Asia-pacific Journal of Convergent Research Interchange
Vol.10, No.3, March 31 (2024), pp.1-10
http://dx.doi.org/10.47116/apjcri.2024.03.01

High-Level Synthesis Method of Lenet5 for FPGA Implementation

Lenet5의 FPGA 구현을 위한 High-Level Synthesis 방법

Man Soo Han[1]

한만수

Professor, Dept. of Information and Communications, Mokpo National University, Republic of Korea, mshan@mnu.ac.kr

Abstract: This paper introduces a method of implementing Lenet5, the best-known convolutional neural network, on FPGA (field programmable gate arrays) using HLS (high-level synthesis) tools. HDL (hardware description language) and HLS tools are mainly used in the FPGA implementation of Lenet5. HLS is implemented directly from C code to FPGA, so the development period is short, but compared to HDL, there are limits to the optimization that users can do. In this paper, to improve the operation speed of Lenet5, the memory was duplicated, and the for loop statement was unrolled to execute the execution statements within the loop simultaneously and in parallel without repeating them. To loop unroll the for loop statement, we introduce a method of implementing one-dimensional array variables as registers and dividing two-dimensional or more array variables into multiple memories with the same number of loop statement repetitions. In addition, to improve the access speed of some array variables, we introduce a method of designating a 2-port memory that can be accessed twice within one clock period. The method proposed in this paper is actually implemented on an FPGA, and its performance and pros and cons are compared with existing HDL implementations and HLS methods.

Keywords: Lenet5, FPGA, Memory Duplication, Parallel Processing, HLS

요약: 본 논문에서는 합성곱 신경망중에서 가장 잘 알려진 Lenet5를 HLS (high-level synthesis) 툴을 사용하여 FPGA (field programmable gate arrays)로 구현하는 방법을 소개한다. Lenet5의 FPGA 구현에는 HDL (hardware description language) 툴과 HLS 툴이 주로 사용된다. HLS는 C 코드에서 바로 FPGA로 구현되므로 개발기간이 짧지만 HDL에 비해 사용자가 할 수 있는 최적화에 한계가 있다. 본 논문에서는 Lenet5의 동작속도 향상을 위해 반복문 내의 실행문을 반복 실행하지 않고 병렬로 동시에 실행하기 위해 메모리를 복제하고 for loop문을 unrolling 하였다. For loop문을 loop unrolling하기 위해, 1차원 배열변수는 레지스터로 구현하고, 2차원 이상의 배열변수는 loop 문 반복횟수와 동일한 개수로 메모리를 여러 개로 분리하는 방법을 소개한다. 또한 일부 배열변수의 access 속도 향상을 위해 1 클록내에서 2회 access 할 수 있는 2 ports 메모리로 지정을 하는 방법을 소개한다. 본 논문에서 제안한 방법을 FPGA에 실제로 구현하고 기존의 HDL 구현 및 HLS 방법들과 성능 및 장단점을 비교한다.

핵심어: Lenet5, FPGA, 메모리복제, 병렬처리, HLS

Received: November 03, 2023; 1st Review Result: December 08, 2023; Accepted: February 26, 2024

Copyright ⓒ 2024 KCTRS

1. 서론

딥러닝은 자동차, 헬스케어, 자연어처리, 이미지 처리 등의 응용에 가장 많이 사용되는 인공지능 기술이다. 특히 딥러닝 기술중에서도 합성곱 신경망 (CNN: convolutional neural network)은 높은 인식율을 가지고 있어서 영상인식 및 분류, 의료 이미지 분석 등과 같은 영상 처리에 가장 많이 사용되고 있다. 최근에는 실시간 영상 처리 및 보안성 제공을 위해 CNN을 서버나 별도의 원격 처리장치를 거치지 않고 단말 장치에서 직접 실행하는 방법에 대한 연구가 많이 수행되고 있다. 단말 장치에서 CNN을 직접 실행하는 경우에는 처리 속도와 더불어 비용, 전력 소비량등을 고려하여야 한다.

