High Efficiency VLSI Implementation of an Edge-directed Video Up-scaler Using High Level Synthesis

Meng Li[1,3], Peng Zhang[2],Chuang Zhu[1],Huizhu Jia[1*],Xiaodong Xie[1], Jason Cong[2,3], Wen Gao[1]
[1]National Engineering Laboratory for Video Technology, Peking University, Beijing, China
[2]Computer Science Department, University of California, Los Angeles, USA
[3]UCLA/PKU Joint Research Institute in Science and Engineering
{mmli,czhu,hzjia,donxie,wgao}@pku.edu.cn,{pengzh,cong}@cs.ucla.edu

Abstract--**Image scaling is a fundamental algorithm used in a large range of digital image applications. In this paper, we propose an efficient VLSI architecture for a novel edge-directed linear interpolation algorithm. Our VLSI design is implemented using high level synthesis (HLS) tool, which generates RTL modules from C/C++ functions. HLS provides significantly improved design productivity compared to the traditional RTL-based design flow. So we explored a large design space including several fine-grained and coarse-grained optimizations in the pipeline architecture design. Our architecture is verified in a working system based on Xilinx Kintex-7 FPGA. Experiments show that our design can process UHD (3840*2160) videos at 30fps with moderate resource utilization.**

Index Terms--**interpolation, VLSI implementation, High Level Synthesis, FPGA, UHD, video scaling.**

I. Introduction

Image scaling is widely used in many digital applications on the consumer electronics field, such as digital camera, mobile phone, high-definition television, tablet computer and so on. In these applications, we need to enlarge the images in a low resolution video to fit the high-resolution screen, especially with the popularity of ultra-high definition (UHD) television.

In general, there are two categories of image scaling algorithms: linear and nonlinear methods. Classical linear interpolation methods have the benefit of low complexity, such as the most widely used bilinear interpolation and bicubic interpolation methods. However, these algorithms tend to generate blocking and aliasing artifacts because they do not take the geometry structures of the image texture into account. To improve the performance, Xin *et al.* [1] proposed a new edge-directed interpolation method, which substantially improves the subjective quality of the interpolated images over conventional linear interpolation by estimating local covariance coefficients based on geometric duality. Zhang *et al.* [2] proposed a new edge-guided nonlinear interpolation technique through directional filtering and data fusion, where missing pixels are fused by the linear minimum mean square-error estimation technique. However, these high quality image scaling algorithms are hard to be implemented in VLSI due to the high complexity and high memory bandwidth requirement. To derive a real-time video scaling application, Kim *et al.* [3] proposed an area-pixel model which has fine-edge and changeable smoothness. Nuno *et al.* [4] proposed a FPGA design based on bicubic interpolation algorithm. Chen *et al.* [5] proposed a VLSI design based on edge-oriented technique, in which a simple edge catching technique is adopted to preserve the image edge features effectively. Chen *et al.* [6] presented a low-cost high-quality scaling implementation and used a T-model convolution kernel to minimize the memory buffers. But the subjective quality of these fast interpolation methods is not satisfactory.

In our previous work [7], an interpolation algorithm based on local structure estimation was proposed, which preserved the local structure well and had a low complexity. In this paper, we propose an efficient VLSI implementation based on [7], including some hardware-oriented algorithm optimization. The VLSI architecture is implemented using high level synthesis [8] tool which raises the level of abstraction beyond register transfer level (RTL) and gives us better control over the optimizations of the system architecture. Some fine-grained and coarse-grained optimization methods for the full pipelining design are also discussed, such as how to handle the streaming data to avoid large line buffer and high memory requirement. Our verification platform is Xilinx Kintex-7 FPGA, on which we explore different design options and get the optimal solution using Pareto-optimal method. Our design can process a video resolution of UHD (3840*2160) at 30f/s in real time with moderate resource utilization, which outperforms all the other existing methods so far.

II. Algorithm Overview

Without loss of generality, the algorithm that implemented to scale up a still image by a factor of two in each spatial dimension is first considered. For 2-D images signals, texture direction is very important since such direction related property of edges directly affects the visual quality around edge areas, which inspires us to build an edge prediction model to get the direction property and estimate the high-resolution pixels from its low-resolution (LR) counterpart based on the geometric regularity in the local area.

