2021-2학기 SoC 설계 방법론

Team 1 Final Project Report

주제: HLS를 이용한 Sparse Convolution 용 CSR 기반 Rule Generator 하드웨어 설계

2021.12.23

2020252244 이장환

2020298097 박성민

2019200648 이민재

목차

1 개요

1.1 프로젝트 배경

자율주행 기술에 사용되는 3D object detection은 **그림 1**과 같이 LiDAR 데이터로부터 Point cloud data를 수집하여 연산에 활용한다. 그러나 Point cloud data는 위치 정보가 다양하여 데이터 처리가 쉽지 않아 SECOND와 같은 모델에서는 Voxel이라는 grid 형태로 묶은 뒤, Convolution을 진행 하여 특징을 추출한다.

그러나 이러한 Voxel은 3D 공간 내에 0.098%만 존재할 정도로 매우 희박하게 존재하기 때문에 기존의 Dense convolution의 형태와 같이 sliding window 형식으로 연산하게 되면 매우 비효율적 이기 때문에, 유효한 voxel에 대해서만 연산하는 sparse convolution을 사용하게 된다.

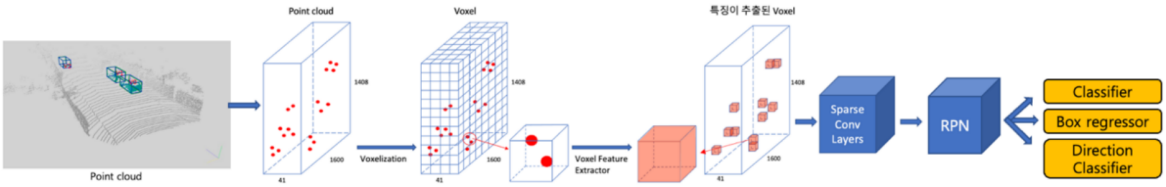


그림 1 3D 객체 검출 모델인 SECOND의 추론 과정

Sparse convolution을 연산하기 위해서는 input data와 partial sum data의 3D 공간 내의 위치를 기술해주는 Rule이라는 Look up table이 필요하게 된다. 기존의 CPU에서는 이를 구현하기 위해서 **그림 2**의 예시 코드처럼 hash function을 사용하여, 같은 3D 공간 내의 위치에서 accumulate되는 partial sum의 정보를 찾게 된다. 그러나 hash table은 임베디드 환경에서 구현이 어렵기 때문에 병목 현상으로 작동할 수 있다.

따라서 본 프로젝트에서는 Sparse encoding의 한 형태인 CSR format을 활용하여, hash table을 사용하지 않고 Rule을 만들 수 있는 방법을 제안하고 하드웨어로 구현하고자 한다.

```
for (int j = 0; j < numActIn; ++j) {
  batchIdx = indicesIn(j, 0);
  numValidPoints = getValidOutPosTranspose<Index, NDim>(
      indicesIn.data() + j * (NDim + 1) + 1, kernelSize, stride, padding,
      dilation, outSpatialShape, validPoints);
  for (Index i = 0; i < numValidPoints; ++i) {
    pointPtr = validPoints + i * (NDim + 1);
    auto offset = pointPtr[NDim];
    auto index = tv::rowArrayIdx<Index, NDim>(pointPtr, outSpatialShape) +
                 spatialVolume * batchIdx;

    auto iter = hash.find(index);
    if (iter == hash.end()) {
      for (unsigned k = 1; k < NDim + 1; ++k) {
        indicesOut(numAct, k) = pointPtr[k - 1];
      }
      indicesOut(numAct, 0) = batchIdx;
      hashval = numAct++;
      hash[index] = hashval;
    }else{
      hashval = iter->second;
    }
    // indicePairs: [K, 2, L]
    indicePairs(offset, 0, indiceNum[offset]) = j;
    indicePairs(offset, 1, indiceNum[offset]++) = hashval;
  }
}
```

그림 2 기존의 CPU에서 Rule을 만들기 위한 연산 과정 (예시 코드)

1.2 프로젝트 목표

프로젝트의 결과물로는 CSR format을 이용한 Rule generator와 hash table을 이용한 Rule generator를 C++로 구현하고 HLS 최적화를 적용하여, 둘 사이의 latency 및 hardware resource를 비교하고자 한다. 이를 이해 먼저 C++을 통해 구현한 알고리즘의 Functionality를 검증하고, 검증된 코드를 바탕으로 HLS에 적용하여 하드웨어 구현을 진행한다.

HLS을 통해 하드웨어를 구현할 때에는 성능을 최대로 높이기 위해서 C++로 구현된 코드에 loop unrolling과 같은 최적화 방법을 적용해보고, 최적화가 가능한 부분과 최적화가 잘 적용되지 않는 부분을 분석하도록 한다. 또한 Point cloud를 사용하는 3D object detection 모델에 따라 2D Convolution에서 사용되는 데이터에 대해서 Rule을 잘 생성하는 것이 중요하기에 실제 3D object detection 모델인 PointPillars의 경우에 대해서 적용해보도록 한다.

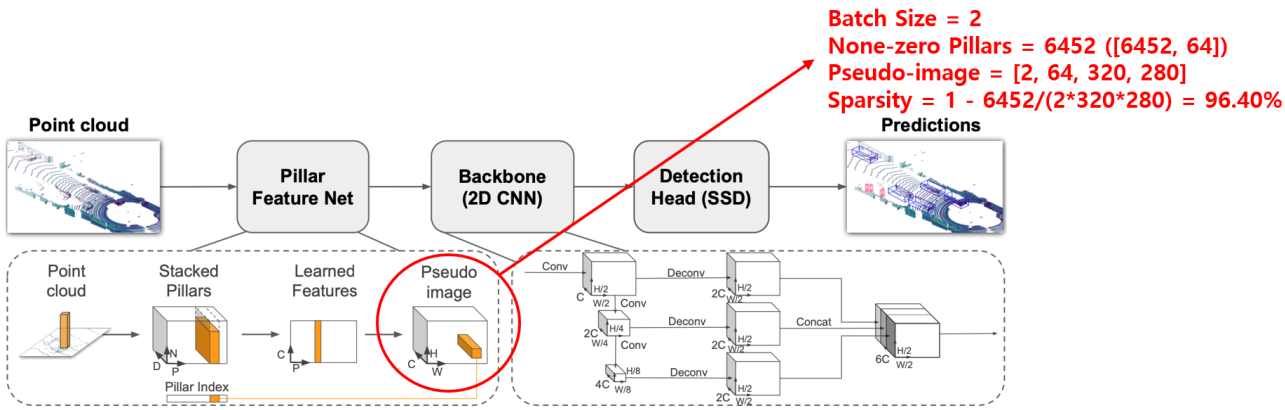


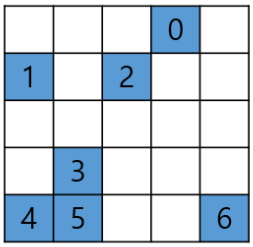
그림 3 PointPillars의 모델 및 해당 모델의 2D Convolution에서의 실제 데이터 Sparsity

그림3은 실제 PointPillars 모델에 대한 구조와 2D Convolution에서 사용되는 데이터의 Sparsity를 나타내고 있다. 그림에서 볼 수 있듯이 데이터의 Sparsity가 대략 95~96% 정도 되는 것을 알 수 있다. 최종적으로 CSR format을 이용한 Rule generator의 성능을 확인하기 위해, 해당 데이터 Sparse 특성을 고려하였다. 총 2가지 경우의 Input size를 고려하는데, 첫 번째는 32x32의 작은 크기를 가지는 input size에 4% Sparsity를 가지는 40개의 coordinate 좌표를 사용하고, 두 번째는 실제 PointPillars 모델의 320x280 크기의 Input size와 5%의 Sparsity를 가지는 4551개의 coordinate 좌표를 사용하였다.

