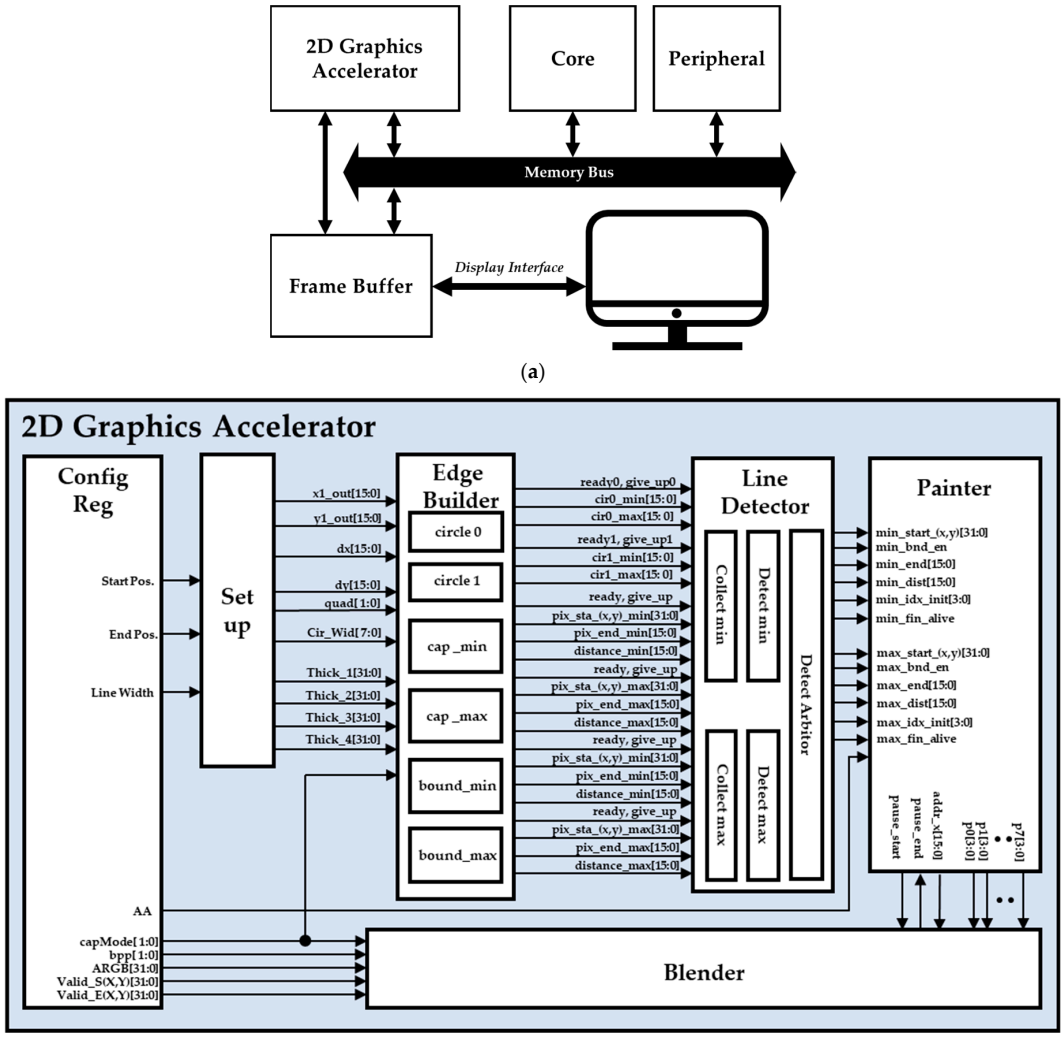

Figure 8. Architecture of the processor and 2D graphics accelerator. (**a**) Architecture of the processor with 2D graphics accelerator; (**b**) Architecture of the 2D graphics accelerator.