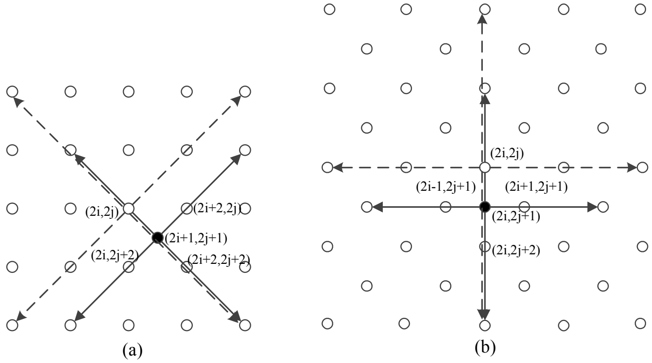


Fig. 1 Proposed image interpolation algorithm when interpolating even pixels(a) and odd pixels(b)

The edge prediction model is a system of linear equations $\vec{\mathrm{b}} = A\vec{x}$ where $\vec{b}$ represents the predicted values on each

This work is partially supported by grants from the Chinese National Natural Science Foundation under contract No.61171139, the Major National Scientific Instrument and Equipment Development Project of China under contract No. 2013YQ030967, and National High Technology Research and Development Program of China (863 Program) under contract No.2012AA011703.

direction with $\vec{b} = (b_1, b_2, \dots, b_M)^T \in R^M$, $A = (a_{i,j})_{M\times N}$ is a $M \times N$ matrix, in which each row represents N pixels in M-th direction, and $\vec{x} = (x_1, x_2, \dots, x_N)^T \in R^N$ is a set of linear filter coefficients. According to [7], the matrix A is constrained to 2×4, which means we use 4 pixels on 2 mutually orthogonal directions, and the area is mark with two mutually orthogonal lines illustrated in Fig.1. Also, vector $\vec{x} = (-1/8, 5/8, 5/8, -1/8)^T$ is a set of typical low-pass filter coefficients. Considering all the pixels in LR image are known, weight values (represented by $\vec{w}$) can be described as prediction errors between the true pixel values and their predicted ones. We can get vector $\vec{w}$ after applying this model to four pixels around the pixel to be interpolated by (1), where $\vec{y} = (p_{i,j}, p_{i,j})^T$and $p_{i,j}$ is the original pixel value.

$$\vec{w} = \sum_{k=1}^{4} (\vec{b} - \vec{y})_k = (w_1, w_2)^T \tag{1}$$

Then the elements of vector w are normalized into 0~1. To achieve the tradeoff between performance and hardware cost, a second order model is used in the normalization:

$${w_1}^* = \frac{{w_2}^2}{{w_1}^2 + {w_2}^2}\ ;\ {w_2}^* = 1 - {w_1}^* \tag{2}$$

At last, the prediction model is applied to the missing pixel and the interpolated value can be obtained by

$$\text{p} = \overrightarrow{w^*}^T A\vec{x} \tag{3}$$

This model is also adaptive when interpolating the interlacing lattice $Y_{i,j}(i + j = odd)$from the lattice $Y_{i,j}(i + j = even)$, as shown in Fig.1(b).

III. VLSI ARCHITECTURE

Different from the traditional design flows based on RTL descriptions such as Verilog, our architecture design is described in C code, and converted into RTL automatically by HLS tools. HLS raises the level of abstraction beyond register transfer level and gives us efficiency in exploring the optimizations of the hardware architecture.