이를 통해 최종적인 프로젝트의 목표로는 CSR format의 특성을 활용하여 Rule generator를 HLS를 통해 설계하는 것이 목표이다. 또한 설계한 하드웨어의 성능 분석을 통해 추가적인 성능 개선 가능한 부분을 파악하고 최대한 성능 개선을 이루고자한다.

2 Compressed Sparse Row (CSR) format을 이용한 Rule generator

2.1 CSR format 개념

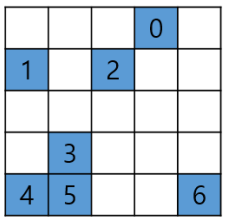


csr_row: [0, 1, 3, 3, 4, 7]
csr_col: [3, 0, 2, 1, 0, 1, 4]

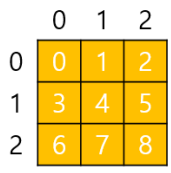
그림 4 CSR format을 활용하여 Sparse한 데이터를 표현하는 방법

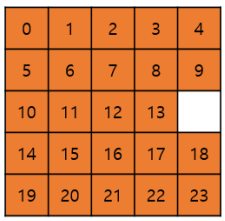
Compressed Sparse Row (CSR) format은 sparse한 데이터를 encoding 하는 방법 중 한 가지이다. 일반적으로 2D에 존재하는 데이터의 위치를 표현하는 방법은 Coordinate[x, y]로 나타내는 것이지만, CSR format은 CSR row, CSR column이라는 두가지 형태로 데이터를 나타낼 수 있다. CSR row는 각 row 당 존재하는 데이터의 개수를 표현하고 CSR column은 해당 row에서 어느 column index에 해당하는 위치에 존재하는지를 나타내게 된다.

예를 들어, **그림 4**의 예시의 경우, 첫번째 row에 존재하는 데이터의 수가 1개이므로, CSR row에 첫번째 row에 존재하는 데이터 개수를 1로 나타내게 된다. 이 때, CSR row의 가장 첫번째에는 0을 저장하여 첫번째 row에 존재하는 데이터 개수를 파악할 수 있도록 도와준다. 그 다음 row부터는 해당 row에 존재하는 개수만큼 더해서 다음 값에 저장하게 된다. 또한 Non-zero element가 있는 데이터 개수만큼 CSR col의 값이 저장되는데, 이는 해당 데이터가 어느 column index에 존재하는지 나타낼 수 있게 된다. 즉, 위에서부터 차례대로 첫번째 데이터는 4번째 두번째와 세번째 데이터는 각각 첫번째와 세번째에 존재하므로, **그림 4**와 같이 구해지는 것을 알 수 있다.



csr_row: [0, 1, 3, 3, 4, 7]
csr_col: [3, 0, 2, 1, 0, 1, 4]





Rule

Kernel 0	Kernel 1	Kernel 2	Kernel 3	Kernel 4	Kernel 5	Kernel 6	Kernel 7	Kernel 8
0 → 9	0 → 8	0 → 7	0 → 4	0 → 3	0 →2	1 →1	1 →0	2 →1
1 → 11	1 → 10	2 → 11	1 → 6	1→5	2 →6	2 →3	2 →2	3 →10
2 → 13	2 → 12	3 → 19	2 → 8	2→7	3 →14	3 →12	3 →11	5 →14
3 → 21	3 → 20		3 → 16	3→15	5 →19	4 →15	4 →14	6 →17
			4 → 20	4→19	6 →22	5 →16	5 →15	
			5 → 21	5→20			6 →18	
				6→23				

그림 5 Sparse한 데이터와 Kernel을 통해 생성되는 Rule의 결과

CSR format의 특성을 활용하면 **그림 5**와 같이 Rule generator를 생성할 수 있다. 이를 위해 본 프로젝트에서는 HLS를 사용하여 해당 하드웨어를 설계한다.

2.2 HLS를 이용한 CSR format 기반의 Rule generator 하드웨어 구현

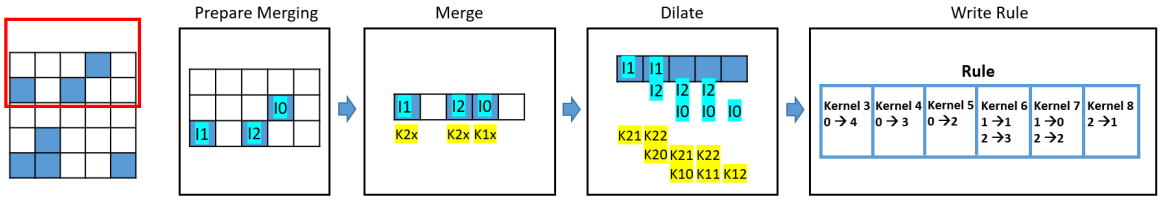


그림 6 CSR format 기반의 Rule generator 하드웨어 구조?

CSR format 기반의 Rule generator의 하드웨어의 구성은 **그림 6**과 같이 Merge row를 하기 위해 크게 1.Prepare Merging, 2.Merge, 3.Dilate, 4.Write Rule, 4가지로 구성된다. 1.Prepare Merging는 3x3 Convolution에 대해서 특정 Output row에 해당하는 데이터의 결과가 될 수 있는 3개의 Input row들을 추출하는 부분이 된다. 다음으로 2.Merge는 후보 군이 되는 3개의 row에 대해서 Merge했을 때의 결과를 만들어 주며, 실제 데이터가 존재하는 위치의 Input index와 Merge되어 나타나지 않는 Kernel의 Vertical index를 나타낼 수 있는 정보(Input-Kernel pair)를 만들어주게 된다. 3.Dilate에서는 Merged column을 통해서 3x3 Convolution에 의해 생성되는 Output의 모든 위치들을 계산해준다. 그렇게 구해진 Dilated column은 이전에 구했던 Input-Kernel pair에 해당하는 정보화 함께 4.Write Rule 부분으로 전달해주면, 4.Write Rule에서는 결과를 Rule에 작성하는 것으로 하나의 Output row에 대한 연산이 끝나게 된다. 각 부분에 해당하는 코드 구현은 Appendix에 있는 **그림 7~10**과 같다.

