

Bicubic Interpolation 실시간 하드웨어 설계 및 딥러닝 U-Net(Resblock)을 적용한 후처리

가톨릭대학교 정보통신전자공학부

201520501 박민경, 201621575 한재욱, 201520950 이문규, 201621531 최진원

Abstract

Bicubic Interpolation은 삼차원 함수를 이용한 이미지 보간법으로, 높은 해상력을 보여 컴퓨터 비전 영역에서 많이 사용되는 알고리즘이다. 그러나 처리 시간이 오래 걸려 실시간으로 결과를 확인하기 어렵다는 단점이 있다. 본 논문에서는 이를 해결하여 Bicubic Interpolation의 성능을 기존대로 유지함과 동시에 실시간으로 4배 확장 가능한 하드웨어 시스템을 제안한다. 보간에 필요한 참조 이미지의 픽셀을 저장할 메모리와 8개의 연산 프로세서, 입출력을 번갈아 하는 레지스터 구현했다. 또한 파이프라이닝을 적용하여 다단 Mux, Demux의 임계 경로(Critical Path)를 최소화 했다. 제안한 구조는 verilogHDL로 설계 및 동부하이텍 110nm 라이브러리로 합성하여 최대 동작 주파수 510MHz를 보이며 14K의 게이트를 사용한다. 나아가 제안한 구조의 Bicubic Interpolation 해상력의 한계를 소프트웨어적으로 딥러닝 처리하여, 평균 PSNR 1.3dB 향상된 성능을 보인다.

1. 개 요

최근 크게 발전한 컴퓨터 하드웨어의 성능으로 더 높은 해상도의 그래픽 처리를 더 짧은 시간 내에 할 수 있게 되었다. 이에 FHD를 넘어서 4K, 5K같은 초고해상도 모니터들이 실제로 상용화 되어 가고 있으며 선명한 고해상도의 영상의 공급도 크게 늘어나고 있다. 하지만 과거에 촬영되거나 만들어진 영상은 Vector 그래픽으로 만든 영상이 아닌 이상 영상의 크기를 확대했을 때, 영상의 화질이 크게 손상되는 문제가 있다.

이에 영상 보간법은 많은 연구가 진행되고 있다. 대표적인 알고리즘으로 가장 가까운 주변의 화소를 통해 보간이 이루어 지는 Nearest Neighbor interpolation(최근접 보간법), 인접한 4개의 픽셀값과 거리비를 고려한 Bilinear Interpolation(쌍선형 보간법), 더 나아가 16개의 픽셀을 참고하여 거리에 따른 가중치(weight)를 고려한 Bicubic Interpolation(쌍삼차 보간법) 등이 있다.

이중에 컴퓨터 비전(Computer Vision)에서 이미지 보간법으로 많이 쓰이는 알고리즘이 Bicubic Interpolation이다. Bilinear Interpolation과 대조해봤을 때 참고하는 주위 픽셀이 4배 더 많고, x, y 축의 거리 가중치를 고려함에 따라 보다 더 좋은 해상도의 영상을 얻을 수 있기 때문이다. 최근에는 딥러닝(Deep-learning)이 발달함에 따라 GPU 연산을 통해 여러 딥러닝 모델을 적용해 최적의 보간법의 연구가 많이 진행되고 있다.

그러나 이러한 연구의 한계는 실제 현장(Field)에서 적용하기 어렵다는 점이다. 복잡한 알고리즘으로 인한 수많은 합성곱(convolution)으로 초소형 카메라로 촬영한 영상을 직접 실시간(Real-time)으로 확대하여 처리 결과물을 확인하기가 매우 어렵다. 촬영기기에 우수한 성능을 가진 CPU, GPU를 연동시키기는 비용적으로 힘들뿐더러, 무엇보다 동영상 촬영을 실시간으로 보간하여 결과물을 확인하기에는 힘들기 때문이다. 또한 전성비(電性比)에도 매우 낮은 효율을 보일 수 밖에 없다.

컴퓨터 비전 분야에서 Bicubic Interpolation이 많이 사용되고 있으며 그 중 영상의 4배 확장에 따른 많은 연구가 진행되고 있다. 이에 따라 본 연구는 4배 확장 가능한 Bicubic Interpolation 하드웨어를 구현하였다. 병렬 프로그래밍과 알고리즘을 변형하여 소프트웨어가 가진 한계들을 극복했으며 성능에 따른 비용과 연산의 복잡도 등의 Trade-Off 요소를 여러 방면에서 고려했다. 그 결과 기존 소프트웨어 방식의 성능을 유지하면서 고속처리가 가능한 하드웨어를 구현했다. 나아가 Bicubic Interpolation 알고리즘이 가진 성능의 한계를 극복하기 위해 본 연구의 결과

를 개선된 딥러닝 모델 “U-Net(Resblock)-소프트웨어”를 통한 후처리를 적용 하였다. 이로 인해 성능이 우수한 결과값을 얻어 차후 통합할 하드웨어 설계나 Hw/Sw Co-Design을 위한 선행 연구가 될 수 있도록 제안하였다.