A. Overview structure of our design

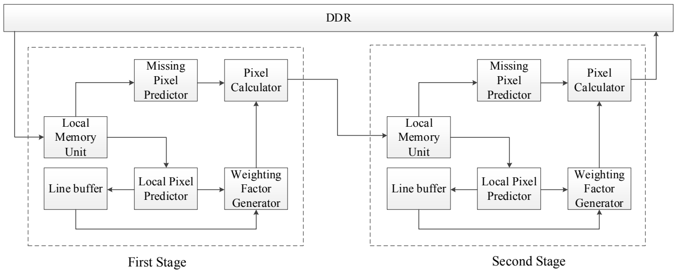


Fig.2 Flow diagram of proposed hardware architecture

Overall, the whole process is divided into two stages. First, pixels $Y_{i,j}(i + j = even)$ are interpolated by the original pixels in LR image, as shown in Fig.1(a). Second, pixels $Y_{i,j}(i + j = odd)$ are interpolated by the original pixels in LR image and the interpolated pixels at stage one, as shown in Fig.1(b). Each stage is composed of six modules shown in Fig.2, local memory unit (LMU), line buffer, missing pixel predictor (MP), local pixel predictor (LP), weighting factor generator (WG), and pixel calculator (PC). This is a line-based streaming structure where the first pipeline stage processes pixels line by line and directly stores them in the parallel memory unit in the second stage; the second stage will start processing when all the pixels required arrive.

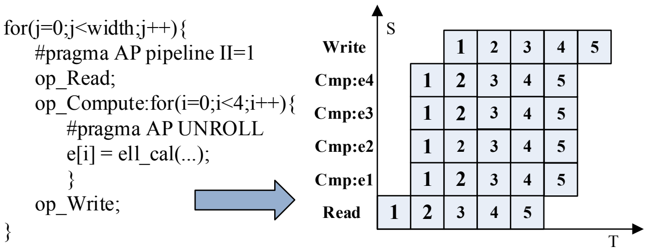
To reduce the complexity of multiplication operation in the computation of $\vec{\text{b}} = A\vec{x}$, we convert the floating-point operations into the fixed-point ones. To keep the computation precision, pixel values are first left-shifted by 3 before the multiplication and right-shifted by 3 after that. In the design, MP and LP have six addition operators respectively. According to(1) and (2), WG has one addition, one subtraction, two multiplications and one division, respectively. Adding up two multiplications and one addition in PC, the whole design has *14* adders, *3* subtracters, *4* multipliers, and *1* divider.

B. Detailed Optimizations

1) Loop unrolling and loop pipelining

By default the iterations of loops in C code are executed sequentially in HLS, and the same operations in different iterations share the same hardware resources. By simply adding a pragma AP UNROLL in loop *i*, multiple parallel hardware duplications of the loop body are created to improve the throughput, as shown in Fig.3.

In addition, pipelining is also a commonly used technique which allows concurrent execution of operations to improve the throughput. By adding a "pipeline" pragma in loop *j* (see Fig.3), HLS will generate the pipeline data path for these operations, allowing executions of successive loop iterations to overlap in time. By HLS, we can avoid the non-trivial register insertion and retiming tuning in manual RTL design.



```
for(j=0;j<width;j++){
    #pragma AP pipeline II=1
    op_Read;
    op_Compute:for(i=0;i<4;i++){
        #pragma AP UNROLL
        e[i] = ell_cal(...);
        }
    op_Write;
}
```

Fig.3 Loop unrolling and loop pipelining

2) Data reuse and Memory partitioning

As we can see from Fig.1, multiple data elements in the same array are required simultaneously in each clock cycle, but the typical on-chip SRAM in FPGAs only have two access ports(such as Xilinx Block RAM). Due to this resource contention, the minimum achievable pipeline initiation interval (II) is limited, which is the cycles between the execution start time of the successive loop iterations. In this design, we propose an architecture combining array partitioning and a two-level data reuse (DR) scheme (block-level and register-level) to increase the pipeline performance.

The calculation of an interpolated pixel as in Fig. 4 requires a window of the neighboring four original pixels. And the windows of the successive pixels overlapped, which provide the opportunity to reuse the input data between loop iterations.

As in Fig.4, buf1_up is used to store a row of intermediate results in the overlapped area. In the same way, buf1_down is used to store the intermediate results in the overlapped area of a single row. Thus, after the block level DR scheme, only two pixels marked by asterisk pass through the pixel prediction model-LP and MP. As described in Fig.4, the two mutually orthogonal lines cover all the pixel access, and there are 14 memory accesses in one cycle period. To further reduce data access redundancy, a shift register chain structure marked by dotted box is proposed to achieve register level DR.