In the setup module, the operation to generate the coordinates of the four edges is executed based on the width of the line and the distance between the start point and end point. These coordinates are used for the edge builder module, which is the next pipelined stage. Figure 9 is a block diagram to explain the operations of the edge builder. The edge builder receives the following data signals: minimum and maximum (x, y) coordinates of the points, the distance between the start point and end point (dx, dy), width of the circle to paint when the cap mode is circle, line width, and cap mode. The module generates coordinates of the borders with these signals and submodules. Figure 10 shows all of the cap modes. The edge builder has three selectable cap modes, perpendicular, vertical, and circle, to paint the line caps. The circle submodule generates the coordinates to paint a pixel, which is circular-shaped on edges. The cap submodule generates the coordinates that are parallelogram-shaped, and rectangle-shaped. The line submodule generates borders of the line except for the edges. The entire submodule operates in parallel to provide fast execution. The generated coordinates are sent to the line detector module.

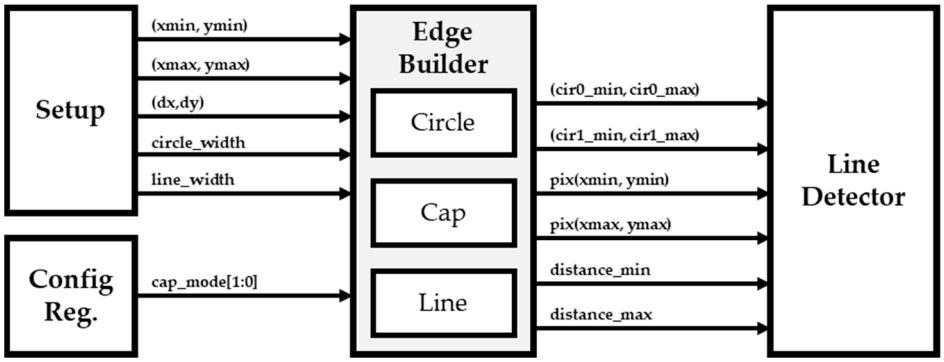

Figure 9. Block diagram of the edge builder.

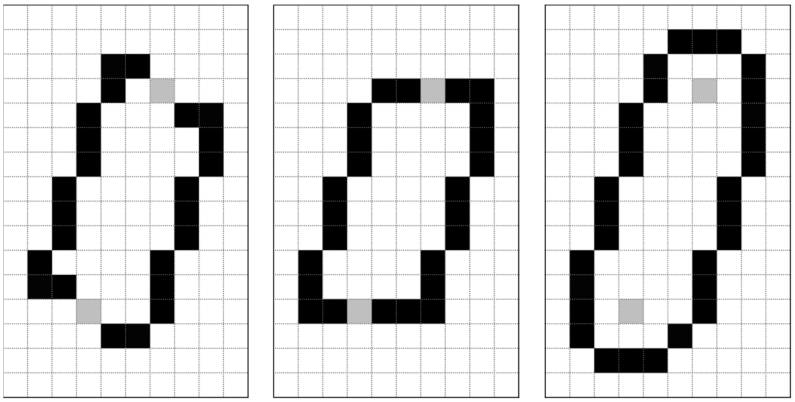

Figure 10. Cap modes of the edge builder.

As the circle submodule generates the whole circular edge, removing the coordinates that are inside the borders is required. This process is done by the line detector module. The line detector receives the coordinates from the edge builder and detects which coordinate is a valid border. Then, it transfers the valid borders, and the minimum and maximum value of the coordinates, to the painter module. The painter module generates the coordinates inside the borders and paints the pixels of generated coordinates by writing the RGBA data to the memory at a certain address. The address to write the RGBA data can be configured by writing the address to the config register through the memory bus. In addition, the module smooths the pixels at borders through the anti-aliasing when anti-aliasing mode is set on the config register. The written RGBA data are used by the blender module. The blender is a module to draw the line to the display device. As the

frame buffer has the previous image drawn, blending the drawing line with the image is required. Therefore, the blender performs the alpha-blending with the previous image and the coordinates of the line to draw. Finally, the blender writes the updated image to the frame buffer, and provides the images to be shown to the display device.

4. Implementation and Analysis

In order to implement and verify the 2D graphics accelerator, we verified the algorithms that are required for the 2D graphics accelerator by programming software. We describe the scripts using MATLAB to verify the algorithms, which are line-drawing, anti-aliasing, alpha-blending, and drawing various line caps. As the algorithms are verified, we transformed the algorithms in accordance with the register-transfer level (RTL) and designed the accelerator with Verilog HDL.