본 논문의 구성은 다음과 같다. 2장에서 Software Bicubic Interpolation 알고리즘을 설명하고 3장에서 제안한 시스템의 구조와 스케줄링에 설명한다. 4장에서는 합성 결과 토대로 성능 분석과 5장에서는 딥러닝을 이용한 후처리의 성능 분석이 이어지고 6장에서는 결론을 맺는다.

2. Software Bicubic Interpolation

그림 1은 Bicubic Interpolation 알고리즘 수식¹⁾으로, 함수 $f(x)$ 는 각 픽셀의 좌표 값을 인자로 받아 거리 가중치를 계산하는 함수다. A, C 행렬은 각각 (i, j) 좌표 기준으로 부터 x, y 축 거리 가중치를 구하기 위한 행렬이며, B 행렬의 I(i, j)는 확장 전 원본 영상의 픽셀 값을 나타낸다. 원본 영상 기준으로 16개의 참조 픽셀 값을 담고 있다. 3개의 A, B, C 행렬곱을 통해 연산된 최종 결과 H(x, y)는 확장된 이미지의 픽셀 값이다.

$$f(x) = \begin{cases} \frac{3}{2}|x|^3 - \frac{5}{2}|x|^2 + 1 & 0 \leq x < 1 \\ -\frac{1}{2}|x|^3 + \frac{5}{2}|x|^2 - 4|x| + 2 & 1 \leq x < 2 \\ 0 & other \end{cases}$$

$$A = \begin{bmatrix} f1 \\ f2 \\ f3 \\ f4 \end{bmatrix} = \begin{bmatrix} f(1+v) \\ f(v) \\ f(1-v) \\ f(2-v) \end{bmatrix}^r \quad B = \begin{bmatrix} I(i-1,j-1) & I(i-1,j) & I(i-1,j+1) & I(i-1,j+2) \\ I(i,j-1) & I(i,j) & I(i,j+1) & I(i,j+2) \\ I(i+1,j-1) & I(i+1,j) & I(i+1,j+1) & I(i+1,j+2) \\ I(i+2,j-1) & I(i+2,j) & I(i+2,j+1) & I(i+2,j+2) \end{bmatrix}^r \quad C = \begin{bmatrix} f1 \\ f2 \\ f3 \\ f4 \end{bmatrix} = \begin{bmatrix} f(1+u) \\ f(u) \\ f(1-u) \\ f(2-u) \end{bmatrix}$$

$$H(x,y) = A \cdot B \cdot C$$

그림 1. Software Bicubic Interpolation 알고리즘

Bicubic interpolation을 보다 쉽게 설명하기 위해 그림 2를 참고 하여, 빨간색 픽셀은 원본 이미지로, 참조해야 하는 픽셀의 좌표이다. 4배 보간을 기준으로 파란색 위치 H(8,12)의 값을 위해 다음과 같은 계산이 이루어진다. 빨간색 픽셀 I(2,3)을 기준으로 x축 거리 비가 [1,0,1,2] 이고 y축 거리 비 또한 동일하기 때문에 원본 이미지의 16픽셀을 참고하여 4번에 걸쳐 x 축 거리 가중치합(weightsum)이 이루어진다. 그 결과에 대한 y축 거리 가중치 곱을 통해 최종값이 정해진다.

$$\begin{aligned} x1_{weightsum} &= f(1) \times I(1,2) + f(0) \times I(2,2) + f(1) \times I(3,2) + f(2) \times I(4,2) \\ x2_{weightsum} &= f(1) \times I(1,3) + f(0) \times I(2,3) + f(1) \times I(3,3) + f(2) \times I(4,3) \\ x3_{weightsum} &= f(1) \times I(1,4) + f(0) \times I(2,4) + f(1) \times I(3,4) + f(2) \times I(4,4) \\ x4_{weightsum} &= f(1) \times I(1,5) + f(0) \times I(2,5) + f(1) \times I(3,5) + f(2) \times I(4,5) \\ H(8,12) &= f(1) \times x1_{weightsum} + f(0) \times x2_{weightsum} + f(1) \times x3_{weightsum} + f(2) \times x4_{weightsum} \end{aligned}$$