Prepare Merging, Merge와 Dilate 모든 부분들이 Sparse하게 존재하는 데이터의 개수에 따라서 언제 끝날지 서로 다르기 때문에, 각 부분에서 연산이 언제 끝날지 모른다는 문제가 존재한다. 따라서, while(!condition) 혹은 break와 같은 것을 사용해서 구현 했었는데, 이러한 경우에는 '**#pragma HLS UNROLL**'을 적용하는 것이 쉽지 않았다. 처음에 각 부분에 '**#pragma HLS UNROLL**'을 적용하였더니, **그림 7**과 같이 Merge Loop에서 엄청나게 많은 Clock cycle이 소요되는 비현실적인 결과가 나오게 되었다.

```
        doneSignal = entry0[e0_counter] + entry1[e1_counter] + entry2[e2_counter];
        merge_loop: while(doneSignal != COL_SIZE*3){
#pragma HLS UNROLL
            lowestIdx = minIdx(entry0[e0_counter],entry1[e1_counter],entry2[e2_counter]);
            // can be parallelized
            if ((lowestIdx&1)==1){
                kernelIdx = 0;
                inputIdx = e0_counter + csr_row[r-1];
                lowestVal = entry0[e0_counter];
                e0_counter++;
                input_index[kernelIdx][merge_counter] = inputIdx;
            }
            if ((lowestIdx&2)>>1==1){
                kernelIdx = 1;
                inputIdx = e1_counter + csr_row[r];
                lowestVal = entry1[e1_counter];
                e1_counter++;
                input_index[kernelIdx][merge_counter] = inputIdx;
            }
            if ((lowestIdx&4)>>2==1){
                kernelIdx = 2;
                inputIdx = e2_counter + csr_row[r+1];
                lowestVal = entry2[e2_counter];
                e2_counter++;
                input_index[kernelIdx][merge_counter] = inputIdx;
            }
            merged_col[merge_counter] = lowestVal;
            // If no more value in next index -> skip
            doneSignal = entry0[e0_counter] + entry1[e1_counter] + entry2[e2_counter];
            merge_counter++;
        }
```

merge_loop	134	3	4294967252	4294966929

그림 7 Merge에 #pragma HLS UNROLL을 적용하였을 때, 비현실적인 loop cycle 결과

HLS에서 언제 끝날지 모르는 연산에 대한 UNROLL이 적용할 수 없었기 때문에, 최대로 소요되는 loop count(해당 코드에서는 'COL_SIZE'(**그림 8**))를 고려하여 고정적인 크기의 loop가 수행될 수 있도록 **그림 8**과 같이 for loop 형태로 코드를 수정하여 '**#pragma HLS UNROLL**'을 다시 적용해본 결과 **그림 9**의 결과와 같이 Unrolling이 적용되어 대략 **x4.78**만큼 빨라진 것을 확인할 수 있었다.

```
        merge_loop: for (int i=0; i<COL_SIZE; i++){
#pragma HLS UNROLL
            lowestIdx = minIdx(entry0[e0_counter],entry1[e1_counter],entry2[e2_counter]);
            // can be parallelized
            if ((lowestIdx&1)==1){
                kernelIdx = 0;
                inputIdx = e0_counter + csr_row[r-1];
                lowestVal = entry0[e0_counter];
                e0_counter++;
                input_index[kernelIdx][i] = inputIdx;
            }
            if ((lowestIdx&2)>>1==1){
                kernelIdx = 1;
                inputIdx = e1_counter + csr_row[r];
                lowestVal = entry1[e1_counter];
                e1_counter++;
                input_index[kernelIdx][i] = inputIdx;
            }
            if ((lowestIdx&4)>>2==1){
                kernelIdx = 2;
                inputIdx = e2_counter + csr_row[r+1];
                lowestVal = entry2[e2_counter];
                e2_counter++;
                input_index[kernelIdx][i] = inputIdx;
            }
            if (lowestVal==COL_SIZE){
                break;
            }
            merged_col[i] = lowestVal;
            // If no more value in next index -> skip
        } // end merge_loop
```

그림 8 Constant loop count에 대해 연산을 수행할 수 있도록 Merge 부분을 수정한 코드

Modules	Avg II	Max II	Min II	Avg Latency	Max Latency	Min Latency
csr				13009	13009	13009
csr_Pipeline_init_rule_cnt				9	9	9
csr_Pipeline_init_out_csr_row				33	33	33
csr_Pipeline_cal_num_feature_per_row				32	32	32
csr_Pipeline_init_entry	405	458	374	32	32	32
csr_Pipeline_prepare_merge_loop1_0				0	0	0
csr_Pipeline_prepare_merge_loop1_1				3	3	3
csr_Pipeline_prepare_merge_loop0_1				0	0	0
csr_Pipeline_prepare_merge_loop0_2				0	0	0
csr_Pipeline_prepare_merge_loop2_0	406	458	375	1	4	0
csr_Pipeline_prepare_merge_loop2_1	406	458	375	1	4	0
csr_Pipeline_prepare_merge_loop2_2	406	457	375	1	4	0
csr_Pipeline_init_merge	405	455	375	32	32	32
csr_Pipeline_merge_loop	405	455	375	14	34	4
csr_Pipeline_dilation_loop	435	1143	393	12	37	5
csr_Pipeline_write_rule_loop	406	458	375	292	292	292
row_loop	405	458	374	405	458	374

#pragma HLS UNROLL

Modules	Avg II	Max II	Min II	Avg Latency	Max Latency	Min Latency
csr				2720	2720	2720
csr_Pipeline_prepare_merge_loop1_0				0	0	0
csr_Pipeline_prepare_merge_loop1_1				3	3	3
csr_Pipeline_prepare_merge_loop0_1				0	0	0
csr_Pipeline_prepare_merge_loop0_2				0	0	0
csr_Pipeline_prepare_merge_loop2_0	85	137	55	1	4	0
csr_Pipeline_prepare_merge_loop2_1	85	137	55	1	4	0
csr_Pipeline_prepare_merge_loop2_2	85	136	55	1	4	0
minIdx						
csr_Pipeline_dilation_loop	91	182	72	12	37	5
row_loop	84	137	54	84	137	54

그림 9 #pragma HLS UNROLL을 적용하였을 때의 Latency 비교 결과

하지만 여전히 Merge의 loop을 분석해본 결과 (**그림 10**) Unrolling이 제대로 수행되지 않았는데, 그 이유는 Column index에 대해 select된 bit에서 'lowestVal'(**그림 8**)을 제거하고 entry counter를 증가시켜야 하는데, min 값이 어느 부분에서 나올지 몰라서 생기는 문제로 파악하였다. 즉 각 loop 별로 수행되어야 하는 연산은 이전에 loop count에서 실행되는 결과가 활용되어야하는 dependency가 존재하는 문제가 있다.