단말에서 CNN을 직접 실행하기 위한 대표적인 방법이 GPU (graphics processing unit) 사용과 FPGA (field programmable gate arrays) 사용이다. FPGA는 GPU와 달리 별도의 처리장치를 필요로 하지 않아서 저비용, 저전력 및 유연성을 장점으로 갖는다. FPGA에 CNN을 구현하기 위해서는 주로 HLS (high-level synthesis)와 HDL (hard description language) 구현의 두가지 방법이 사용되고 있다. HDL 구현은 Verilog HDL 또는 VHDL (very high-speed integrated circuit HDL)과 같은 구현용 HDL 언어를 사용해서 CNN을 직접 FPGA에 구현하는 방식이며 HLS는 구현용 HDL 언어를 사용하지 않고 일반적인 C, C++로 작성된 코드를 컴파일해서 CNN을 FPGA로 구현하는 방식이다.

FPGA 내부에는 플립플롭, LUT (look-up table), RAM (random access memory) 등의 자원이 제한적으로 포함되어 있다. HDL은 구현용 언어이므로 이러한 제한적인 자원들을 사용해서 CNN의 구현할 때 성능 및 사용자원량등에서 최적화가 용이하다. 반면 HLS는 C 또는 C++ 로 작성된 코드를 사용해서 FPGA로 구현하므로 사용자가 개입할 수 있는 부분이 상대적으로 제한적이다. 따라서 HDL에 비해 성능 및 사용자원등에서 최적화가 어렵다. 반면 HLS는 일반적인 C, C++ 언어를 사용하므로 개발에 소요되는 시간이 단축되는 장점이 있다. HDL에 사용되는 대표적인 툴로는 Xilinx사의 Vivado가 있으며 HLS에 사용되는 대표적인 툴에는 Xilinx사의 Vitis HLS가 있다.

본 논문에서는 대표적인 CNN인 Lenet5[1]를 고려한다. Lenet5는 weight, bias, node로 구성되며 각각의 값이 32비트 부동소숫점으로 표현된다. Lenet5를 HLS 또는 HDL을 사용해서 자원이 제한적인 FPGA를 사용해서 구현하기 위해 많은 연구들이 수행되었다. 대부분의 연구들에서는 자원 사용량을 줄이기 위해 부동소숫점 데이터를 고정소숫점 데이터로 변환하는 방법을 사용하였다. [2]에서는 부동소숫점 데이터를 바로 사용해서 동작속도가 느리고 자원을 많이 사용하였다. [3]에서는 weight, bias에 16 비트 고정소숫점 데이터를 사용하였다. [4]에서는 weight 18 비트 및 node 16 비트, [5]에서는 11 비트, [6]에서는 24 비트, [7]에서는 weight에 8비트, node에 12비트, [8]에서는 16비트 고정소숫점 데이터를 사용하였다. 이들 논문들은 구현 방법으로 HLS를 사용하였으며 대부분 HLS 툴에서 일반적으로 사용되는 loop unrolling 및 파이프라인 등의 설계 방법을 적용하여 성능 개선을 시도하였다.

HDL을 사용하여 Lenet5를 구현한 방법으로는 [1][9][10] 등이 있다. [9]에서는 weight와 node에 각각 16 비트 고정소숫점 데이터를 사용하고 동작속도 향상을 위해 파이프라인 방법을 적용하였다. [10]에서는 weight 및 node에 7비트 이하의 고정소숫점 데이터를 사용하였다. [10]는 Lenet5 이외의 다른 CNN의 구현에 공통으로 적용할 수 있는 방법을 제안하였다. 따라서 Lenet5의 구현에 적합한 최적화가 부족해서 구현결과에서는 DSP (digital signal processer)와 같은 FPGA의 자원이 많이 사용되었다. [1]에서는 weight의