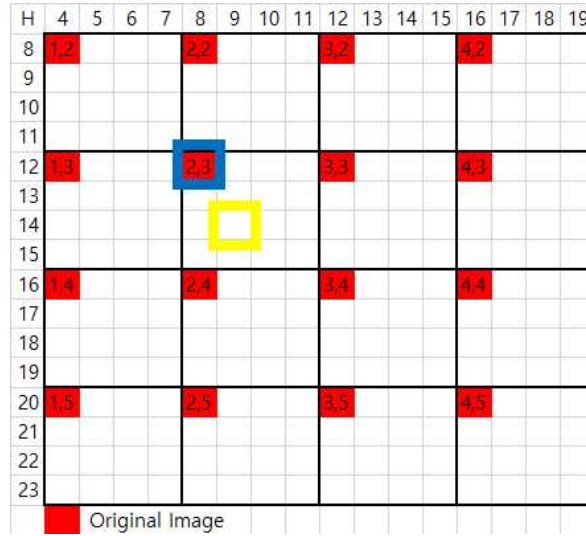


그림 2. Bicubic Interpolation 보조 그림 및 수식

만약 노란 박스, H(9,14) 의 값을 정하게 된다면 빨간 픽셀의 좌표는 변함없이 x축, y축 거리 비율만 달라진다. 이 때 x축 [1.25, 0.25, 0.75, 1.75] 거리 값을, y축 [1.75, 0.75, 0.25, 1.25] 의 값을 함수 f 에 넣어 계산하게 된다.

3. 제안한 시스템의 구조

기존에는 하나의 픽셀을 구하고 다음 픽셀을 구하는 방식 이었다. 본 연구는 8개의 연산 프로세서만으로, 효율적으로 4픽셀 씩 병렬적으로 값을 계산한다. 또한 함수를 구현할 필요 없이, 확장 계수가 정해짐에 따라 거리 비에 따른 함수 값이 상수이므로 이를 면밀히 분석하여 최소한의 비용으로 곱셈기를 구현하였다.

그림 3은 제안한 시스템의 전체 구조로, 4배 확장이 이루어지기 위한 원본 이미지의 16개의 참조 픽셀을 저장하기 위한 쉬프트 방식의 메모리를 구현 하였다. 이는 16:4 Demux를 통해서 행 단위로 4개의 값이 동일하게 X_PE 모듈로 전달된다. X_PE는 x축 특정 거리만큼 가중치 곱을 위한 연산 모듈로, 각 픽셀의 해당 거리만큼 가중합을 계산한다. 이를 통해 동시에 4 픽셀을 보간하기 위한 x축 계산이 진행된다. 계산된 값은 16:16 Demux를 통해 Y_PE 모듈로 나누어 들어가는데, 이를 통해 각 픽셀의 특정 y축 거리만큼 가중합이 계산되고, 최종 보간 작업이 완료 된다.