After applying the two-level DR scheme, we need to fetch 5 pixels in one cycle period. Then memory partitioning is used to place the original array into 5 non-overlapping banks, and each bank is implemented with a separate memory block to allow simultaneous accesses to different banks. By rewriting the two-dimensional array to five one-dimensional arrays, HLS will automatically divide the array into 5 partitions (each partition is a single line of the array). Thus, no access conflict and data dependency exists, and pipeline II can be minimized to 1 clock cycle.

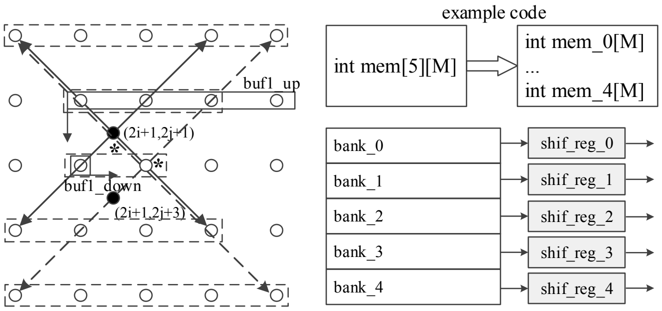


Fig.4 Data reuse and memory partitioning

3) *Loop tiling*

Line buffers typically utilize significant on-chip memory resource in image processing design. With the popularity of UHD resolution, even a single-line buffer will cost huge hardware resource. By simply transforming the *for* loop in C code, the traditional line-based scanning can be changed into a tiled pattern, as shown in Fig.5. Thus, the original line buffer size can be reduced according to the tile size. The overhead of this new scanning method is the pipeline setup time when we start a new row in the tile, which is also related to the tile size. In section IV, we will discuss how to make the tradeoff.

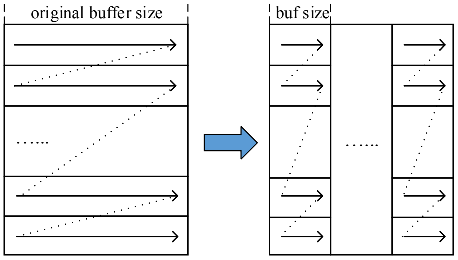


Fig.5 Scanning sequence before and after loop tiling

4) *Double buffer and Software pipelining*

Fig.4 shows that LMU consists of five memory banks after partitioning and each bank stores the data of a pixel line. The problem is that every time before starting the computation process, LMU needs to fetch data from external DDR. To overlap the communication and computation, an additional line buffer is used as double buffer to prefetch data used in next iteration. Besides the double buffer used in LMU, we also use double buffer (buf1 and buf2 in Fig.7) when writing back to DDR.

As for the parallelism inside computation process, software pipelining is adopted to guarantee five lines of data in LMU are under the computation process concurrently while the additional line buffer is reading from DDR. By default, two functions sharing no common arguments will be executed in parallel in HLS. Using a special coding style as in Fig 7, we can achieve the block-level data flow pipelining as in Fig 6.

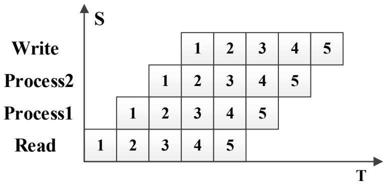


Fig.6 Scheduling of the parallel tasks

Assume that each processing element is composed of 4 components (represented by function read(), process1(), process2(), write() respectively). The first component reads data from DDR and stores them in LMU of stage one; the second component reads from LMU at stage one, performs computation of pixel $Y_{i,j}(i + j = even)$, and then writes the result to LMU at stage two; the third component reads from LMU at stage two, performs computation of pixel $Y_{i,j}(i + j = odd)$, and then writes the result into an temporary on-chip buffer; the last component writes the data back into DDR. The pseudo code is shown in Fig.7, where block1_N and block2_N stand for the Nth bank of the LMU at stage one and stage two, respectively. When coming to a new iteration, function read() refresh a single line data from DDR, in which mod operation is used instead of data shift operation.

```
//pipelining setup
……
for(k=7;k<(height+6);k++){
      if(k%6==0){
            read(block1_1);
            process1(block1_2~block1_6,block2_1,block2_2);
            process2(block2_3~block2_12,buf2);
            write(buf1);
      }
      …
      if(k%6==5){
            read(block1_6);
            process1(block1_1~block1_5,block2_11,block2_12);
            process2(block2_1~block2_10,buf1);
            write(buf2);
      }
}
```