 Copyright ⓒ 2024 KCTRS

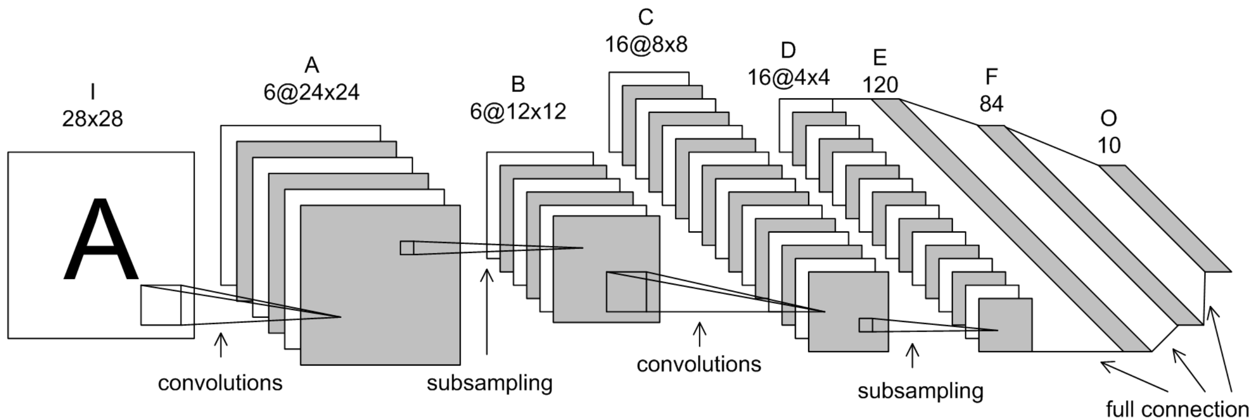
비트수가 7, 6, 5인 세가지 경우에 대해서 구현을 하였다. 성능 향상을 위해 메모리 복제와 loop unrolling을 같이 사용하는 방법을 제안하였다.

본 논문에서는 [1]에서 제안된 메모리 복제와 loop unrolling을 같이 사용하는 방법을 HDL이 아닌 HLS를 사용하여 구현한다. 구현 툴로는 Xilinx Vitis HLS를 사용하며 weight의 비트수가 7, 6, 5인 각각의 경우에 대한 구현 결과를 hit ratio, FPGA 자원 사용량, 동작속도 측면에서 [1]의 결과와 비교한다. 또한 기존 방법들과도 동작속도, hit ratio, FPGA 자원 사용량 등을 비교하여 HLS의 구현 방법이 HDL의 구현 방법에 비해서 갖는 장점 및 단점을 보인다.

본 논문은 다음과 같이 구성된다. 2절에서는 제안한 HLS 구현 방법을 소개하고 3절에서는 제안한 방법을 FPGA로 구현한 결과 및 기존 방법들의 성능과 비교한 결과를 소개한다. 4절에서는 결론 및 추후과제를 제시한다.

2. Lelet5의 HLS 구현

2.1 Lenet5 구조



[그림 1] Lenet5의 구조

[Fig. 1] Structure of Lenet5

[그림 1]은 본 논문에서 고려하는 Lenet5의 구조를 나타낸다. 노드 I는 입력 노드이며 본 논문에서는 28x28 픽셀 이미지를 사용한다. 각 픽셀은 8비트 값을 갖는다. 노드 A와 C는 각각 convolution 연산, 노드 B와 D는 subsampling 연산, 노드 E, F 및 O는 full connection 연산으로 노드값이 계산된다. Convolution 연산 및 full connection 연산에서 노드의 activation 함수로는 ReLu 함수를 사용하였다. 노드 O는 activation 함수로 softmax 함수가 사용되지만 본 논문에서는 학습이 종료된 Lenet5를 구현하므로 softmax 함수를 사용하지 않고 노드값을 직접 사용하여 대소비교를 하였다.

[표 1]에 Lenet5의 각 단계별로 node, weight 및 bias의 개수를 보였다.

Copyright ⓒ 2024 KCTRS

[표 1] Node, weight 및 bias 개수

[Table 1] Number of Nodes, Weight, Biases

Step	Input node	Weight	Bias	Output node
1	28x28	6x5x5	6	6x24x24
2	6x24x24			6x12x12
3	6x12x12	16x6x5x5	16	16x8x8
4	16x8x8			16x4x4
5	16x4x4	16x4x4x120	120	120
6	120	120x84	84	84
7	84	84x10	10	10

본 논문에서는 node, weight 및 bias 값을 고정소숫점으로 나타낼 때 사용하는 비트수에 대해서 [1]에서 제시된 3가지 경우를 고려한다. [1]에서는 MNIST 테스트 데이터로 hit ratio가 99.14%가 나오는 Lenet5의 weight, bias 및 node 데이터를 3가지 경우로 quantization 하였다. 32비트 부동소숫점 값을 고정소숫점으로 quantization을 하게 되면 일반적으로 hit ratio의 저하가 발생된다. Case 3은 weight 비트수가 5인 경우로 비트수가 충분하지 않아 hit ratio가 97.99%로 저하되어서 재학습을 해서 hit ratio를 98.54%로 향상시켰다. [표 2]에 각 경우에 대해 node, weight 및 bias 별 비트수와 hit ratio를 보였다.