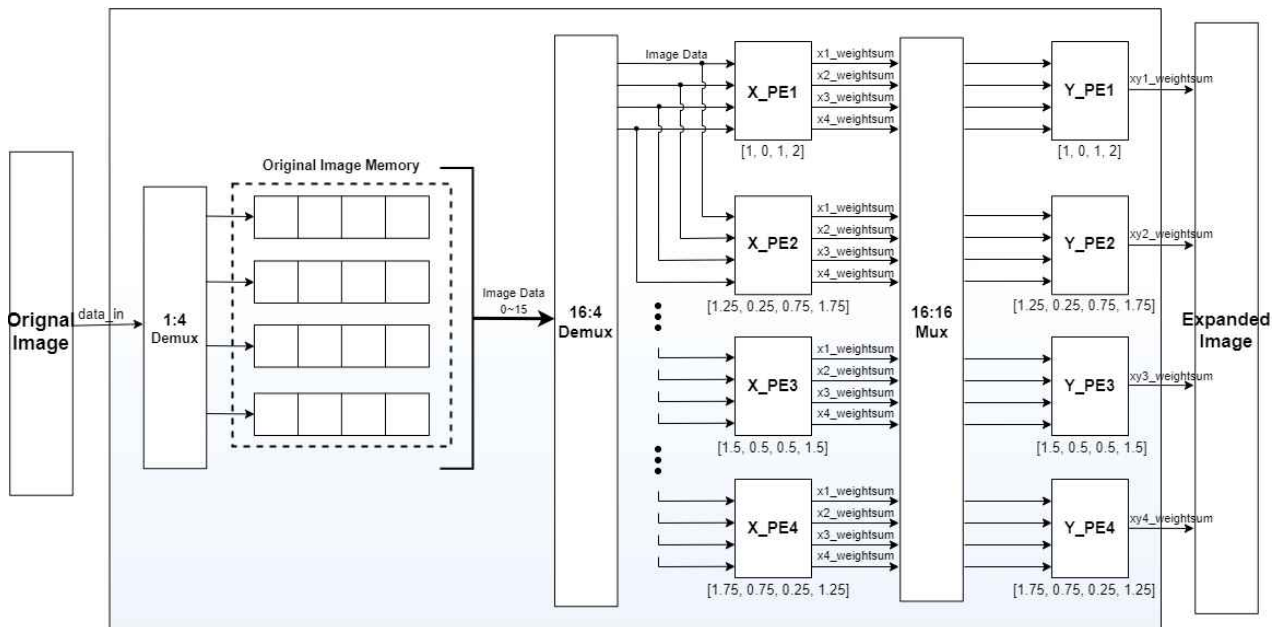


그림 3. 제안한 시스템의 전체 구조

그림 4는 이미지 보간을 위한 순서로, 4배 확장을 위해 한 픽셀당 16개의 보간 값이 들어가야 한다. 제안한 구조

는 16개의 값 중 x, y축 거리 가중치 값이 서로 다른 픽셀끼리 묶어 4 픽셀씩 구하는 방식이다. 1이라 마킹되어 있는 픽셀의 보간 값을 병렬적으로 먼저 구하고, 후에 2,3,4 순서로 이어진다.

H	8	9	10	11
12	1	2	3	4
13	4	1	2	3
14	3	4	1	2
15	2	3	4	1

그림 4 이미지 보간 순서

그림 5는 X_PE 모듈의 구조로, 거리 가중치 곱을 위한 곱셈기와 연산장치를 구현했다. 또한 연산 결과 값을 1:8 Demux로 분배하여 4 픽셀씩 동시에 구하기 위해 Tick, Tock 레지스터에 저장한다. 그 후 8:4 Mux를 통해 Y_PE 모듈로 값을 전달한다.

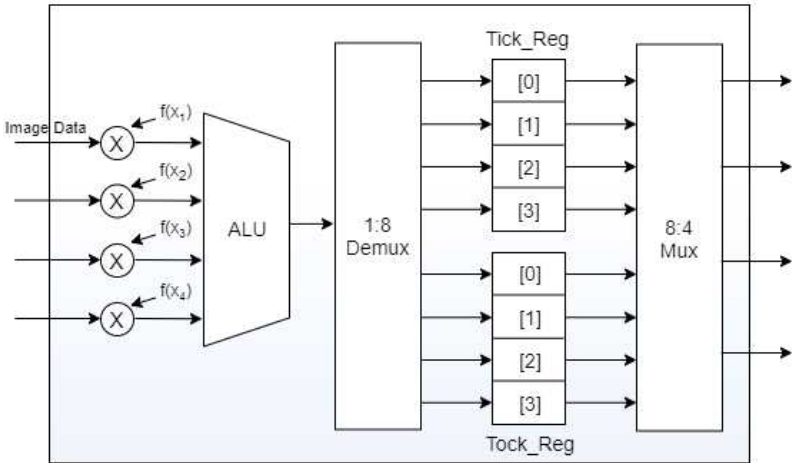


그림 5. X_PE 모듈의 구조

기존 알고리즘에서는 거리 비에 따른 가중치를 계산하기 위해 함수 f(x)를 통해 연산했다. 제안한 구조는 확장 계수의 결정, 규칙적인 거리 비를 적극 활용하여 함수를 구현할 필요 없이 표 1과 같이 상수 값으로 결정하였다. 이를 통해 쉬프트 방식의 곱셈 연산기를 각 X_PE 모듈 번호 마다 지정 거리만큼 곱이 이루어 질수 있도록 구현했다. 이는 Full Adder 기반의 ALU를 통해 x축 방향 가중치 합이 연산이 된다.

표 1. 보간 함수 f(x)의 구현

X_PE	X_PE1				X_PE2			
x	1	0	1	2	1.25	0.25	1.25	2.25
f(x)	0.0000000	1.0000000	0.0000000	0.0000000	-0.0703125	0.8671875	-0.0703125	0.0000000

X_PE	X_PE3				X_PE4			
x	1.5	0.5	1.5	2.5	1.75	0.75	1.75	2.75
f(x)	-0.0625000	0.5625000	-0.0625000	-0.1875000	-0.0234375	0.2265625	-0.0234375	0.0000000

제안한 구조는 실시간 시스템을 위해 x,y 거리 연산 모듈이 중단 없이 병렬적으로 계속 연산할 수 있도록 X_PE 모듈마다 4개의 저장 공간을 가지는 Tick, Tock 레지스터를 구현하였다. 표 2와 같이 4 clock 에 걸쳐 Tick [0],[1],[2],[3] 순으로 그림 2의 가중합(ws)이 저장된다. 다음 16 픽셀을 위해 이후 4 clock 동안, Tock 에 마찬가지로 저장된다. 자세한 설명은 3.1 절에 기술하였다.