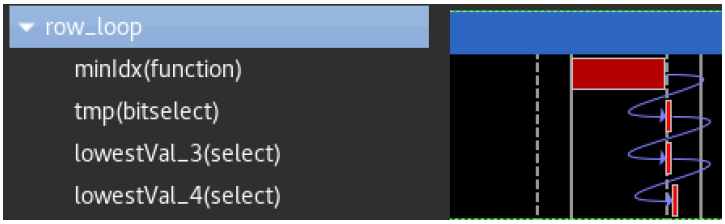


그림 10 Merge loop에 Unrolling을 적용했을 때 발생하는 문제

동일하게 Dilate 부분에 대해서 확인해본 결과, 그림 11의 'Dilation_subloop1'에서도 가변적인 loop count가 존재하여 Unrolling을 적용하기 어려웠다. 따라서 Unrolling을 적용하기 쉽지 않은 부분들이 존재하여 최적화를 위해서는 Dataflow 방법이 필요하다고 판단하였다. Dataflow에 대한 설명은 4장에서 자세히 설명하도록 하겠다. 마지막으로 2장의 내용을 정리하자면, '**#pragma HLS**'를 통해 최적화를 진행한 결과 중첩된 loop가 존재하더라도, CSR format 기반의 Rule generator의 특성상 각 loop간의 연산 되는 데이터의 dependency가 존재하여 정상적으로 적용되지 않는 것을 확인할 수 있었다.

```
dilation_loop: for (merged_col_counter=1; merged_col_counter < COL_SIZE-1; merged_col_counter++){
    if (merged_col[merged_col_counter]-merged_col[merged_col_counter-1] <= 3){ // update high only
        high = merged_col[merged_col_counter];
        if (merged_col[merged_col_counter+1]==COL_SIZE){
            if (debug==true){
                printf("[Dilation] Grouping done %d to %d\n",low,high);
                }
            dilation_subloop0: for (int i=max(0,low-1); i<=min(COL_SIZE-1,high+1); i++) {
                out_csr_col_per_row[nnz++] = i;
            }
            break;
        }
    } else {
        if (debug==true){
            printf("[Dilation] Grouping done %d to %d\n",low,high);
        }
        dilation_subloop1: for (int i=max(0,low-1); i<=min(COL_SIZE-1,high+1); i++) {
            out_csr_col_per_row[nnz++] = i;
        }
        low = merged_col[merged_col_counter];
        high = merged_col[merged_col_counter];
        if (merged_col[merged_col_counter+1]==COL_SIZE){
            if (debug==true){
                printf("[Dilation] Grouping done %d to %d\n",low,high);
            }
            dilation_subloop2: for (int i=max(0,low-1); i<=min(COL_SIZE-1,high+1); i++) {
                out_csr_col_per_row[nnz++] = i;
            }
            break;
        }
    }
} // end each column index
```

그림 11 Dilate 부분에 있는 가변적인 loop count

3 Functionality 검증

3.1 간단한 예제를 통한 Functionality 검증

Functionality 검증을 위해 **그림 5**에서 수행한 5x5 크기의 2 Dimension coordination 공간에 임의의 7개의 point의 데이터를 가지고 동일한 결과를 얻을 수 있는지 확인하기 위해 HLS로 구현한 Rule generator의 실행 결과를 출력하여 비교해 보았다. 2장에서 설명한 것과 같이 Merge → Dilate → Write Rule을 모든 Output row에 대해서 수행하면 최종적인 Rule의 Table을 얻을 수 있다. Rule generator를 통해 얻은 결과는 그림 12의 아래와 같고, 실제로 얻을 수 있는 그림 12의 위의 표와 비교해보면 동일하게 출력된 것을 확인할 수 있다. 이로써 Functionality에 대한 검증은 이것으로 증명하였으며, Github에 업로드한 코드를 실행하면 동일한 결과를 확인해볼 수 있다.

Kernel 0	Kernel 1	Kernel 2	Kernel 3	Kernel 4	Kernel 5	Kernel 6	Kernel 7	Kernel 8
0 → 9	0 → 8	0 → 7	0 → 4	0 → 3	0 →2	1 →1	1 →0	2 →1
1 → 11	1 → 10	2 → 11	1 → 6	1→5	2 →6	2 →3	2 →2	3 →10
2 → 13	2 → 12	3 → 19	2 → 8	2→7	3 →14	3 →12	3 →11	5 →14
3 → 21	3 → 20		3 → 16	3→15	5 →19	4 →15	4 →14	6 →17
			4 → 20	4→19	6 →22	5 →16	5 →15	
			5 → 21	5→20			6 →18	
				6→23				

Generated Rule

```
================ Summary ================

[Result] Output CSR generation done
Output CSR Row
0 5 10 14 19 24
Output CSR Col
0 1 2 3 4 0 1 2 3 4 0 1 2 3 0 1 2 3 4 0 1 2 3 4

[Result] Rule generation done
   Kernel  0      Kernel  1      Kernel  2      Kernel  3      Kernel  4      Kernel  5      Kernel  6      Kernel  7      Kernel  8
  Input Output   Input Output   Input Output   Input Output   Input Output   Input Output   Input Output   Input Output   Input Output
|     0    9  ||     0    8  ||     0    7  ||     0    4  ||     0    3  ||     0    2  ||     1    1  ||     1    0  ||     2    1  |
|     1   11  ||     1   10  ||     2   11  ||     1    6  ||     1    5  ||     2    6  ||     2    3  ||     2    2  ||     3   10  |
|     2   13  ||     2   12  ||     3   19  ||     2    8  ||     2    7  ||     3   14  ||     3   12  ||     3   11  ||     5   14  |
|     3   21  ||     3   20  ||    -1   -1  ||     3   16  ||     3   15  ||     5   19  ||     4   15  ||     4   14  ||     6   17  |
|    -1   -1  ||    -1   -1  ||    -1   -1  ||     4   20  ||     4   19  ||     6   22  ||     5   16  ||     5   15  ||    -1   -1  |
|    -1   -1  ||    -1   -1  ||    -1   -1  ||     5   21  ||     5   20  ||    -1   -1  ||    -1   -1  ||     6   18  ||    -1   -1  |
|    -1   -1  ||    -1   -1  ||    -1   -1  ||    -1   -1  ||     6   23  ||    -1   -1  ||    -1   -1  ||    -1   -1  ||    -1   -1  |
```

그림 12 Functionality Check를 위해 그림 5에 대한 예제를 수행한 결과

4 실험

4.1 Simple example & PointPillars 데이터를 활용한 성능 비교

지금까지 설계 및 검증한 HLS 기반의 하드웨어를 통해 두 가지 Dataset에 대한 성능을 알아보고자 한다. 첫 번째의 실험에서는 32x32 2D 공간에 임의의 좌표에 위치하는 40개의 point(**Sparsity 96%**)를 사용하여 SW & HW emulation 및 F1 instance에서 실행한 결과에 대해서 알아본다. 다음으로 PointPillars 모델을 통해 실제로 얻어지는 Sparse한 2D Pillars에 대해서 Rule을 생성하는 것을 앞의 경우와 동일하게 SW & HW emulation 및 F1 instance에서 실행해본다. 정리하면, Dataset은 다음과 같다.

1. Simple example: 40 points in 32x32 2D space (Sparsity: 96%)

2. PointPillars: 4551 Pillars (points) in 320x280 2D space (Sparsity: 95%)

다음 두 가지에 대해서 실행한 결과는 그림 13과 같다. 두 경우는 Sparsity에는 비슷한 부분이 존재하지만, 실제

존재하는 데이터의 크기 자체에 차이가 존재하므로 이러한 결과가 나왔음을 직관적으로 판단할 수 있다. 하지만 좀 더 자세히 성능에 영향을 주는 부분을 파악하기 위해서 추가적으로 3가지의 실험을 진행하였다.

Dataset	Simple example	PointPillars data	
SW emulation	7.225 ms	830.828 ms	**x115**
HW emulation	0.096 ms	6.466 ms	**x67**
Run on F1	0.207 ms	8.600 ms	**x42**

그림 13 Simple example과 PointPillars 데이터에 대한 실행 결과

4.2 CSR 기반 Rule generator 성능 분석을 위한 추가 실험

3가지의 실험은 Dataset의 Row size, Column size, # of Data 에 대해서 직접적인 연관성을 파악하기 위함이다. 따라서 각각의 값들에 대한 영향을 파악하기 위해서 해당 값을 제외한 다른 값들은 고정한 상태에서 해당 변수 값만 바꿔가면서 HW emulation을 진행하였다. 기본적인 세팅은 Simple example에 해당하는 32x32 2D space 상의 40개의 point이고, 변화를 위해 Column size를 32에서 64, 128까지 변화를 주었고, Row size도 Column size와 동일하게 변화를 주었으며, # of Data는 40, 80, 160, 320, 640개까지 변화를 주면서 수행한 시간을 분석해 보았다. 해당 결과는 **그림 14**의 표와 같다. 표의 결과를 통해 Rule generator의 성능이 Column size와 # of Data에 Dependent하다는 것을 유추할 수 있다.

