Output Domain Downscaler

Mert Büyükmıhçı[1], Vecdi Emre Levent[2], Aydin Emre Guzel[2], Ozgur Ates[2], Mustafa Tosun[2], Toygar Akgün[3], Cengiz Erbas[3], Sezer Gören[1], and Hasan Fatih Ugurdag[2(✉)]

[1] Department of Computer Engineering, Yeditepe University, Istanbul, Turkey
sgoren@cse.yeditepe.edu.tr
[2] Department of Electronics and Electrical Engineering, Ozyegin University, Istanbul, Turkey
fatih.ugurdag@ozyegin.edu.tr
[3] ASELSAN, Ankara, Turkey
takgun@aselsan.com.tr

Abstract. This paper offers an area-efficient video downscaler hardware architecture, which we call Output Domain Downscaler (ODD). ODD is demonstrated through an implementation of the bilinear interpolation method combined with Edge Detection and Sharpening Spatial Filter. We compare ODD to a straight-forward implementation of the same combination of methods, which we call Input Domain Downscaler (IDD). IDD tries to output a new pixel of the downscaled video frame every time a new pixel of the original video frame is received. However, every once in a while, there is no downscaled pixel to produce, and hence, IDD stalls. IDD sometimes also skips a complete row of input pixels. ODD, on the other hand, spreads out the job of producing downscaled pixels almost uniformly over a frame. As a result, ODD is able to employ more resource sharing, i.e., can do the same job with fewer arithmetic units, thus offers a more area-efficient solution than IDD. In this paper, we explain how ODD and IDD work and also share their FPGA synthesis results.

1 Introduction

Downscalers are found in many image processing applications. This work addresses video streaming applications and hence needs to be real-time, which opens the door for hardware implementation.

Downscaling produces a lower resolution version of the input image. The purpose is to do this with the least quality loss in the image. The simplest downscaler in the literature is the Nearest Neighbor method (NN) [1]. NN is more area-efficient and easier to implement than other methods, for instance, Bicubic

This work has been partially supported by the Artemis JU Project ALMARVI (Algorithms, Design Methods, and Many Core Execution Platform for Low-Power Massive Data-Rate Video and Image Processing), Artemis GA 621439 [6] and TUBITAK (The Scientific and Technological Research Council of Turkey) Project number 114E343.

T. Czachórski et al. (Eds.): ISCIS 2016, CCIS 659, pp. 262–269, 2016.
DOI: 10.1007/978-3-319-47217-1_28

Interpolation (BcubI) [2] and Adaptable K-Nearest [3] methods. However, the drawback of NN is that the resulting image/frame contains blocking and aliasing artifacts. On the other hand, BcubI can handle blocking and aliasing issues well and produce high quality images; however, because of its complexity and memory requirements, its implementation is difficult and costly. A compromise is possible though. Another method, called Bilinear Interpolation (BlinI) [4], that can also handle blocking and aliasing issues, has lower complexity and hence lower cost than BcubI. Although its output has lower quality than BcubI, the downscaled images it produces are acceptable. Chen [5] proposes an enhanced BlinI downscaler that uses an edge detection algorithm and Sharpening Spatial Filter (SSF) before BlinI to prevent the blurring caused by BlinI.

In this paper, we propose a novel area-efficient implementation of the enhanced downscaler in [5]. We call our downscaler implementation Output Domain Downscaler (ODD) and the straight-forward implementation in [5] as Input Domain Downscaler (IDD). Note that both ODD and IDD apply to also other downscaling algorithms.

IDD tries to output a new pixel every time a new input pixel is received. However, once every few input pixels, there is no downscaled pixel to produce, and IDD stalls (i.e., idles). IDD sometimes also skips a complete row of input pixels. ODD, on the other hand, spreads out the job of producing downscaled pixels almost uniformly over a frame. As a result of that, ODD is able to do more resource sharing, i.e., can do the same job with fewer arithmetic units, thus offers a more area-efficient solution than IDD. In this paper, we implement our ODD architecture with a downscale ratio between 1 and 2 with no loss of generality. That is because it is best to achieve larger downscale ratios of BlinI by applying a downscale ratio between 1 and 2 multiple times. Note that we implemented Verilog RTL generators for ODD and IDD, which are highly parameterized, instead of implementing fixed instances of the two architectures with a specific downscale ratio, fps, and frame resolution. Besides datapath optimizations, we also did memory optimizations as well.