표 2. Tick, Tock Register 저장 방식

Clock	1clk	2clk	3clk	4clk	5clk	6clk	7clk	8clk
X_PE1_Tick	[0] x1_ws	[1] x2_ws	[2] x3_ws	[3] x4_ws				
X_PE1_Tock	-	-	-	-	[0] x1'_ws	[1] x1'_ws	[2] x1'_ws	[3] x1'_ws

그림 6은 Y_PE 모듈의 구조로, 그림 3의 16:16 Demux 통해 값들이 적절히 분배되어 Y_PE 모듈에 입력된다. 각 모듈마다 특정 거리의 가중치 곱이 이루어지도록 설계하였다. 곱셈기와 ALU 연산장치 X_PE 모듈과 동일하게 구성하였으며, 이 모듈을 통해 최종 픽셀 값이 결정된다. Y_PE 4개의 모듈로 1 Clock에 4 픽셀씩 값을 결정할 수 있으며, X_PE와 동기화는 3.1절에 기술하였다.

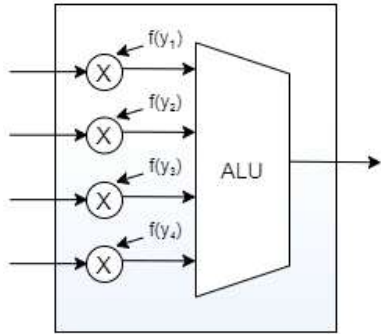


그림 6. Y_PE 모듈의 구조

3.1 X_PE와 Y_PE의 동기화

그림 7은 X_PE의 Tick, Tock 레지스터의 기능을 설명한 그림이다. 4 Clock 에 걸쳐 x축 거리비가 [1, 0, 1, 2] 인 Column 1, [1.25, 0.25, 0.75, 1.75]인 Column 2, 마찬가지로 Column 3,4 의 픽셀 값을 구한 뒤 바로 다음 Clock에 Column 5의 값을 구하기 위해 저장-출력을 번갈아 하는 Tick, Tock을 설계하였다.

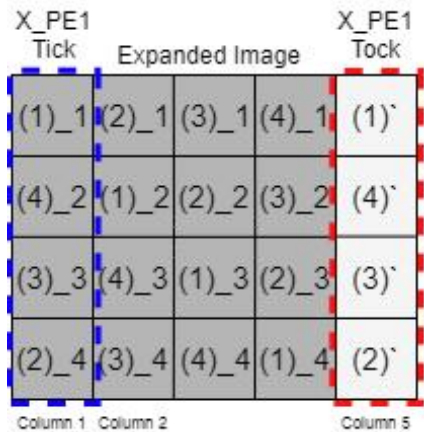


그림 7. Tick, Tock 데이터 스케줄링

표 3과 같이 X_PE1 로 인해 Column1 가중치 합(ws)이 결정되고, 마찬가지로 X_PE 2~4를 통해 Column 2~4 또한 계산되면 최종적인 y 거리 가중치 합을 통한 픽셀 값 결정만 남는다. 이는 Y_PE 4개의 모듈을 통해 1 Clock 에 Column 1~4를 병렬적으로 처리한다. 이로써 좌표 (1)-1,2,3,4 의 픽셀 값들이 정해진다. 다음 Clock 에도 Tick 에 저장된 Column 1~4에 해당하는 x축 거리 가중치 값을 이용해 (2)-1,2,3,4의 픽셀 값이 결정된다.

표 3. 제안한 시스템의 데이터 스케줄링

Clock	1clk	2clk	3clk	4clk	5clk	6clk	7clk	8clk
X_PE1_Tick	[0] x1_ws	[1] x2_ws	[2] x3_ws	[3] x4_ws				
X_PE1_Tock	-	-	-	-	[0] x1'_ws	[1] x1'_ws	[2] x1'_ws	[3] x1'_ws
X_PE1	-	-	-	Column 1	Column 1	Column 1	Column 1	Column 5
X_PE2	-	-	-	Column 2	Column 2	Column 2	Column 2	Column 6
Y_PE1	-	-	-	(1)-1	(2)-1	(3)_1	(4)_1	(1)'-1
Y_PE2	-	-	-	(1)-2	(2)_2	(3)_2	(4)_2	(1)'-2
Y_PE3	-	-	-	(1)-3	(2)_3	(3)_3	(4)_3	(1)'-3
Y_PE4	-	-	-	(1)-4	(2)_4	(3)_4	(4)_4	(1)'-4
data_in	-	-	-	-	-	-	입력/shift	입력/shift