Fig.7 Example code of the software pipelining

IV. Experimental Results

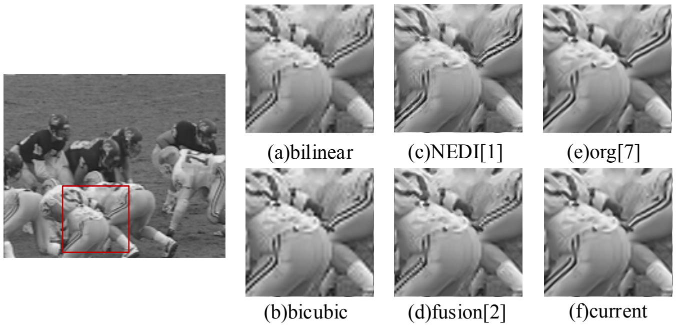
This interpolation algorithm is first described using C code, and then synthesized into RTL using Xilinx Vivado HLS version 2013.2, and implemented into FPGA configuration by Xilinx ISE 14.4. Our verification target is Xilinx Kintex-7 FPGA KC705 platform. The execution performance is measured by a hardware *timer* integrated in FPGA.

A. Subjective qualities and PSNR results

Our design is tested by grey level image, and the PSNR results compared with other publications are listed in Table.1, including BL, BC, NEDI in [1] and fusion in [2]. Our previous work [7] is the software implementation for a hardware-oriented algorithm. Our PSNR result outperforms the first four references and just has a slightly loss compared to [7] as the fixed-point operation and simplified data normalization. Subjective results shown in Fig.8 also validate the excellent quality of our design, especially for the details in the edges.

Table.1 PSNR results of the reconstructed HR image by different methods

Images	BL	BC	NEDI [1]	fusion [2]	org [7]	our
Foreman 352x288	29.82	30.01	30.49	30.46	30.60	30.52
Lena352x288	30.13	30.42	30.53	30.27	30.89	30.84
News 352x288	28.84	29.38	28.73	29.16	30.17	29.94
Football 352x288	27.42	28.04	27.38	27.38	28.61	28.53
Ice 704x576	35.36	35.74	35.92	36.42	36.54	36.27
Crew704x576	36.31	36.98	35.81	36.56	37.03	36.93
Calendar 1280x720	28.10	28.25	27.64	28.32	28.54	28.53
Raven 1280x720	42.52	43.68	42.63	43.03	43.91	43.42
Average	32.31	32.81	32.39	32.70	33.29	33.12



(a)bilinear (c)NEDI[1] (e)org[7]

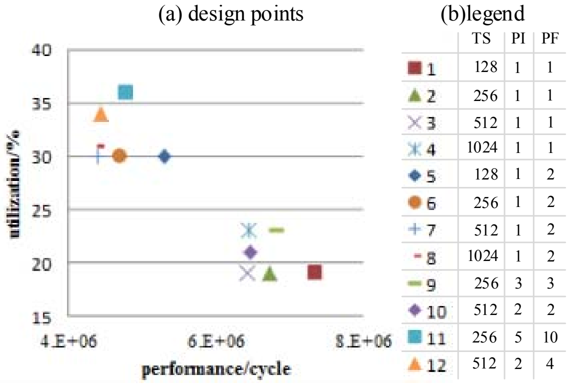
(b)bicubic (d)fusion[2] (f)current

Fig.8 Subjective results of different methods. (e)is the privious work in [7] and (f) is the current result

B. Design space exploration

In this section, we will explore different design spaces, including different tile size, pipeline versus parallelism and different duplication factor to balance the pipeline. As we can see from Fig.6, the pipeline structure is most effective when each task has the same latency. In data communication part, we assign ports counting as allocating bandwidth between function read() and write(). And different duplication factors are tested on data computation part to get balance between communication and computation.