2 The Downscaling Algorithm

The downscaling algorithm implemented in this work is the algorithm in [5], which is based on BlinI. [5] proposes the idea of detecting edges and boosting the pixels around them with SSF in order to circumvent the blur caused by BlinI.

When Edge Detection (ED), SSF, and BlinI are considered altogether, a sliding of 8 input pixels shown in Fig. 1a are used around the downscaled pixel (e.g., pixels P, Q, R). These 8 pixels are used to decide the values of the 4 pixels (pointed to by the arrows) immediately around the downscaled pixel, which are then used by BlinI. In Fig. 1, the input pixels (the dots) are at integer locations, while the downscaled pixels of P, Q, R are at fractional locations with a distance of 1.5 between them, assuming that the downscale ratio is 1.5. If P is at an x coordinate of 1.3, then Q and R are at respectively 2.8 and 4.3. When we take the integer part of these coordinates, we get 1, 2, and 4. These numbers show

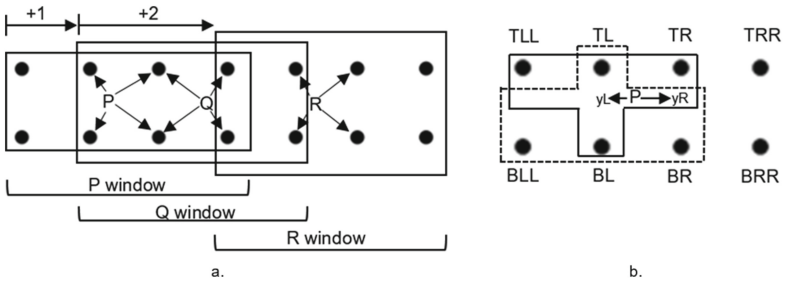

Fig. 1. a. ODD's sliding window b. SSF and BlinI's windows when edge is at L

the starting positions of these consecutive sliding windows. One way to describe this is that the sliding window sometimes shifts by 1 and sometimes by 2. This is our way of looking at it (i.e., the ODD way). Another way to look at this is that sliding window always shifts by 1 but sometimes it does not produce a downscaled pixel. This is the IDD way of looking at it.

Top 4 of these 8 pixels are used for ED. That are the pixels marked with TLL (Top Left Left), TL, TR, TRR as shown in Fig. 1b. In order to find if there are edges at pixel P, the Asymmetry parameter, A, for that pixel needs to be computed as defined by Eq. 1. If A is more positive than a positive threshold, it means that there is a vertical edge at the horizontal position of L (no horizontal edges are considered). If A is more negative than the negative of the same threshold, there is an edge at R.

$$A = |P_{TRR} - P_{TL}| - |P_{TR} - P_{TLL}| \tag{1}$$

Suppose an edge is detected at the horizontal position of L (as opposed to R), then the T-like convolutional window in Fig. 1b is used to recompute the input pixel at location TL, which is the pixel where the edge is detected. The neighboring pixels are multiplied by -1 and pixel TL is multiplied by the sharpening coefficient, S, and the sum is divided by $S - 3$. The pixel below where the edge is detected (BL) is also recomputed by the SSF, hence the dotted window in Fig. 1b. If the edge is detected at R, then SSF shifts the two T-like windows to the right by one position. Hence, SSF uses all 8 pixels to compute two pixels and then replaces either TL and BL pixels or TR and BR.