이와 동시에 data 입력이 이루어진다. 그림 3의 Original Image Memory에는 해당 영역의 보간에 필요한 참조 픽셀만을 저장하는데, 이 때 1 Clock 마다 입력하여 동기화 시키게 되면 필요한 참조 픽셀을 적절히 이용하게 된다. 이렇게 X_PE 모듈은 Column 5 값 연산을 위해 원본 이미지의 다음 픽셀을 입력 받아 연산하여 Tock에 저장한다. 그 결과 Tock 출력을 통해 진한 회색의 16 픽셀을 구하고 바로 새로운 영역의 값을 실시간으로 구하게 된다. 4 Clock 동안 Tick 레지스터 출력으로 Y_PE 모듈과 정합이 이루어 질 때, X_PE 모듈은 새로운 데이터를 입력 받아 Tock 레지스터에 저장된다. 4 Clock 후에 이와 반대로 이루어지며, 이로 인해 실시간으로 매 Clock 마다 4 픽셀씩 결과를 얻게 된다.

4. 성능 분석

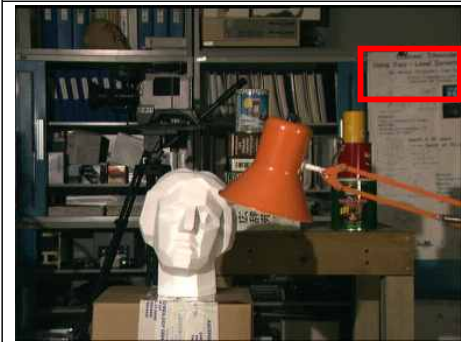
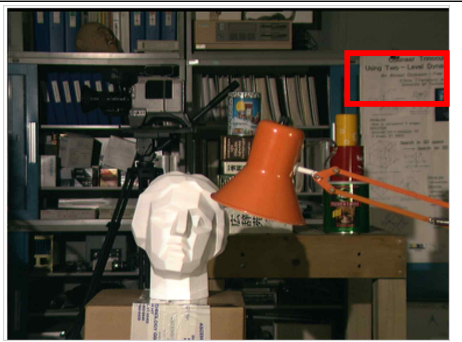
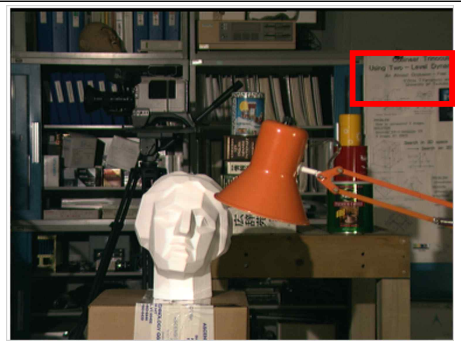
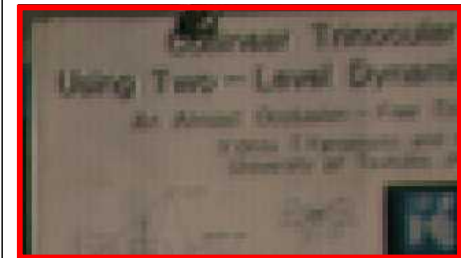
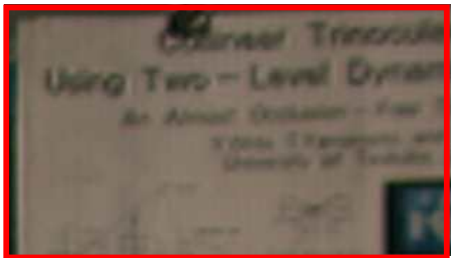
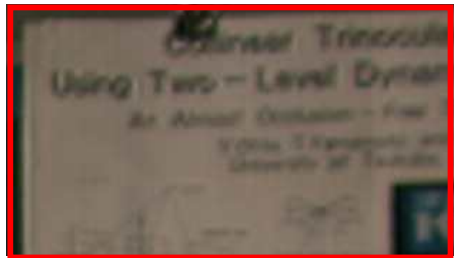
제안한 시스템은 verilogHDL로 설계하여 Modelsim으로 검증하였으며, Synopsis를 사용하여 합성하였다. 이미지의 가로, 세로 각각 2배씩 확장하여 총 4배의 확장이 이루어진다. Gray, RGB 영상 모두 처리 가능하며, RGB 영상의 경우 성분마다 독립적으로 입력하여 처리하는 방식이다. 동부 하이텍 0.11um 라이브러리를 사용하여 합성한 결과 전체 시스템은 14K의 게이트를 필요로 하며 최대 Clock 속도는 510 MHz를 보인다.