In order to evaluate the 2D graphics accelerator, we integrated the accelerator into the processor, which includes Cortex M0 as a core, by interfacing the accelerator and the core with an AHB-Lite bus. Furthermore, the function that generates the interrupt request signal when the drawing of one line is complete is added. Next, before synthesizing the processor to hardware, we simulated the processor on Vivado 2020.1 version to verify the functionality of the accelerator by executing a customized testbench with a sample program included in the internal ROM of the processor. The embedded program performs the same work as previous MATLAB scripts. The interrupt request signal is generated when the accelerator completes the drawing of one line, and the next configuration of the line is performed by the program.

The synthesis and implementation were executed with the same Vivado tool with a Xilinx xc7z010clg400 FPGA. Table 1 shows the resource utilization of the 2D graphics accelerator and the processor. The result presents that the resource usage of the 2D graphics accelerator is suitable for embedded systems as the utilization of the processor containing the 2D graphics accelerator does not exceed eighty percent of the programmable logic.

Table 1. Resource utilization of the processor including 2D graphics accelerator.

	Resource	Synthesis	Utilization %
2D graphics Accelerator	LUTs [1]	5050	28.69
	Flip-Flops [2]	3087	8.77
	DSP [3]	3	3.75
Processor	LUTs	13,923	79.10
	Flip-Flops	4501	12.79
	DSP	4	5

[1] total of 17,600 [2] total of 35,200, [3] total of 80.

Table 2 presents the performance of the accelerator on 1024 × 768 resolution at 30 frames per second. In order to evaluate the line-drawing performance, we set up the start point and end point as (50, 50) and (700, 900), which are almost the top-left and bottom-right edges of the display, and tested for various conditions such as operating frequency and line width. The result shows that even if the width is as thick as 50 pixels, line-drawing can be performed with more than one line per frame when the operating frequency is more than 50 MHz. According to this result, the accelerator is suitable for a wide range of applications that have resource limitations and line-drawing-based features such as a real-time scope. However, as the results of Table 2 indicate that the drawing efficiency decreases when the width of the line is small, applying the accelerator to complex graphics applications that are not based on line-drawing can be a challenge.

Table 2. Performance of the 2D graphics accelerator.

1024 × 768 @ 30 fps	**Width**					
	1 px			**50 px**		
Clock cycles per line		21,634			246,613	
Operating frequency (MHz)	50	100	120	50	100	120
Times per line (ms/line)	0.43	0.22	0.18	4.93	2.47	2.06
Lines per frame (line/frame)	77	154	185	7	14	16

In order to test the features of the accelerator, line-drawing with various cap modes, anti-aliasing, and alpha-blending, we ran the test firmware on the processor that draws the various kinds of lines by controlling the 2D graphics accelerator with memory access. The processor contains the video graphics array (VGA) controller to display the image in the frame buffer to a display device through a VGA protocol. Consequently, the 2D graphics features, namely line-drawing, alpha-blending, and anti-aliasing, are visually identified by the display device as shown in Figure 11.

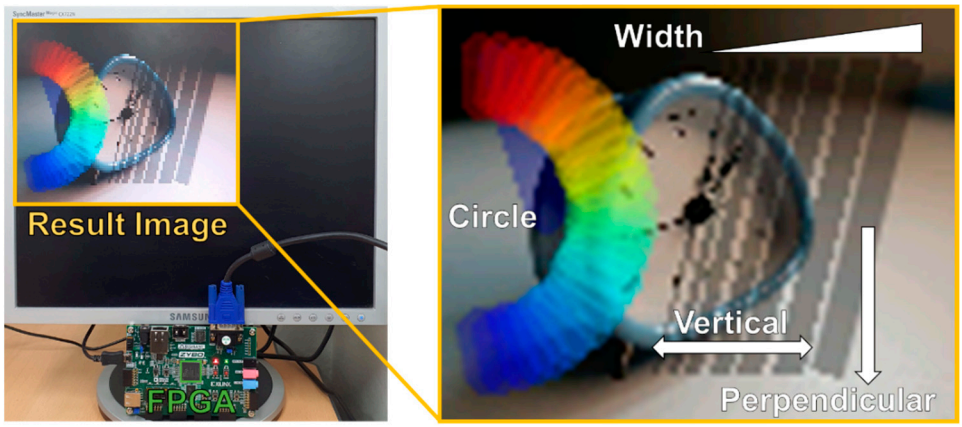

Figure 11. Experimental environment of the field-programmable gate array (FPGA) implementation.