BlinI computes a downscaled pixel as a weighted average of 4 input pixels surrounding it, i.e., TL, TR, BL, BR pixels. To compute output pixel P, which we also denote by P_{xy}, we first compute two intermediate pixel values (Eqs. 2 and 3, namely, P_{yL} and P_{yR} (see Fig. 1b for locations of yL and yR), as weighted averages of pixels vertically positioned with respect to them, where dy is the weight of the bottom pixel and $1 - dy$ is the weight of the top pixel. Then, we take a weighted average of the two intermediate pixels to compute the pixel value at downscale location (x, y) and arrive at Eq. 4. Note that dx and dy are

respectively fractional parts the x and y coordinates of the downscaled pixel P, in other words, they constitute the displacement of P from input pixel TL.

$$P_{yL} = (P_{BL} - P_{TL})dy + P_{TL} \tag{2}$$

$$P_{yR} = (P_{BR} - P_{TR})dy + P_{TR} \tag{3}$$

$$P = P_{xy} = (P_{yR} - P_{yL})dx + P_{yL} \tag{4}$$

3 Output Domain Downscaler

Consider a video stream at 90 frames per second (fps) and full HD resolution (1920 by 1080 pixels per frame). If the downscaler is running at a clock frequency of 187 MHz, then we will be receiving one input pixel per clock cycle. If we designed the hardware of our downscaler in a brute-force manner (i.e., the IDD way), then we would be shifting our sliding window of 8 input pixels to the right by one pixel every clock cycle just like most designers do in most video streaming applications.

Consider a downscale ratio of 1.8. Then, we would be producing 1067 downscaled output pixels per one line of a video frame. That is, we would be idling in 853 (=1920 − 1067) non-consecutive cycles. We would also be idling for 360 complete lines, each time 1920 cycles back to back. That is because the step size in the vertical direction is also equal to the downscale ratio.

However, since sometimes we would need to produce downscaled pixels in back to back cycles, we would have to design an arithmetic datapath that can execute all operations at a throughput (but not necessarily latency) of 1 downscaled pixel per 1 cycle. Therefore, we would not be able to do resource sharing and would employ as many multipliers as multiplication operations, as many adders as addition operations, and so on.

Fortunately, we do not do it that way; we do it as follows. While IDD shifts the sliding window by one position every time a new input pixel is received (i.e., once every Input Cycle Time, or in short, ICT), we slide the window by the scale ratio, 1.8, in a time period of 3 times ICT (i.e., Output Cycle Time, or in short, OCT). If ICT is 1 cycles per input pixel, then our OCT is 3 cycles per output pixel.

OCT is 3 because we produce N/r^2 output pixels over one frame time if there are N pixels in an input frame. If $r = 1.8$, then we could spread our computations for a downscaled pixel over 3.24 cycles, it would be perfect. However, we have to schedule computations over an integer number of cycles unless we are willing to do loop unrolling. To summarize, $OCT = \lfloor ICT * r^2 \rfloor$.

In our ODD architecture, Output Cycle Time (OCT) determines the cycle time of the datapath (i.e., hence length of the schedule), and that is why it is called "Output Domain". On the other hand, in the naive IDD approach, Input Cycle Time (ICT) determines the cycle time of the datapath, hence the name "Input Domain". OCT is larger than or equal to ICT; therefore, ODD has more opportunity for resource sharing, and in the asymptotic case, uses M/r^2 arithmetic units, whereas IDD uses M arithmetic units.

Fig. 2. a. IDD's top-level b. ODD's top-level

Figure 2 shows the top levels of ODD and IDD architectures. Both ODD and IDD employ a line buffer (Linebuf) and a FIFO. ODD's datapath is connected to the output port of the FIFO, while IDD's datapath is on the input side of its FIFO. Line buffers are, on the other hand, 1 line and 4 pixel long and are due to the 4×2 sliding window the downscaling algorithm uses (shown in Fig. 1).

It is obvious that ODD needs a FIFO. While input pixels are received in raster order at a rate of 1 pixel per cycle, ODD consumes them at a rate of 1.8 pixels (due to the downscale ratio) every 3 cycles. Therefore, it consumes $1.8/3 = 0.6$ pixels per cycle, and as a result the FIFO of input pixels builds up at a rate of 0.4 pixels per cycle. When the downscaler skips a line, then it catches up. It even sometimes leapfrogs the input pixels and waits for the FIFO to fill up as it has a cycle-time of 3 cycles as opposed to the ideal and slower rate of 3.24 cycles.