[표 2] Weight, bias, node의 비트수와 hit ratio

[Table 2] Hit Ratios and Number of Bits for Weight, Bias and Node

Case	Weight, bias (bits)	A, B, C, D, E, F (bits)	O (bits)	Hit ratio (%)
1	7	10	13	98.97
2	6	9	11	98.89
3	5	7	9	98.54

2.2 HLS 구현

[그림 2]는 노드 A 값을 계산을 나타내는 C 코드이다. I는 입력 노드, w0는 weight, b0는 bias를 나타낸다. 노드 A값을 계산하기 위해서는 //c1 부분을 6*24*24*5*5 = 86,400회 반복해야 한다. //div 부분은 입력 노드 I를 256으로 나누기 위한 코드이며 이부분을 6*24*24 = 3,456번 반복해야 한다. 이러한 반복 횟수를 줄이기 위해 [1]에서는 메모리 복제와 loop unrolling을 사용하고 이를 HDL 코드로 작성하였다. 본 논문에서는 HLS 툴을 사용하여 메모리 복제와 loop unrolling을 적용한다.

Copyright ⓒ 2024 KCTRS

```
for (i=0; i<=5; i++)
   for (j=0; j<=23; j++)
      for (k=0; k<=23; k++){
         t=0;
         for (m=0; m<=4; m++)
            for (n=0; n<=4; n++)
               t += I[j+m][k+n]*w0[i][m][n];    //c1
         A[i][j][k] = t/256 + b0[i];              //div
      } //for_k
```

[그림 2] 노드 A 계산 코드

[Fig. 2] Calculation Code of Node A

```
for (j=0; j<=5; j++){
   j0=j, j1=j+6, j2=j+12, j3=j+18;
   for (k=0; k<=23; k++){
      for (i=0; i<=5; i++){   // for1
#pragma HLS UNROLL
         t0[i]=t1[i]=t2[i]=t3[i]=0;
      } //for_i
      for (m=0; m<=4; m++){
         for (n=0; n<=4; n++){
            for (i=0; i<=5; i++){   // for2
#pragma HLS UNROLL
               t0[i] += Ia[j0+m][k+n]*w0[i][m][n];
               t1[i] += Ia[j1+m][k+n]*w0[i][m][n];
               t2[i] += Ib[j2+m][k+n]*w0[i][m][n];
               t3[i] += Ib[j3+m][k+n]*w0[i][m][n];
            } //for_i
         }} //for_n, for_m
         for (i=0; i<=5; i++){   //for3
#pragma HLS UNROLL
            A[i][j0][k] = t0[i]/256 + b0[i]; //a1
            A[i][j1][k] = t1[i]/256 + b0[i]; //a2
            A[i][j2][k] = t2[i]/256 + b0[i]; //a3
            A[i][j3][k] = t3[i]/256 + b0[i]; //a4
         } //for_i
}} //for_k, for_j
```

[그림 3] 노드 A 의 병렬 계산 코드

[Fig. 3] Parallel Calculation Code of Node A

HLS 지시어를 사용하여 입력 노드 I를 Ia와 Ib로 복제하고 loop를 unrolling한 코드를 [그림 3]에 보였다. #pragma HLS UNROLL 지시어는 for loop 문을 unrolling하여 코드를 반복 실행하지 않고 병렬 동시 실행을 하라는 지시어이다. 예로 //for1의 t0[i]=t1[i]=t2[i]=t3[i]=0; 코드는 i=0,..,5까지 6번 반복 실행하지 않고 t0[0]=t1[0]=t2[0]=t3[0]=0; …, t0[5]=t1[5]=t2[5]=t3[5]=0; 와 같이 동시에 병렬로 실행된다. 이러한 병렬 실행을 위해서는 t0[i], t1[i], t2[i], t3[i]는 1 클록에서 read 또는 write 횟수가 1로 제한되는 RAM으로 구현되면 안되고 1 클록에서 read 및 write 횟수가 제한이 없는 레지스터로 구현되어야 한다. 배열 변수인 t0[]를 레지스터로 구현하는 HLS 지시어는 다음과 같다.

```
#pragma HLS ARRAY_PARTITION dim=1 type=complete variable=t0
```