One of the essential things in verifying the feasibility of the 2D graphics accelerator is to identify the area of the actual synthesized circuit. In order to identify the area, we synthesize the accelerator by Synopsys design compiler N-2017.09-SP2 version using the 180 nm CMOS process. Table 3 summarizes the synthesis result. The result shows that the total area of the accelerator is 742,494 um^2, which is around 75K gate counts. The results from Tables 2 and 3 show that the accelerator can be realized through a chip with acceptable performance, drawing more than one line per frame. Therefore, attaching the 2D graphics accelerator to the embedded processor can be a suitable solution to deal with design specifications when the application of the system can effectively be composed with line-drawing features.

Table 3. Synthesis result of the 2D graphics accelerator.

Process Technology	**180 nm CMOS**
Operating frequency (MHz)	100
Area (um^2)	742,494.25
Estimated gate count	75,406

5. Conclusions

In this paper, we proposed a 2D graphics accelerator, based on line-drawing, for embedded systems. As line-drawing can be a basic element of image drawing in specific

applications, defining required 2D graphics as a set of multiple lines is an effective way to implement graphic features rather than other methods. The accelerator provides the basic line-drawing features and user-centric features that improve visualization, such as alpha-blending and anti-aliasing. In order to implement these 2D graphics features, we analyzed the line-drawing algorithm and required functions. Moreover, we optimized the algorithm and functions for hardware realization. By transforming the binary division and reducing the size of arithmetic calculation in the algorithm, the algorithm can be implemented with fewer arithmetic units and enables the hardware to operate with low power and few resources. We also constructed a system-on-a-chip including the accelerator for embedded systems. We also included the designed accelerator in the processor, which is used for embedded systems. The accelerator is connected to the core through the memory bus of the processor to receive line configuration and start signals from the core. As the accelerator is directly connected to the frame buffer, the accelerator works independently of the core while performing the line-drawing process. Based on these characteristics of the architecture, the core can execute other jobs while the accelerator performs graphics processes. As a result, the overall performance of the processor with applications using 2D graphics can be improved. In addition, the results of the FPGA implementation and the synthesis using the 180 nm CMOS process show that the accelerator is feasible to realize.

In future work, we will apply our 2D graphics accelerator to a variety of applications that are implemented on embedded systems, compare the performance of the accelerator with other methods, such as implementation with a GPP or GPU. As the drawing performance of the accelerator is not suitable for complex, microscopic graphic processes, classifying and finding the applications that have appropriate conditions to apply the accelerator is necessary. We expect that applying the 2D graphics accelerator based on line-drawing to the processor can be effective in a variety of embedded systems.

Author Contributions: Conceptualization, H.W.O., J.K.K., and G.B.H.; methodology, J.K.K.; software, G.B.H.; validation, H.W.O., J.K.K., and G.B.H.; investigation, H.W.O. and J.K.K.; writing—original draft preparation, H.W.O.; writing—review and editing, H.W.O. and S.E.L.; visualization, H.W.O. and G.B.H.; supervision, S.E.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT). No. 2019R1F1A1060044, 'Multi-core Hardware Accelerator for High-Performance Computing (HPC)'. This research was also funded by the Ministry of Trade, Industry & Energy (MOTIE, Korea) under the Industrial Technology Innovation Program. No. 10076314, 'Development of lightweight SW-SoC solution for respiratory medical device'.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Yoon, Y.H.; Hwang, D.H.; Yang, J.H.; Lee, S.E. Intellino: Processor for Embedded Artificial Intelligence. *Electronics* **2020**, *9*, 1169. [CrossRef]
2. Guo, F.; Wan, W.; Zhang, W.; Feng, X. Research of Graphics Acceleration Based on Embedded System. In Proceedings of theInternational Conference on Audio, Language and Image Processing, Shanghai, China, 16–18 July 2012; pp. 1120–1124.
3. Cheng, K.; Wang, Y. Using mobile GPU for general-purpose computing—A case study of face recognition on smartphones. In Proceedings of the 2011 International Symposium on VLSI Design, Automation and Test, Hsinchu, Taiwan, 25–28 April 2011; pp. 1–4.
4. Reddy, B.; Kim, Y.; Yun, S.; Seo, C.; Jang, J. Real-Time Driver Drowsiness Detection for Embedded System Using Model Compression of Deep Neural Networks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops, Honolulu, HI, USA, 21–26 July 2017; pp. 121–128.
5. Duffau, C.; Grabiec, B.; Blay-Fornarino, M. Towards Embedded System Agile Development Challenging Verification, Validation and Accreditation: Application in a Healthcare Company. In Proceedings of the IEEE International Symposium on Software Reliability Engineering Workshops, Toulouse, France, 23–26 October 2017; pp. 82–85.
6. Chen, Y.L.; Chiang, H.H.; Chiang, C.Y.; Liu, C.M.; Yuan, S.M.; Wang, J.H. A vision-based driver nighttime assistance and surveillance system based on intelligent image sensing techniques and a heterogamous dual-core embedded system architecture. *Sensors* **2012**, *12*, 2373–2399. [CrossRef] [PubMed]