As shown in Fig.9, the performance and hardware resource utilization shown for each design. Each point in Fig.9 represents a design option with specified tile size (TS), pipeline II (PI) and parallelism factor (PF). Large tile size is supposed to have better performance, while memory utilization stands out when TS exceeds 1024, like design 4. Though smaller pipeline II deserves more effort, parallelism brings more significant increase on resource consumption, like design 10 compared with design 3. According to Pareto-optimal method, design 3 and 7 are considered for the final design, where design 3 is a normal design (chosen for the test in section C) and design 7 has duplication resources whereas double throughput.


(a) design points

(b)legend

	TS	PI	PF
■ 1	128	1	1
▲ 2	256	1	1
× 3	512	1	1
✳ 4	1024	1	1
◆ 5	128	1	2
● 6	256	1	2
+ 7	512	1	2
- 8	1024	1	2
— 9	256	3	3
◆ 10	512	2	2
■ 11	256	5	10
▲ 12	512	2	4

Fig.9 Design space exploration

C. Comparison with other VLSI designs

Table.2 lists the comparison results of five low-complexity VLSI designs. Compared with the four previous designs, this work can process a video resolution of UHD (3840*2160) at 30f/s in real time with moderate resource utilization, which outperforms all the other methods.

Table.2 Results comparison with previous works

	Win[3]	BC[4]	edge[5]	method[6]	proposed
FPGA device/ASIC Library	NA	Xilinx Virtex-II Pro	Altera CyclonelI EP2C8F256C6	TSMC's 0.13μm process	Xilinx Kintex-7
Line Buffer	1	6	1	1	NA
Core Area	29K gate counts	890 CLBs	1.06K logic elements	6.08K gate counts	0.98K logic elements
Max Frequency	65MHz	100MHz	109MHz	280MHz	300MHz
Throughput (pixel/s)	NA	NA	200M	280M	300M

V. CONCLUSION

We proposed an efficient VLSI implementation for an interpolation algorithm in this paper. Our VLSI architecture is synthesized using high level synthesis tool, and some fine- and coarse-grained optimization methods were discussed for the pipelining design. We perform design space exploration and find the Pareto-optimal solutions on FPGA target. And our working design can process a video resolution of UHD at 30f/s in real time with moderate resource utilization.

REFERENCE

[1] X. Li and M. T. Orchard. New edge-directed interpolation. IEEE Trans on Image Processing, 10(10): 1521-1527, 2001.

[2] Zhang D, Wu X. An edge-guided image interpolation algorithm via directional filtering and data fusion[J]. Image Processing, IEEE Transactions on, 2006, 15(8): 2226-2238.

[3] C. H. Kim, S. M. Seong, J. A. Lee, and L. S. Kim, "Winscale : An image scaling algorithm using an area pixel model," IEEE Trans. Circuits Syst.Video Technol., vol. 13, no. 6, pp. 549–553, Jun. 2003.

[4] M. A. Nuno-Maganda, et al, "Real-time FPGA based architecture for bicubic interpolation: An application for digital image scaling," in Proc. IEEE Int. Conf. Reconfigurable Computing FPGAs, 2005, pp. 8–11.

[5] P. Y. Chen, C. Y. Lien, and C. P. Lu, "VLSI implementation of an edge oriented image scaling processor," IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol. 17, no. 9, pp. 1275–1284, Sep. 2009.

[6] Chen S L. VLSI implementation of a low-Cost High-quality Image Scaling Processor[J]. Circuits and Systems II: Express Briefs, IEEE Transactions on, 2013, 60(1): 31-35.

[7] Sun Hang, et al. "A Novel Hardware-Based UHD Video Up-Scaler Based on Local Structure Estimation." Advances in Multimedia Inform Process.–PCM 2013. Springer International Publishing, 2013. 430-441.

[8] Jason Cong et al. High-level synthesis for FPGAs: From prototyping to deployment. IEEE TCAD, pages 473–491, 2011.