Copyright ⓒ 2024 KCTRS

나머지 배열 변수인 t1[], t2[], t3[] 도 비슷하게 레지스터로 구현하였다. 또한 //for2 부분에서 w0[i][m][n] 배열 변수는 w00[m][n], w01[m][n], …, w05[m][n]과 같이 별도의 6개의 변수로 분리하고 각 분리된 변수를 RAM으로 구현하였다. 이를 위한 HLS 지시어는 다음과 같다. w0[][][]는 3차원 배열이고 첫번째 차원만 분리하므로 (즉, dim = 1) 6개의 변수로 분리가 된다.

```
#pragma HLS ARRAY_PARTITION dim=1 type=complete variable=w0
```

배열변수 Ia[][], Ib[][]는 동일 클록내에서 2번 read 동작을 해야하므로 2 ports RAM으로 구현하였다. 이를 위한 HLS 지시어는 다음과 같다.

```
#pragma HLS BIND_STORAGE variable=Ia type=ram_t2p
```

//for3 부분에서도 배열변수 A[][][]를 6개의 배열변수 A0[][], A1[][], …, A5[][]로 분리하였다. Ai[][], i=0,…,5는 동일 클록내에서 2번 write가 실행될 수 있도록 2 ports RAM으로 구현하였다. 또한 b0[] 배열변수도 레지스터로 구현하였다. 이를 위한 HLS 지시어는 각각 다음과 같다.

```
#pragma HLS ARRAY_PARTITION dim=1 type=complete variable=A

#pragma HLS BIND_STORAGE variable=A type=ram_t2p

#pragma HLS ARRAY_PARTITION dim=1 type=complete variable=b0
```

다른 노드값의 계산에서도 메모리 복제와 loop unrolling을 동시에 적용하였다. 메모리 복제를 노드 B에도 적용하였으며 노드 I의 경우와 비슷하게 Ba, Bb 두개로 복제하였다. 병렬실행을 위해 배열변수들은 1차원 배열의 경우 레지스터로 구현하고 다차원 배열변수들은 loop 실행 횟수 만큼의 독립된 배열변수들로 분리하였다. [표 3]에 각 단계별 구현 설정을 보였다.