Column size	Time
32	0.090 ms
64	0.113 ms
128	0.155 ms

Row size	Time
32	0.090 ms
64	0.111 ms
128	0.105 ms

# of Data	Time
40	0.090 ms
80	0.118 ms
160	0.153 ms
320	0.230 ms
640	0.244 ms

그림 14 Row/Column size 및 # of Data에 따른 성능

4.3 Dataflow Optimization 적용 시 성능 비교

2장에서 보았듯이, Merge 및 Dilate에서 Unrolling을 적용하기가 매우 어려웠기 때문에, Dataflow 방법을 적용하여 성능을 최적화 하고자 했다. 그래서 각 부분을 나눠서 구현하고 내부에 Data-stream을 사용하여 Dataflow가 적용될 수 있도록 코드를 수정해주었다. CSR format 기반의 Rule generator에 Dataflow를 적용할 수 있도록 **그림 15**와 같이 5개의 모듈로 나눠서 구현하였으며, 각 모듈 간에 주고 받는 데이터는 각 화살표에 나타낸 부분을 Data-stream 형태로 구현하여 Dataflow가 잘 작동하도록 해주었다.

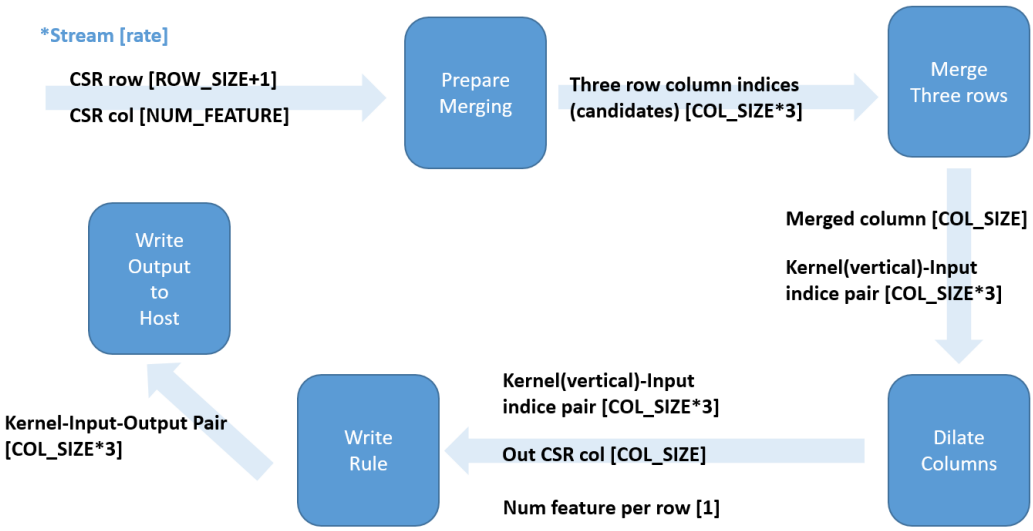


그림 15 Dataflow를 적용하기 위해 나눠진 HLS Code의 Overview

Simple example에 대해서 두 가지 방식으로 Dataflow를 적용해 보았는데, Dataflow(R)는 Row loop 단위에서 Data-stream을 적용하여 Dataflow가 이루어지도록 한 것이다. 그렇게 진행한 결과 **그림 16**과 같이 모듈 간에 사용되는 Data-stream이 Column size에 dependent 한 것을 볼 수 있다. 따라서 Dataflow가 PointPillars와 같이 Column size가 큰 Dataset에 대해서는 동작하지 않는 문제가 발생한다. 그래서 두번째의 경우인 Dataflow(RC)는 Row loop 안에 있는 Column loop에서 Data-stream을 주고 받을 수 있도록 수정하였다 (**그림 17**). 하지만 **그림 17**의 빨간 네모 칸과 같이 여전히 Column size에 대해서 dependent한 부분이 존재하여, 이 마저도 PointPillars에 대해서 Dataflow를 적용할 수 없었다.

```
extern "C" {
void vadd(hls::vector<DTYPE,(ROW_SIZE+1)> *csr_row,
          hls::vector<DTYPE,(NUM_FEATURE)> *csr_col,
          hls::vector<DTYPE, 3> *out_Rule){
#pragma HLS INTERFACE mode=m_axi bundle=gmem0 port=csr_row depth=600
#pragma HLS INTERFACE mode=m_axi bundle=gmem1 port=csr_col depth=600
#pragma HLS INTERFACE mode=m_axi bundle=gmem2 port=out_Rule depth=600

#pragma HLS DATAFLOW
    hls::stream<hls::vector<DTYPE,(ROW_SIZE+1)>> CsrRow("CsrRowStream");
    hls::stream<hls::vector<DTYPE,(ROW_SIZE+1)>> CsrRow_("CsrRow2Stream");
    hls::stream<hls::vector<DTYPE,NUM_FEATURE>> CsrCol("CsrColStream");
    hls::stream<hls::vector<DTYPE,COL_SIZE*3>> Candidates("CandidatesStream");
    hls::stream<hls::vector<DTYPE,COL_SIZE>> MergedCol("MergedColStream");
    hls::stream<hls::vector<DTYPE,COL_SIZE>> MergedCol_("MergedCol2Stream");
    hls::stream<hls::vector<DTYPE,COL_SIZE*3>> KIPairs("KIPiarsStream");
    hls::stream<hls::vector<DTYPE,COL_SIZE*3>> KIPairs_("KIPiars2Stream");
    hls::stream<hls::vector<DTYPE,COL_SIZE*3>> KStream("KStream");
    hls::stream<hls::vector<DTYPE,COL_SIZE*3>> IStream("IStream");
    hls::stream<hls::vector<DTYPE,COL_SIZE*3>> OStream("OStream");
    hls::stream<DTYPE> RuleWriteLength("RuleWriteLengthStream"); // kernel, input, output
    hls::stream<DTYPE> OutNumFeaturePerRow("OutNumFeaturePerRowStream");
    hls::stream<hls::vector<DTYPE,COL_SIZE>> OutCsrCol("OutCsrColStream");
    readCsrData(CsrRow, CsrCol, csr_row, csr_col);
```