표 4. 화면 규격(RGB)에 따른 성능분석

화면 규격	크기	보간법 적용	Software vs Proposed PSNR	fps
HD	1280*720	2560*1440 (QHD)	81.02 dB	184
FullHD	1920*1080	3840*2160 (UHD)	80.35 dB	81
UHD	3840*2160	7680*4320 (8K UHD)	80.49 dB	20
2K	2048*1080	4096*2160 (4K)	80.54 dB	76
4K	4096*2160	8192*4320 (8K)	81.05 dB	20

표 4는 화면 규격에 따른 성능 분석으로 RGB 이미지 기준, 우수한 fps를 가진다. 또한 기존 Software 방식과 비교했을 때 유사한 성능을 보인다. 제안한 구조는 데이터 스케줄링, x, y축 거리 가중치 PE 연산 프로세서, Tick, Tock 레지스터 구현으로 인해 4 Clock에 16 픽셀의 보간 결과를 완성한다. 이로 인해 정지 영상은 물론, 동영상에도 적용이 가능함으로 실시간 시스템의 성능을 만족시켰다. 크기에 따른 이미지의 Software 알고리즘과 제안한 구조 결과 간의 높은 PSNR 수치로 보아 매우 유사하다고 판단할 수 있다. 확장계수가 정해짐에 따라 f(x)의 결과를 표 1과 같이 고정 소숫점으로 구현 가능하며 이는 f(x) 값과 차이가 없다. 다만 차이가 있는 부분은 Y_PE 모듈에서의 곱셈 연산이다. 기존에서는 소숫점 자리끼리 곱이 이루어지면서 자릿수 확장이 이루어지지만 제안한 구조는 하드웨어의 성능을 고려하여 고정 소숫점 곱셈 연산을 적용하므로 근소한 오차가 있다. 그러나 시스템의 출력은 정수 픽셀 값이기에 성능 차이는 굉장히 미미한 수준이다. 표 5는 원본 영상에 기존 알고리즘과 제안한 시스템의 결과를 분석한 영상으로 원본 영상에서 흐릿하게 보이던 텍스트가 보간 결과 비교적 자세히 보인다. 또한 확장된 두 영상간의 PSNR 수치 81.05dB로 매우 높은 유사성을 보인다. 텍스트는 육안적으로 기존 알고리즘과 제안한 시스템의 결과가 비슷하다.

표 5. 시스템 결과의 분석 영상

보간 전 원본 영상 (384 * 288)	Original 알고리즘 적용 (768 * 576)	제안한 시스템 (768 * 576)
		
		

이로써 Bicubic interpolation 의 성능은 유지하며, 14K의 적은 게이트 수로도 고속 처리가 가능한 하드웨어 시스템이 된다.

5. 개선된 U-NET(Resblock)을 이용한 소프트웨어 후처리

제안한 구조의 Bicubic Interpolation 알고리즘은 기존 Software 방식과 유사한 성능을 보인다. 그러나 알고리즘의 한계로 인해 수준 높은 해상력은 얻기 힘들다. 이에 따라 제안한 구조를 Deep learning-소프트웨어를 통한 후처리를 하여 뛰어난 해상력 얻을 수 있다. 차후 이를 통합한 하드웨어 설계나 Hw/Sw Co-design을 위한 선행 연구가 이루어 질수 있도록 제안한다.

CNN 모델을 이용하여 학습용 데이터를 Convolution Layer마다 필터와 Convolution을 거쳐 출력 데이터를 얻는다. 이후 이 데이터를 다음과 같이 학습시킨다.

- 1. 학습 데이터를 원본 데이터(label) 와 비교 학습을 시켜 필터값을 얻는다.
- 2. 필터의 가중치와 편향(Bias) 값을 Gradient descent 알고리즘과 Adam optimizer를 통해 학습시켜 개선된 필터를 얻는다.
- 3. 학습이 끝난 후 입력 영상을 필터를 씌워 출력 데이터를 얻는다.

* 학습데이터는 800장의 이미지로 구성된 "DIV2K Train Data Set" 을 제안한 시스템 알고리즘으로 고속 처리하여 얻었다.