[표 3] 구현 설정

[Table 3] Implementation Options

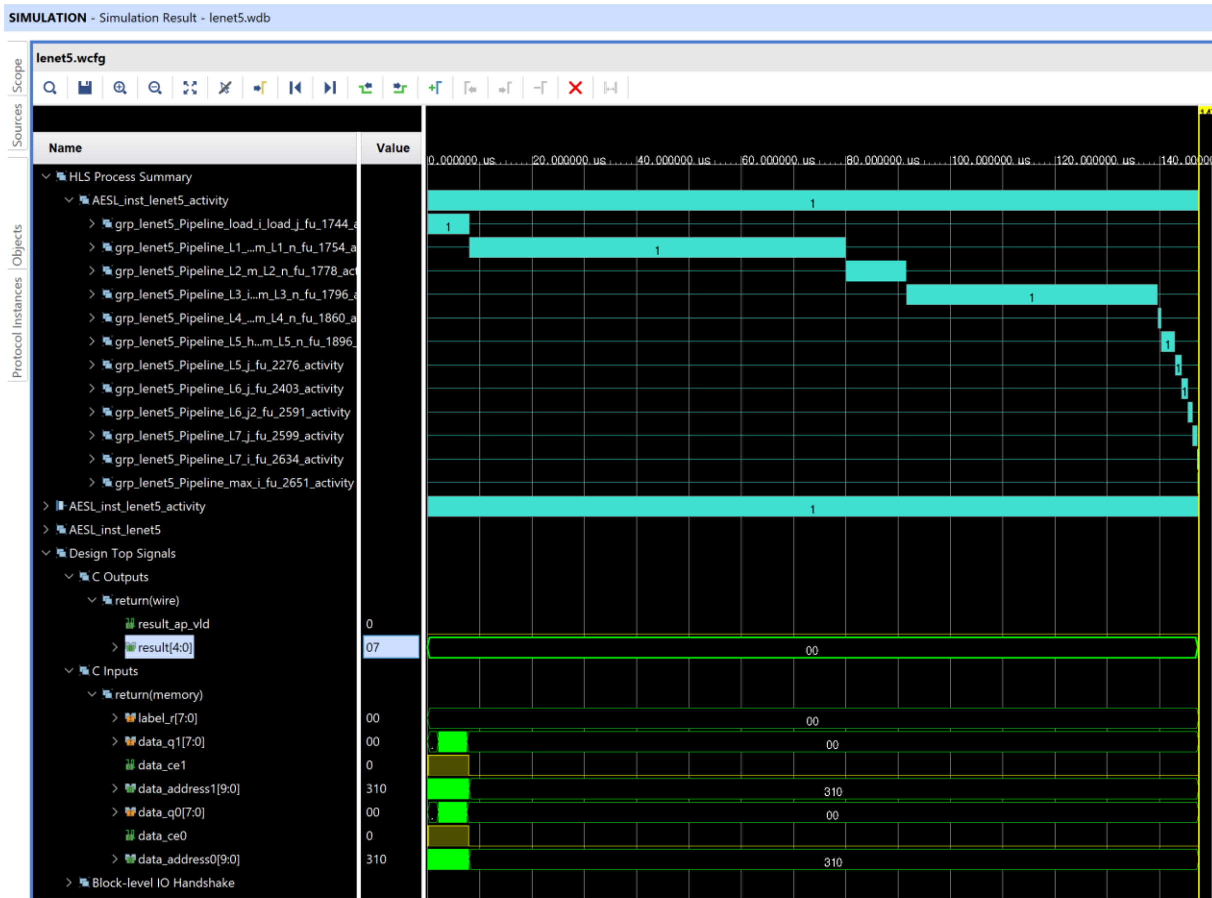
Step	Implementation options
1	I: memory duplication with Ia and Ib Ia, Ib: 2 ports RAM A: separated into 6 array variables, 2 ports RAM w0: separated into 6 array variables b0: register
2	B: memory duplication with Ba and Bb Ba, Bb: separated into 6 array variables, 2 ports RAM
3	C: separated into 16 array variables, 2 ports RAM w2: separated into 16 array variables b2: register
4	D: separated into 16 array variables
5	w4: separated into 120 array variables
6	w5: separated into 84 array variables
7	w6: separated into 10 array variables

 Copyright ⓒ 2024 KCTRS

3. 구현 결과 및 분석

본 논문에서는 구현 툴로 Xilinx사의 Vitis HLS 2023.2를 사용하였으며 FPGA는 [1]과 동일한 Xilinx사의 저가형 FPGA인 xc7a200tsbg484-3를 사용하였다. Vitis HLS 2023.2 툴은 loop의 반복횟수가 64이하면 파이프라인 동작으로 합성하도록 기본 설정되어 있다. 본 논문의 Lenet5 경우에는 테스트 결과 파이프라인 동작은 구현된 로직이 복잡하고 동작속도가 느려서 적합하지 않았다. 본 논문에서는 Vitis HLS 2023.2 툴의 파이프라인 동작을 사용하지 않고 본 논문의 2절에서 제안한 최적화 방법을 사용해서 합성하였다. Lenet5의 합성 결과를 MNIST 테스트 데이터로 검증하여 원래의 C 코드와 합성된 결과가 같음을 확인 하였다.

[그림 4]는 경우3일 때 합성후의 Lenet5의 1회 동작에 대한 시뮬레이션 결과를 나타낸다. 시뮬레이션시에 사용한 클록은 10ns이다. I행렬 변수에 테스트 이미지를 로드하는데 786 클록, 단계1부터 단계7까지 각각, 7207, 1159, 4809, 67, 382, 209, 109 클록이 소요되고 기타동작에 필요한 5 클록까지 포함하여 총 14,733 클록이 사용되었다.