그림 16 Row loop 단위에서 Data-stream을 적용하여 Dataflow를 구현했을 때의 Write Rule

```
extern "C" {
void vadd(hls::vector<DTYPE,(ROW_SIZE+1)> *csr_row,
          hls::vector<DTYPE,(NUM_FEATURE)> *csr_col,
          hls::vector<DTYPE,3> *out_Rule){
#pragma HLS INTERFACE mode=m_axi bundle=gmem0 port=csr_row depth=600
#pragma HLS INTERFACE mode=m_axi bundle=gmem1 port=csr_col depth=600
#pragma HLS INTERFACE mode=m_axi bundle=gmem2 port=out_Rule depth=600

#pragma HLS DATAFLOW
    hls::stream<hls::vector<DTYPE>> CsrRow("CsrRowStream");
    hls::stream<hls::vector<DTYPE>> CsrRow_("CsrRow2Stream");
    hls::stream<hls::vector<DTYPE>> CsrCol("CsrColStream");
    hls::stream<hls::vector<DTYPE,COL_SIZE*3>> Candidates("CandidatesStream");
    hls::stream<DTYPE> MergedCol("MergedColStream");
    hls::stream<DTYPE> MergedCol_("MergedCol2Stream");
    hls::stream<hls::vector<DTYPE,3>> KIPairs("KIPiarsStream");
    hls::stream<hls::vector<DTYPE,3>> KIPairs_("KIPiars2Stream");
    hls::stream<hls::vector<DTYPE,3*KERNEL_SIZE*KERNEL_SIZE>> RuleStream("RuleStream"); // kernel, input, output
    hls::stream<DTYPE> RuleLength("RuleLength");
    hls::stream<DTYPE> OutNumFeaturePerRow("OutNumFeaturePerRowStream");
    hls::stream<hls::vector<DTYPE>> OutCsrCol("OutCsrColStream");
    readCsrData(CsrRow, CsrCol, csr_row, csr_col);
```

그림 17 Column loop 단위에서 Data-stream을 적용하여 Dataflow 구현했을 때의 Write Rule

Candidates에 해당하는 Data-stream은 그림 18에서와 같이 Merge를 하기 위해 사용될 Input data의 candidate row에 존재하는 값들을 찾기 위해서 CSR format의 CSR row 값을 통해 추출하게 되는데, 이때 for loop의 count 가 사전에 정확히 알 수 없기 때문에 최대의 경우인 Column size 만큼 실행될 것을 대비하여 Candidate 값을 한 번에 Column 단위로 받아 올 수 밖에 없기 때문에 발생하는 문제이다. 이 부분은 코드 수정을 통해서 모든 Column loop에 대한 Data-stream을 적용하여 Dataflow를 완성할 수 있을 것으로 예상한다.

```
void prepare_merge(hls::stream<DTYPE, COL_SIZE*3> &Candidates,
                   hls::stream<DTYPE> &CsrRow_,
                   hls::stream<DTYPE> &CsrRow,
                   hls::stream<DTYPE> &CsrCol){
    hls::vector<DTYPE> csr_row;
    hls::vector<DTYPE> csr_col;
    DTYPE csr_row_b[ROW_SIZE+1];
    DTYPE csr_col_b[NUM_FEATURE];
    for (int i=0; i<=ROW_SIZE; i++){
        csr_row = CsrRow.read();
        csr_row_b[i] = csr_row;
        CsrRow_.write(csr_row);
    }
    for (int i=0; i<NUM_FEATURE; i++){
        csr_col = CsrCol.read();
        csr_col_b[i] = csr_col;
    }

    prepare_merge_row_loop: for(int r=0; r<ROW_SIZE; r++){
        hls::vector<DTYPE, COL_SIZE*3> candidate;
        init_entry: for(int i=0; i<COL_SIZE*3; i++){
            candidate[i] = COL_SIZE;
        }
        if (r==0){
            prepare_merge_loop0_1: for(int i=0; i<(csr_row_b[r+1]-csr_row_b[r]); i++){
                candidate[COL_SIZE+i] = csr_col_b[csr_row_b[r]+i];
            }
            prepare_merge_loop0_2: for(int i=0; i<(csr_row_b[r+2]-csr_row_b[r+1]); i++){
                candidate[COL_SIZE*2+i] = csr_col_b[csr_row_b[r+1]+i];
            }
        }
        else if (r==ROW_SIZE-1){
            prepare_merge_loop1_0: for(int i=0; i<(csr_row_b[r]-csr_row_b[r-1]); i++){
                candidate[i] = csr_col_b[csr_row_b[r-1]+i];
            }
            prepare_merge_loop1_1: for(int i=0; i<(csr_row_b[r+1]-csr_row_b[r]); i++){
                candidate[COL_SIZE+i] = csr_col_b[csr_row_b[r]+i];
            }
        }
        else {
            prepare_merge_loop2_0: for(int i=0; i<(csr_row_b[r]-csr_row_b[r-1]); i++){
                candidate[i] = csr_col_b[csr_row_b[r-1]+i];
            }
            prepare_merge_loop2_1: for(int i=0; i<(csr_row_b[r+1]-csr_row_b[r]); i++){
                candidate[COL_SIZE+i] = csr_col_b[csr_row_b[r]+i];
            }
            prepare_merge_loop2_2: for(int i=0; i<(csr_row_b[r+2]-csr_row_b[r+1]); i++){
                candidate[COL_SIZE*2+i] = csr_col_b[csr_row_b[r+1]+i];
            }
        }
        Candidates.write(candidate);
    } // end row
}
```

그림 18 Prepare Merging에서 Merge를 위해 사용될 Candidate를 뽑아낼 때에 CSR format의 값을 통해 그 경우가 결정되므로 정확한 loop count를 알지 못해 최대 Column size만큼 실행했을 때에 대해서 결과를 넘겨주기 때문에 생기는 문제

그림 18번을 통해 설명한 문제를 해결하지 못한 상태에서 Simple example에 대해서는 Dataflow를 적용할 수 있어서, Simple example에 해당하는 Dataset을 통해 실험을 진행한 결과를 그림 19번 표에 정리하였다.

Optimization	Dataflow X	Dataflow (R)	Dataflow (RC)
SW emulation	7.225 ms	11.364 ms	2.202 ms
HW emulation	0.096 ms	0.078 ms	0.174 ms
Run on F1	0.207 ms	-	-

그림 19 Dataflow(R)과 Dataflow(RC)를 적용하였을 때 Dataflow를 적용하지 않는 경우와의 비교

5 결론 및 개선점

HLS를 사용하여 CSR format 기반의 Rule generator를 생성해본 결과, HLS에서 제공하는 '**#pragma HLS**'를 적용하는 데 어려움이 있었다. 그 이유는 중첩된 loop가 존재하더라도 loop문 사이에 dependency가 존재할 경우 한번에 하나씩 수행하여 다음 loop에 대해서 연산을 수행할 수 있기 때문에, 병렬 연산 자체가 불가능하기 때문이다. 따라서, Dataflow를 적용하면 이러한 문제에서도 성능 개선을 이루어 낼 것이라고 생각하였다. 하지만, 여전히 Data-stream의 사이즈가 Column size에 dependent한 문제가 있어서 Dataset이 커지면 지원하는 bit-width를 넘어서서 Data-stream을 적용할 수가 없게 된다.

본 프로젝트를 통해서 CSR format 기반의 Rule generator를 설계해 보았는데, Prepare Merging에 해당하는 모듈에서 Candidate를 뽑아내는 부분의 코드를 수정하여 Column에 independent하게 만들 수 있다면 Dataflow를 적용했을 때, Dataset의 크기에 상관없이 Scalable하게 구현 및 동작시킬 수 있을 것으로 생각한다. 또한 그림 19의 결과는 굉장히 적은 데이터를 사용하였기에, Dataflow를 적용한 것에 대한 성능 개선이 뚜렷하게 나타나지 않는 것으로 예상한다. C 앞에서 언급하였던 Column size에 dependent한 문제를 해결한다면, PointPillars 데이터에서는 훨씬 뚜렷한 성능 개선을 얻을 것으로 생각한다.