7. Marchesan, G.C.; Carara, E.A.; Zanetti, M.S.; de Oliveira, L.L. Exploring the Training and Execution Acceleration of a Neural Network in a Reconfigurable General-purpose Processor for Embedded Systems. In Proceedings of the 17th IEEE International New Circuits and Systems Conference, Munich, Germany, 23–26 June 2019; pp. 1–4.
8. Arslan, S.; Gündüzalp, M.; Türk, E. An embedded system vehicle tracking system application. In Proceedings of the National Conference on Electrical, Electronics and Biomedical Engineering, Bursa, Turkey, 1–3 December 2016; pp. 447–451.
9. Yazgaç, B.G.; Kırcı, M. Embedded system application for sunn pest detection. In Proceedings of the 6th International Conference on Agro-Geoinformatics, Fairfax, VA, USA, 7–10 August 2017; pp. 1–6.
10. Kim, J.K.; Oh, J.H.; Yang, J.H.; Lee, S.E. 2D Line Draw Hardware Accelerator for Tiny Embedded Processor in Consumer Electronics. In Proceedings of the IEEE International Conference on Consumer Electronics, Las Vegas, NV, USA, 11–13 January 2019; pp. 1–2.
11. Huang, H.; Liu, Z.; Chen, T.; Hu, X.; Zhang, Q.; Xiong, X. Design Space Exploration for YOLO Neural Network Accelerator. *Electronics* **2020**, *9*, 1921. [CrossRef]
12. Yoo, J.; Krishnadasan, S.; Shin, T.; Lee, W.; Ryu, S. Accelerating vector graphics on low-end device. In Proceedings of the IEEE International Conference on Consumer Electronics, Las Vegas, NV, USA, 8–10 January 2017; pp. 180–181.
13. Pinto, A.; Harish, Y.S. Maximizing Efficiency in Reference Model Based Verification of 2D Graphics Engine. In Proceedings of the Fourth International Conference on Emerging Trends in Engineering & Technology, Port Louis, Mauritius, 18–20 November 2011; pp. 290–295.
14. Tsiktsiris, D.; Ziouzios, D.; Dasygenis, M. A portable image processing accelerator using FPGA. In Proceedings of the 7th International Conference on Modern Circuits and Systems Technologies, Thessaloniki, Greece, 7–9 May 2018; pp. 1–4.
15. Gogte, V.; Kolli, A.; Cafarella, M.J.; D'Antoni, L.; Wenisch, T.F. HARE: Hardware accelerator for regular expressions. In Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture, Taipei, Taiwan, 15–19 October 2016; pp. 1–12.
16. Moini, S.; Alizadeh, B.; Emad, M.; Ebrahimpour, R. A Resource-Limited Hardware Accelerator for Convolutional Neural Networks in Embedded Vision Applications. *IEEE Trans. Circuits Syst. II Express Briefs* **2017**, *64*, 1217–1221. [CrossRef]
17. Lyons, M.J.; Brooks, D. The Design of a Bloom Filter Hardware Accelerator for Ultra Low Power Systems. In Proceedings of the 2009 ACM/IEEE International Symposium on Low Power Electronics and Design, San Fancisco, CA, USA, 19–21 August 2009; pp. 371–376.
18. Dennl, C.; Ziener, D.; Teich, J. On-the-fly Composition of FPGA-Based SQL Query Accelerators Using a Partially Reconfigurable Module Library. In Proceedings of the IEEE 20th International Symposium on Field-Programmable Custom Computing Machines, Toronto, ON, Canada, 29 April–1 May 2012; pp. 45–52.