[그림 4] 시뮬레이션 결과

[Fig. 4] Simulation Results

[표 4]는 각 경우에 대해서 동작속도를 HDL 구현 방법을 사용한 [1]과 비교한 결과이다. 최소 클록 주기는 회로가 동작 가능한 최소 클록 주기를 의미하며 최소동작

시간은 Lenet5가 전체 동작을 1회 완료하는데 걸리는 시간을 의미한다. [1]의 경우 사용자가 HDL 언어를 사용하여 직접구현을 하므로 최소 클록 주기와 최소동작시간이 HLS 툴을 사용하여 C코드에서 합성하는 경우보다 좋으며 이는 잘 알려진 사실이다 [11]. [표 4]에 나타난 바와 같이 동작시간은 본 논문의 HLS를 사용한 방법이 [1]에 비해 약 2배 정도이다.

[표 4] 최소 클록 주기 및 동작시간

[Table 4] Minimum Clock Period and Minimum Operation Time

Case	Minimum clock period		Minimum operation time	
	[1]	Proposed	[1]	Proposed
1	9.2ns (108.6MHz)	9.74ns (102.7MHz)	0.0791ms	0.147ms
2	8.5ns (117.6MHz)	9.57ns (104.5MHz)	0.0731ms	0.144ms
3	8ns (125MHz)	9.11ns (109.7MHz)	0.0688ms	0.137ms

[표 5]는 구현에 사용된 FPGA 자원들에 대해서 본 논문의 방법을 [1]과 비교하였다. LUT는 lookup table, LUTRAM은 LUT에 포함된 RAM, FF는 플립플롭, BRAM은 block RAM을 의미한다. 제안한 방법은 DSP, BRAM, FF은 [1]에 비해 많이 사용하였지만 LUTRAM과 LUT는 적게 사용하였다. 이는 HLS 합성시에 툴이 주로 DSP를 사용하여 곱셈과 덧셈을 구현한 결과이다. [11]에 알려진 바와 같이 HLS에서는 사용자가 HDL을 사용할 때와 같은 정도로 합성 최적화과정에 관여를 할 수 없고 HLS에서는 사용자의 지시어가 합성과정에서 지켜지지 않는 경우도 발생하는 등의 다양한 제약 조건이 있어서 최적화에 한계가 있어서 발생된 결과이다.

[표 5] FPGA 구현 utilization

[Table 5] FPGA Implementation Utilization

Resource	Case 1		Case 2		Case 3		Available
	[1]	Proposed	[1]	Proposed	[1]	Proposed	
LUT	18845	7830	18747	7186	19245	6954	133800
LUTRAM	173	0	155	0	485	0	46200
FF	6586	8665	5831	7630	4836	5911	269200
BRAM	84	174	78.5	174	35.5	85	365
DSP	149	302	99	287	0	272	740

[표 6]은 다른 논문들의 합성결과와 본 논문의 경우 3의 합성결과를 비교한 방법이다. 본 논문의 결과는 HLS 툴을 사용한 [7]과 [8]의 결과에 비해서 동작시간 (timing)에서 우수함을 알 수 있다. 또한 HDL 툴을 사용한 [3]과 [9]의 결과와 비교해서도 동작시간에서 우수함을 알 수 있다. 그 이유로는 [3]은 고정소숫점 데이터를 부동소숫점 데이터로 변환해서 연산 후 다시 부동소숫점 데이터로 저장하는 과정을 거치며 [9]에서는

Copyright ⓒ 2024 KCTRS

파이프라인 방식으로 최적화를 했기 때문이다. 본 논문에서 적용한 메모리 복제와 loop unrolling 방법이 보다 효율적임을 알 수 있다.

[표 6] 구현결과 비교

[Table 6] Comparison of Implementation Results

	[3]	[9]	[7]	[8]	[1]	Proposed
FPGA	XCZU9EG	xc7a200tsbg484	XCZU9EG-2ffvb1156	XC7Z020-CLG484-1	xc7a200tsbg484-3	xc7a200tsbg484-3
Tool	HDL	HDL	HLS	HLS	HDL	HLS
Clock	150MHz	75MHz	100MHz	Unknown	125MHz	109.7MHz
Precision	16bits floating	16bits fixed	18/12bits fixed	16bits fixed	5/7/9bits fixed	5/7/9bits fixed
Timing	0.162ms	0.526ms	4.631ms	1.07ms	0.0688ms	0.137ms
Hit ratio	99.11%	unknown	98.40%	99.01%	98.54%	98.54%