Appendix

A. Github 코드 주소

전체 코드는 https://github.com/superdocker/SoC-FinalProject-Team1 주소에서 확인할 수 있다.

다음 **그림 16~19**는 Dataflow를 적용하지 않은 Baseline에 해당하는 코드에 대해서만 간단하게 나타낸 것이다.

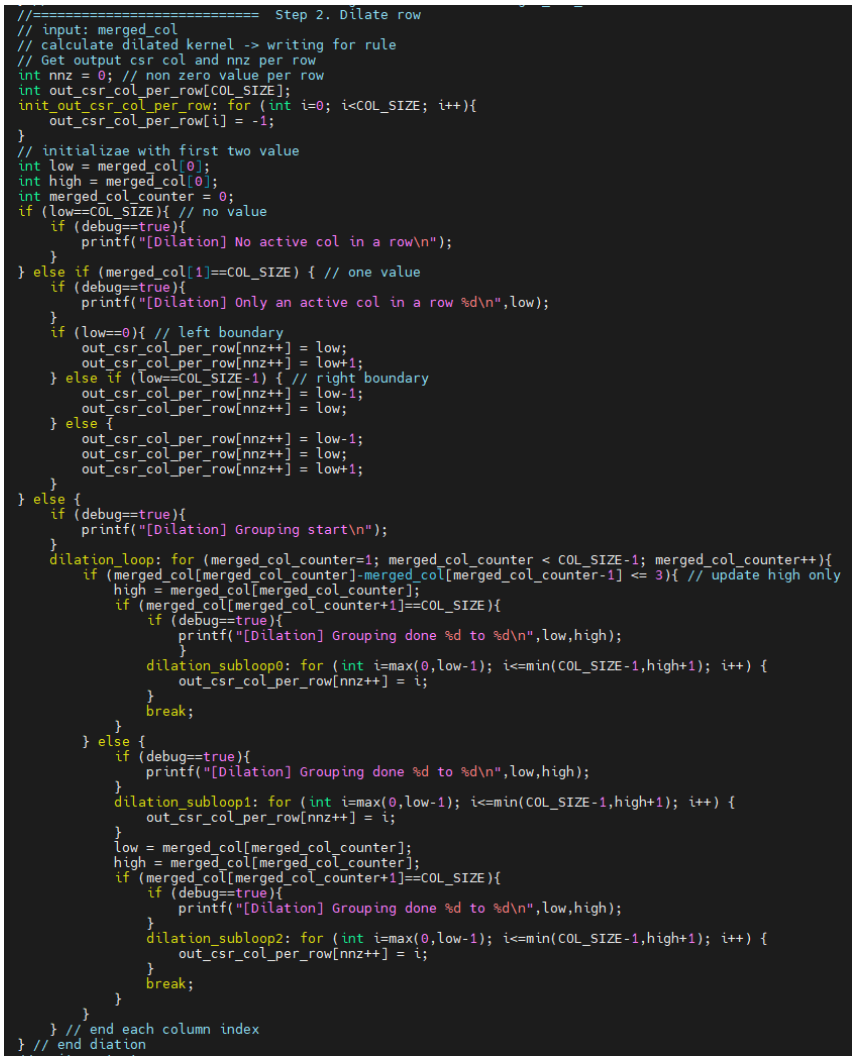
```
row_loop: for(int r=0; r<ROW_SIZE; r++){ // for each rows
    // Initialize with COL_SIZE (any column index could be larger than COL_SIZE)
    //=========================  Step 1. Merge rows
    init_entry: for (int i=0; i<COL_SIZE; i++){
        entry0[i] = COL_SIZE;
        entry1[i] = COL_SIZE;
        entry2[i] = COL_SIZE;
    }
    if (debug==true){
        printf("\nRow %d Start\n",r);
    }

    if (r==0){
        prepare_merge_loop0_1: for(int i=0; i<numFeaturePerRow[r]; i++){
            entry1[i] = csr_col[csr_row[r]+i];
        }
        prepare_merge_loop0_2: for(int i=0; i<numFeaturePerRow[r+1]; i++){
            entry2[i] = csr_col[csr_row[r+1]+i];
        }
    }
    else if (r==ROW_SIZE-1){
        prepare_merge_loop1_0: for(int i=0; i<numFeaturePerRow[r-1]; i++){
            entry0[i] = csr_col[csr_row[r-1]+i];
        }
        prepare_merge_loop1_1: for(int i=0; i<numFeaturePerRow[r]; i++){
            entry1[i] = csr_col[csr_row[r]+i];
        }
    }
    else {
        prepare_merge_loop2_0: for(int i=0; i<numFeaturePerRow[r-1]; i++){
            entry0[i] = csr_col[csr_row[r-1]+i];
        }
        prepare_merge_loop2_1: for(int i=0; i<numFeaturePerRow[r]; i++){
            entry1[i] = csr_col[csr_row[r]+i];
        }
        prepare_merge_loop2_2: for(int i=0; i<numFeaturePerRow[r+1]; i++){
            entry2[i] = csr_col[csr_row[r+1]+i];
        }
    }
```

그림 20 Prepare Merging에 해당하는 코드 부분

```
// Merge tree
// Compare min value iteratively
int merge_counter = 0;
int merged_col[COL_SIZE]; // size should be 3*COL_SIZE (dense case)
int input_index[3][COL_SIZE];
init_merge: for (int i=0; i<COL_SIZE; i++){
    merged_col[i] = COL_SIZE; // initialize with dummy value
    for (int j=0; j<3; j++){
        input_index[j][i] = -1;
    }
}
int e0_counter = 0;
int e1_counter = 0;
int e2_counter = 0;
int lowestIdx;
int lowestVal;
int kernelIdx;
int inputIdx;
// v2: using constant loop count
merge_loop: for (int i=0; i<COL_SIZE; i++){
    lowestIdx = minIdx(entry0[e0_counter],entry1[e1_counter],entry2[e2_counter]);
    // can be parallelized
    if ((lowestIdx&1)==1){
        kernelIdx = 0;
        inputIdx = e0_counter + csr_row[r-1];
        lowestVal = entry0[e0_counter];
        e0_counter++;
        input_index[kernelIdx][i] = inputIdx;
    }
    if ((lowestIdx&2)>>1==1){
        kernelIdx = 1;
        inputIdx = e1_counter + csr_row[r];
        lowestVal = entry1[e1_counter];
        e1_counter++;
        input_index[kernelIdx][i] = inputIdx;
    }
    if ((lowestIdx&4)>>2==1){
        kernelIdx = 2;
        inputIdx = e2_counter + csr_row[r+1];
        lowestVal = entry2[e2_counter];
        e2_counter++;
        input_index[kernelIdx][i] = inputIdx;
    }
    if (lowestVal==COL_SIZE){
        break;
    }
    merged_col[i] = lowestVal;
    // If no more value in next index -> skip
} // end merge_loop
if (debug==true){
    for (int i=0; i<COL_SIZE; i++){
        for (int j=0; j<3; j++){
            if (merged_col[i]!=COL_SIZE & input_index[j][i]!=-1){
                printf("[Merge] Input index %d(Column index %d) by Kernel index %d\n",input_index[j][i],merged_col[i], j);
            }
        }
    }
} //=========================  End Merge rows -> return merged_col_idx
```