On the other hand, it is not obvious that IDD needs a FIFO. However, if we have a non-stallable pipeline at the output of the downscaler, and/or we desire to minimize the amount of logic in that pipeline, we need to buffer the downscaled pixels in a FIFO and spread out the computations in the video pipeline that uses the downscaled frames over a pipeline heart-beat of $\lfloor ICT * r^2 \rfloor$ cycles.

ODD's FIFO is a special FIFO; unlike a regular FIFO, it has different width on the write and read sides. It is 1-pixel wide on the write side and 8-pixel wide on the read side. It is indeed a FIFO as all it needs is a push/pop interface with addresses (i.e., write and read pointers) kept inside. Its write pointer is the coordinates of the input pixel that is being received. Its read pointer is the coordinates of the downscaled pixel that is being currently worked on. However, the FIFO outputs 8 input pixels with addresses based on some arithmetic done with the fractional read pointer. Note that in ODD's case, Linebuf can be merged into the FIFO.

Figure 3a gives a procedural code for the downscaling algorithm implemented in this work. Figure 3b shows its Data Flow Graph (DFG). The schedule obtained by mapping this DFG to arithmetic units (columns of the schedule) is shown in Fig. 3c. Every operation in the DFG is named after its output variable. The subscripts of the variable (thus operation) names in the schedule indicate the index of the output pixel, i.e., its order in the video stream. We scheduled ED, SSF, and BlinI separately.

While [5] does all computations in fixed point arithmetic, we do BlinI part in floating point arithmetic since the algorithmic verification model we are given by our image processing people does BlinI in floating point. The advantage of floating point is that it eliminates the engineering time to fine tune the decimal point location in fixed point. Therefore, ED and SSF use integer arithmetic units

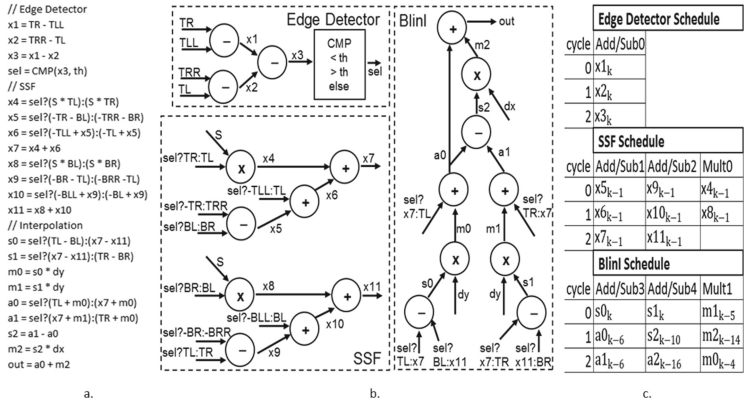

Fig. 3. a. Downscaling algorithm b. Its DFG c. Its schedule for OCT = 3

(non-pipelined), while BlinI uses heavily pipelined floating point units, which is why the degree of functional pipelining in BlinI is quite high ($k-(k-14)+1 = 15$ stages).

4 Synthesis Results

We implemented our architecture not as a fixed RTL design but as a Perl generator that outputs a Verilog RTL design, given design parameters of fps, resolution, clock frequency, and downscale ratio. We targeted a Virtex-7 FPGA. We obtained synthesis results for 90 fps, 1920 × 1080 pixels/frame, clock frequency of 187 MHz, and a downscale ratio of 1.8 for both ODD and IDD.

Hardware resources needed for both ODD and IDD are given in Table 1. Note that FP stands for Floating Point. FP Adders are in fact Add/Sub units. Int. stands for Integer. Although IDD does BlinI with 2 FP multiplications and 4 FP additions/subtractions as opposed to ODD's 3 and 6, respectively, ODD still uses substantially fewer hardware resources.

We have generated and synthesized ODD and IDD for two different cases. One case has an ICT of 1, and the other has an ICT of 2. When OCT is computed for the downscale ratio of 1.8 for these cases, we obtain 3 and 6. Therefore, we have ICT/OCT of 1/3 and 2/6 for these cases.