19. Tumeo, A.; Monchiero, M.; Palermo, G.; Ferrandi, F.; Sciuto, D. A Pipelined Fast 2D-DCT Accelerator for FPGA-based SoCs. In Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI '07), Porto Alegre, Brazil, 9–11 May 2007; pp. 331–336.
20. Cardarilli, G.C.; Di Nunzio, L.; Fazzolari, R.; Re, M.; Silvestri, F.; Spanò, S. Energy Consumption Saving in Embedded Microprocessors Using Hardware Accelerators. *Telkomnika* **2018**, *16*, 1019–1026. [CrossRef]
21. Zhou, Y.; Lyu, Y.; Huang, X. RoadNet: An 80-mW Hardware Accelerator for Road Detection. *IEEE Embed. Syst. Lett.* **2019**, *11*, 21–24. [CrossRef]
22. Melpignano, D.; Benini, L.; Flamand, E.; Jego, B.; Lepley, T.; Haugou, G.; Clermidy, F.; Dutoit, D. Platform 2012, a many-core computing accelerator for embedded SoCs: Performance evaluation of visual analytics applications. In Proceedings of the 49th Annual Design Automation Conference (DAC '12), New York, NY, USA, 3–7 June 2012; pp. 1137–1142.
23. Hegde, G.; Siddhartha; Ramasamy, N.; Kapre, N. CaffePresso: An optimized library for Deep Learning on embedded accelerator-based platforms. In Proceedings of the International Conference on Compliers, Architectures, and Sythesis of Embedded Systems (CASES), Pittsburgh, PA, USA, 2–7 October 2016; pp. 1–10.
24. Simon, W.A.; Qureshi, Y.M.; Levisse, A.; Zapater, M.; Atienza, D. BLADE: A BitLine Accelerator for Devices on the Edge. In Proceedings of the 2019 on Great Lakes Symposium on VLSI (GLSVLSI '19), 2019, New York, NY, USA, 9–11 May 2019; pp. 207–212.
25. Li, Y.; Liu, Z.; Xu, K.; Yu, H. A GPU-Outperforming FPGA Accelerator Architecture for Binary Convolutional Neural Networks. *ACM J. Emerg. Technol. Comput. Syst.* **2018**, *14*, 1–16. [CrossRef]
26. Freeman, H. Computer Processing of Line-Drawing Images. *ACM Comput. Surv.* **1974**, *6*, 57–97. [CrossRef]
27. Ismae, S.; Tareq, O.; Qassim, T. Hardware/software co-design for a parallel three-dimensional bresenham's algorithm. *Int. J. Electr. Comput. Eng.* **2019**, *9*, 148–156.
28. Son, M.; Kang, H.; Lee, Y.; Lee, S. Abstract Line Drawings from 2D Images. In Proceedings of the 15th Pacific Conference on Computer Graphics and Applications (PG'07), Maui, HI, USA, 29 October–2 November 2007; pp. 333–342.
29. Cao, L.; Liu, J.; Tang, X. 3D object retrieval using 2D line drawing and graph based relevance reedback. In Proceedings of the 14th ACM International Conference on Multimedia (MM '06), New York, NY, USA, 14–18 October 2006; pp. 105–108.
30. Dey, N.; Mukherjee, A. *Embedded Systems and Robotics with Open Source Tools*; CRC Press: Boca Raton, FL, USA, 2016; pp. 1–185.
31. Angel, E.; Morrison, D. Speeding up Bresenham's algorithm. *IEEE Comput. Graph. Appl.* **1991**, *11*, 16–17. [CrossRef]
32. Reid-Green, K.S. Three early algorithms. *IEEE Ann. Hist. Comput.* **2002**, *24*, 10–13. [CrossRef]
33. Haque, A.; Rahman, M.S.; Bakht, M.; Kaykobad, M. Drawing lines by uniform packing. *Comput. Graph.* **2006**, *30*, 207–212. [CrossRef]