그림 21 Merge에 해당하는 코드 부분



```
//============================  Step 2. Dilate row
// input: merged_col
// calculate dilated kernel -> writing for rule
// Get output csr col and nnz per row
int nnz = 0; // non zero value per row
int out_csr_col_per_row[COL_SIZE];
init_out_csr_col_per_row: for (int i=0; i<COL_SIZE; i++){
    out_csr_col_per_row[i] = -1;
}
// initializae with first two value
int low = merged_col[0];
int high = merged_col[0];
int merged_col_counter = 0;
if (low==COL_SIZE){ // no value
    if (debug==true){
        printf("[Dilation] No active col in a row\n");
    }
} else if (merged_col[1]==COL_SIZE) { // one value
    if (debug==true){
        printf("[Dilation] Only an active col in a row %d\n",low);
    }
    if (low==0){ // left boundary
        out_csr_col_per_row[nnz++] = low;
        out_csr_col_per_row[nnz++] = low+1;
    } else if (low==COL_SIZE-1) { // right boundary
        out_csr_col_per_row[nnz++] = low-1;
        out_csr_col_per_row[nnz++] = low;
    } else {
        out_csr_col_per_row[nnz++] = low-1;
        out_csr_col_per_row[nnz++] = low;
        out_csr_col_per_row[nnz++] = low+1;
    }
} else {
    if (debug==true){
        printf("[Dilation] Grouping start\n");
    }
    dilation_loop: for (merged_col_counter=1; merged_col_counter < COL_SIZE-1; merged_col_counter++){
        if (merged_col[merged_col_counter]-merged_col[merged_col_counter-1] <= 3){ // update high only
            high = merged_col[merged_col_counter];
            if (merged_col[merged_col_counter+1]==COL_SIZE){
                if (debug==true){
                    printf("[Dilation] Grouping done %d to %d\n",low,high);
                    }
                dilation_subloop0: for (int i=max(0,low-1); i<=min(COL_SIZE-1,high+1); i++) {
                    out_csr_col_per_row[nnz++] = i;
                }
                break;
            }
        } else {
            if (debug==true){
                printf("[Dilation] Grouping done %d to %d\n",low,high);
            }
            dilation_subloop1: for (int i=max(0,low-1); i<=min(COL_SIZE-1,high+1); i++) {
                out_csr_col_per_row[nnz++] = i;
            }
            low = merged_col[merged_col_counter];
            high = merged_col[merged_col_counter];
            if (merged_col[merged_col_counter+1]==COL_SIZE){
                if (debug==true){
                    printf("[Dilation] Grouping done %d to %d\n",low,high);
                }
                dilation_subloop2: for (int i=max(0,low-1); i<=min(COL_SIZE-1,high+1); i++) {
                    out_csr_col_per_row[nnz++] = i;
                }
                break;
            }
        }
    } // end each column index
} // end diation
```

그림 22 Dilate에 해당하는 코드 부분

```
// Step 3. Write Rule
int nnz_rule = 0;
int outIdx;
write_rule_loop: for (int i=0; i<COL_SIZE; i++){
    if (merged_col[i]!=COL_SIZE){
        if (merged_col[i]==0){ // left boundary
            write_rule_loop0: for (int j=0; j<3; j++){
                if (input_index[j][i]!=-1){ // must be first group
                    if (debug==true){
                        printf("[Rule] Kernel %d(-) and %d(|) makes input %d to output %d\n",j,1,input_index[j][i], out_csr_row[r]);
                        printf("[Rule] Kernel %d(-) and %d(|) makes input %d to output %d\n",j,0,input_index[j][i], out_csr_row[r]+1);
                    }
                    //Rule[j*3+1][0][rule_counter[j*3+1]] = input_index[j][i];
                    //Rule[j*3+1][1][rule_counter[j*3+1]++] = out_csr_row[r];
                    //Rule[j*3][0][rule_counter[j*3]] = input_index[j][i];
                    //Rule[j*3][1][rule_counter[j*3]++] = out_csr_row[r]+1;
                    Rule[(j*3+1)*RK+rule_counter[j*3+1]] = input_index[j][i];
                    Rule[(j*3+1)*RK+RO+rule_counter[j*3+1]++] = out_csr_row[r];
                    Rule[(j*3)*RK+rule_counter[j*3]] = input_index[j][i];
                    Rule[(j*3)*RK+RO+rule_counter[j*3]++] = out_csr_row[r]+1;
                }
            }
        } else if (merged_col[i]==COL_SIZE-1){ // right boundary
            write_rule_loop1: for (int j=0; j<3; j++){
                if (input_index[j][i]!=-1){ // must be last group
                    if (debug==true){
                        printf("[Rule] Kernel %d(-) and %d(|) makes input %d to output %d\n",j,2,input_index[j][i], out_csr_row[r+1]-2);
                        printf("[Rule] Kernel %d(-) and %d(|) makes input %d to output %d\n",j,1,input_index[j][i], out_csr_row[r+1]-1);
                    }
                    Rule[(j*3+2)*RK+rule_counter[j*3+2]] = input_index[j][i];
                    Rule[(j*3+2)*RK+RO+rule_counter[j*3+2]++] = out_csr_row[r+1]-2;
                    Rule[(j*3+1)*RK+rule_counter[j*3+1]] = input_index[j][i];
                    Rule[(j*3+1)*RK+RO+rule_counter[j*3+1]++] = out_csr_row[r+1]-1;
                }
            }
        } else {
            write_rule_loop2: for (int j=0; j<3; j++){
                if (input_index[j][i]!=-1){
                    for (int n=0; n<nnz; n++){ // is it neccessary?
                        if (merged_col[i]==out_csr_col_per_row[n]){
                            outIdx = n;
                        }
                    }
                    if (debug==true){
                        printf("[Rule] Kernel %d(-) and %d(|) makes input %d to output %d\n",j,2,input_index[j][i],out_csr_row[r]+outIdx-1);
                        printf("[Rule] Kernel %d(-) and %d(|) makes input %d to output %d\n",j,1,input_index[j][i],out_csr_row[r]+outIdx);
                        printf("[Rule] Kernel %d(-) and %d(|) makes input %d to output %d\n",j,0,input_index[j][i],out_csr_row[r]+outIdx+1);
                    }
                    Rule[(j*3+2)*RK+rule_counter[j*3+2]] = input_index[j][i];
                    Rule[(j*3+2)*RK+RO+rule_counter[j*3+2]++] = out_csr_row[r]+outIdx-1;
                    Rule[(j*3+1)*RK+rule_counter[j*3+1]] = input_index[j][i];
                    Rule[(j*3+1)*RK+RO+rule_counter[j*3+1]++] = out_csr_row[r]+outIdx;
                    Rule[(j*3)*RK+rule_counter[j*3]] = input_index[j][i];
                    Rule[(j*3)*RK+RO+rule_counter[j*3]++] = out_csr_row[r]+outIdx+1;
                }
            }
        }
    }
}
```

그림 23 Write Rule에 해당하는 코드 부분

자세한 코드에 대한 설명과 실행 방법은 Github 주소인 https://github.com/superdocker/SoC-FinalProject-Team1 에 업로드 해두었다.