여러 CNN 모델 중 U-Net Mode2)을 다방면으로 학습 시켜 성능이 향상된 모델로 개선했다. batc size는 16, 초기 learning rate 는 1e-4를 사용하였으며, 이미지의 랜덤한 64 pixel을 데이터로 넣어서 학습시켰다. 그림 7은 개선한 U-Net(Resblock)의 구조로, 크게 Encoder, Bridge, Decoder 로 구성된다. 이때 Cont1~ Cont4에 해당하는 Encoder 와 Exp1 ~ Exp4에 해당하는 Decoder에 Resblock을 추가 하였다. 이로 인해 역전파(Back Propagation) 과정을 수월하게 해주어 기존의 U-Net 모델 보다 학습과정에 이점을 가진다.

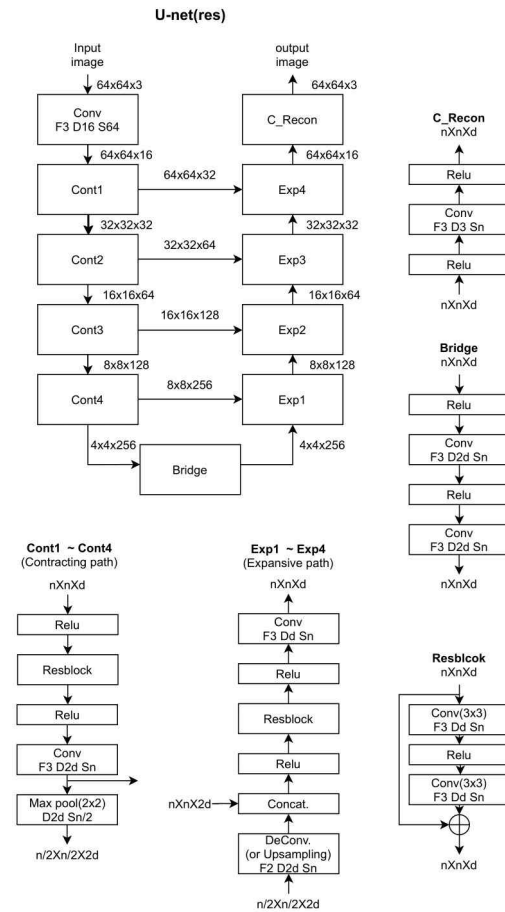


그림 7. U-Net(Resblock)구조

표 6은 제안한 시스템과 U-Net(Resblock)을 적용한 결과 분석이다. 실험 데이터는 총 24장의 영상으로 이루어진 “Kodak 24”를 사용했다. PSNR 분석은 다음과 같이 이루어졌다.

1. 실험 데이터를 4배 축소 뒤 제안한 시스템으로 처리하여 4배 보간 한다. 실험 데이터와 시스템의 결과물을 PSNR 분석 한다.
2. 시스템의 결과를 U-Net(Resblock)을 이용하여 후처리를 한다. 실험 데이터와 후처리 결과물을 PSNR 분석 한다.

제안한 시스템의 결과의 PSNR 평균값은 약 25.42dB 이었으나 이에 U-Net(Resblock)을 적용한 결과는 26.72dB로 약 1.3dB의 성능 향상을 보였다. 3번, 23번의 영상은 각각 0.54dB, 0.25dB 만큼 향상이 이루어 졌는데, 육안으로도 사물의 모서리를 포함한 영상의 전반적인 선명도가 올라갔음을 볼 수 있다.

표 6. Kodak 24 Dataset Average PSNR 분석

Kodak	제안한 시스템	U-Net(Resblock) 적용
Avg.PSNR	25.42147 dB	26.72355 dB



* “Kodak 24”의 3번, 23번 영상 처리 결과

6. 결론

제안하는 실시간 Bicubic Interpolation 시스템은 기존 Software 방식과 근사한 성능을 보이며 여러 영상 크기에 도 우수한 처리속도를 보인다. 병렬 처리를 적용한 8개의 PE 연산 프로세서와 번갈아 입출력의 역할을 하는 Tick, Tock 레지스터 구현 등으로 이러한 성능을 보여 준다. 동부하이텍 110nm 라이브러리로 합성하여 최대 동작 주파 수 510MHz 를 보이며 14K의 게이트를 사용한다. 또한 Gray, RGB 영상 모두 처리 가능하며, “Kodak 24” 데이터 기준 평균 PSNR 25.42dB의 성능을 보인다. 나아가 기존 딥러닝 U-Net을 개선하여 소프트웨어적으로 후처리를 적용한 결과 평균 1.3dB의 향상을 보인다.

참고 문헌

- 1) Yunshan Zhang, Yuhui Li*, Jie Zhen, Jionghao Li, Ran Xie The Hardware Realization of the Bicubic Interpolation Enlargement Algorithm Based on FPGA
- 2) Ronneberger, O., Fischer, P., & Brox, T. (2015, October). U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention* (pp. 234-241). Springer, Cham.