4. 결론

본 논문에서는 Lenet5의 FPGA 구현을 위해 HLS 툴을 사용해서 메모리 복제와 loop unrolling을 적용하는 방법을 소개하였다. 노드값의 계산 속도를 향상시키기 위해 입력 노드값을 저장하는 메모리를 복제하고 loop unrolling으로 병렬 실행을 하였다. 병렬 실행을 위해 1차 배열 변수들은 레지스터로 구현하고 다차원 배열 변수들은 loop의 반복횟수와 동일한 개수의 배열 변수로 분리하였다. 제안한 방법을 Vitis HLS 2023.2 툴을 사용하여 FPGA에 구현 하였으며 HDL 언어를 사용해서 메모리 복제와 loop unrolling을 적용해서 Lenet5를 직접 FPGA에 구현한 방법과 비교하였다. 비교 결과 동작속도와 자원 사용량 측면에서 HDL 언어를 사용한 경우가 우수하였다. 그러나 기존의 HLS를 사용한 다른 논문들의 결과에 비해서는 본 논문의 방법이 우수함을 보였다. 추후 연구로는 Lenet5의 동작속도를 향상시키는 HLS의 최적화방법을 연구할 계획이다.

References

[1] M. S. Han, Implementation of Convolutional Neural Networks using Low-cost FPGAs, Journal of Korean Institute of Intelligent Systems, (2023), Vol.33, No.4, pp.309-319.
DOI: 10.5391/JKIIS.2023.33.4.309

[2] Y. Hou and Z. B. Chen, LeNet-5 improvement based on FPGA acceleration, The Journal of Engineering, (2020), Vol.2020, No.13, pp.526-528.
DOI: 10.1049/joe.2019.1190

[3] Y. Shi, T. Gan, S. Jiang, Design of Parallel Acceleration Method of Convolutional Neural Network Based on FPGA, IEEE 5th International Conference on Cloud Computing and Big Data Analytics (ICCCBDA), IEEE, (2020)
DOI: 10.1109/ICCCBDA49378.2020.9095722

[4] S. Zhai, C. Qiu, Y. Yang, and J. Li, Design of Convolutional Neural Network Based on FPGA, Journal of Physics Conference Series, (2019), Vol.1168, No.6.
DOI: 10.1088/1742-6596/1168/6/062016

Copyright ⓒ 2024 KCTRS

[5] Y. Zhou and J. Jiang, An FPGA-based Accelerator Implementation for Deep Convolutional Neural Networks, 4th International Conference on Computer Science and Network Technology (ICCSNT), IEEE, (2015)
DOI: 10.1109/ICCSNT.2015.7490869

[6] G. Feng, Z. Hu, S. Chen and F. Wu, Energy-Efficient and High-Throughput FPGA-based Accelerator for Convolutional Neural Networks, 13th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT), IEEE, (2016)
DOI: 10.1109/ICSICT.2016.7998996

[7] M. H. Cho and Y. M. Kim, FPGA-Based Convolutional Neural Network Accelerator with Resource-Optimized Approximate Multiply-Accumulate Unit, Electronics, (2021), Vol.10, No.22, 2859.
DOI: 10.3390/electronics10222859

[8] Y. Liang, J. Tan, Z. Xie, Z. Chen, D. Lin, and Z. Yang, Research on Convolutional Neural Network Inference Acceleration and Performance Optimization for Edge Intelligence, Sensors, (2024), Vol.24, No.1, 240.
DOI: 10.3390/s24010240

[9] D. Shan, G. Cong and W. Lu, A CNN Accelerator on FPGA with a Flexible Structure, 5th International Conference on Computational Intelligence and Applications (ICCIA), IEEE, (2020)
DOI: 10.1109/ICCIA49625.2020.00047

[10] T. Pacini, E. Rapuano, L. Fanucci, FPG-AI: A Technology-Independent Framework for the Automation of CNN Deployment on FPGAs, IEEE open access, (2023), Vol.11, pp.32759-32775.
DOI: 10.1109/ACCESS.2023.3263392

[11] L. Huang, DL. L, K. Wang. T. Gao, and A. Tavares, A survey on performance optimization of high-level synthesis tools, Journal of Computer Science and Technology, (2020), Vol.35, No.3, pp. 697-720.
DOI: 10.1007/s11390-020-9414-8

Copyright ⓒ 2024 KCTRS