Linebuf is the same size for both ODD and IDD; however, the FIFO size is different. IDD has a FIFO that is more shallow but wider. That is because it sores the output pixels, which have a 1/1.8 times the rate of input pixels and are wider (32 bits versus 8 bits). Hence, IDD FIFO is 4/1.8 times (45 % of) ODD FIFO. When Linebuf is also taken into account, the memory part of ODD is approximately 60 % of IDD. These numbers are the same for both 1/3 and 2/6 cases.

As for the Datapath, Table 1 first lists the number of arithmetic units per subtask of the downscaler (ED, SSF, BlinI) and the total numbers (Tot.). The number of LUTs and flops these arithmetic units amount to are listed on the lines in

Table 1. Area comparison of ODD and IDD

ICT/OCT	IDD								ODD							
	1/3				2/6				1/3				2/6			
	ED	SSF	BlinI	Tot.	ED	SSF	BlinI	Tot.	ED	SSF	BlinI	Tot.	ED	SSF	BlinI	Tot.
FP Adders	–	–	4	4	–	–	2	2	–	–	2	2	–	–	1	1
FP Multipliers	–	–	2	2	–	–	1	1	–	–	2	2	–	–	1	1
Int. Adders	3	6	–	9	2	3	–	5	1	2	–	3	1	1	–	2
Int. Multipliers	–	2	–	2	–	1	–	1	–	2	–	2	–	1	–	1
Datapath LUTs	4499				2276				2215				1550			
Datapath Flops	3797				2012				1958				1294			
Linebuf Mem.	15392 bits															
FIFO Mem.	37952 bits								17072 bits							
Memory LUTs	3569								2172							
Memory Flops	182								98							
Total LUTs	8068				5845				4387				3722			
Total Flops	3979				2194				2056				1392			

Table 1 that start with “Datapath LUTs” and “Datapath Flops”. The hardware resources ODD needs for the Datapath (LUTs and Flops) are roughly half of what IDD needs in 1/3 case, while it is two thirds in 2/6 case. When we look at the total needed (Datapath + Memory), in 1/3 case ODD requires 54 % of IDD in terms of LUTs and requires 52 % of IDD in terms of flops. Those numbers are 64 % and 63 %, respectively, for the 2/6 case.

5 Conclusion

In this paper, an area-efficient downscaler hardware architecture, called Output Domain Downscaler (ODD) was presented. ODD was compared to Input Domain Downscaler (IDD) architecture, which is the straight-forward approach used in pretty much all downscaler hardware implementations. While ODD is applicable to every downscale algorithm, we have implemented ODD for the downscale algorithm in [5] to show its merits. Our only modification is the use of floating point instead of fixed point in the interpolation stage. We have implemented the same algorithm with IDD as well. We produced ODD and IDD designs from our ODD and IDD Verilog RTL generators for two different cases of input/output rates. We found that ODD uses roughly half the hardware resources of IDD in one case and two thirds in the other case. Hence, we suggest ODD as a viable architecture for a variety of downscale algorithms.

References

1. Caselles, V., Morel, J.M., Sbert, C.: An axiomatic approach to image interpolation. IEEE Trans. Image Process. **7**(3), 376–386 (1998)
2. Nuno-Maganda, M.A., Arias-Estrada, M.O.: Real-time FPGA-based architecture for bicubic interpolation: an application for digital image scaling. In: International Conference on Reconfigurable Computing and FPGAs (ReConFig 2005), Puebla City, pp. 1–8 (2005)
3. Ni, K.S., Nguyen, T.Q.: Adaptable K-nearest neighbor for image interpolation. In: IEEE International Conference on Acoustics, Speech and Signal Processing, Las Vegas, pp. 1297–1300 (2008)
4. Jensen, K., Anastassiou, D.: Subpixel edge localization and the interpolation of still images. IEEE Trans. Image Process. **4**(3), 285–295 (1995)
5. Chen, S.L.: VLSI implementation of an adaptive edge-enhanced image scalar for real-time multimedia applications. IEEE Trans. Circuits Syst. Video Technol. **23**(9), 1510–1522 (2013)
6. Artemis JU Project ALMARVI Algorithms, Design Methods, and Many-CoreExecution Platform for Low-Power Massive Data-Rate Video and Image-Processing, GA 621439. http://www.almarvi.eu