Sparse Matrix Vector Multiplication

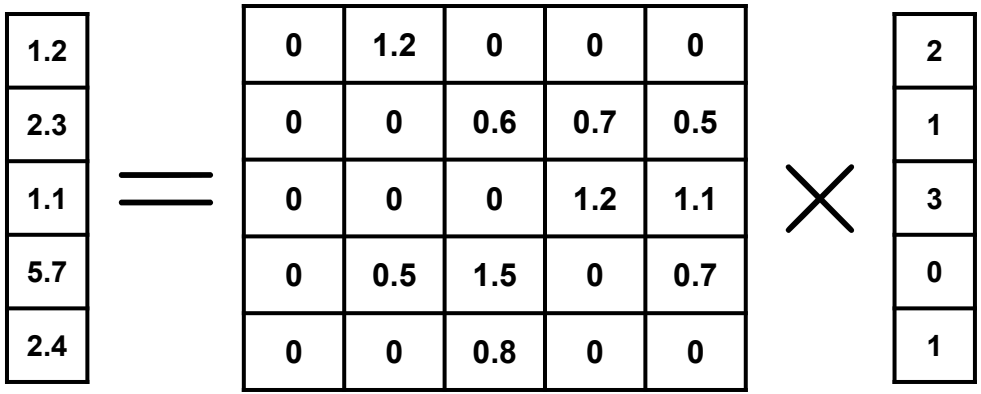
Speaker: Jia-Ming Lin

Outline

- Sparse Matrix Vector Multiplication and Applications
- Compressed Row Storage(CRS)
- Baseline implementation
- C/RTL Cosimulation
- Loop Optimization
- Labs

Sparse Matrix Vector Multiplication and Applications

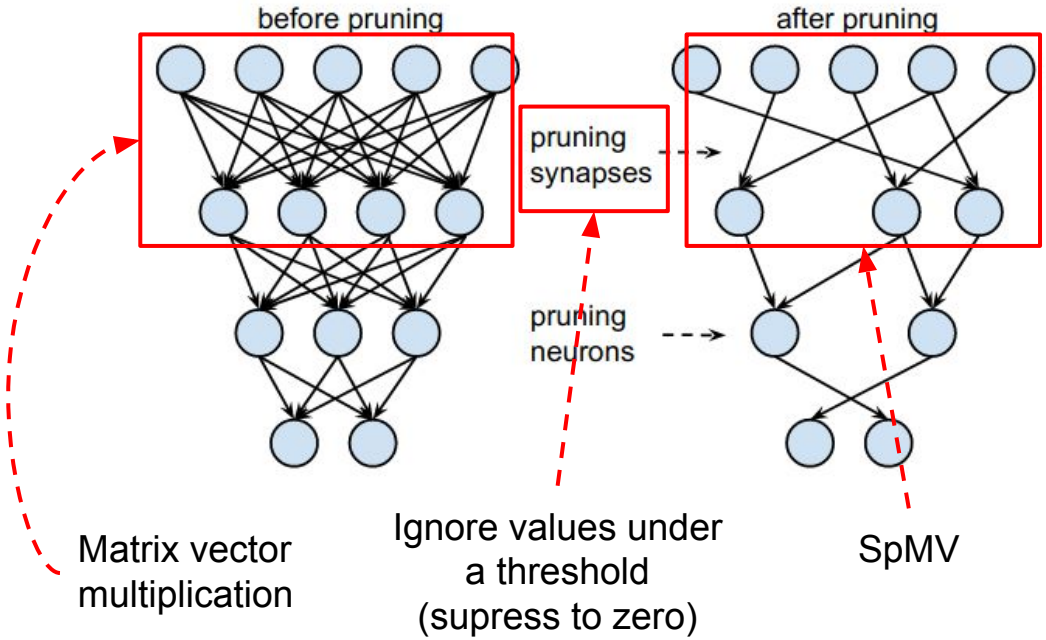
- Sparse Matrix Vector Multiplication(SpMV)



- Zero value does not contribute to the result of vector product
- Discarding the zero values
  - Saving storage
  - Reducing # of mul and add.
- How to store the sparse matrix?
- How to perform multiplication?
  - Without converting to dense matrix-vector multiply

Sparse Matrix Vector Multiplication and Applications

- **Application**: Neural Network Compression



Network	Top-1 Error	Top-5 Error	Parameters	Compression Rate
LeNet-300-100 Ref	1.64%	-	267K	
LeNet-300-100 Pruned	1.59%	-	**22K**	**12**×
LeNet-5 Ref	0.80%	-	431K	
LeNet-5 Pruned	0.77%	-	**36K**	**12**×
AlexNet Ref	42.78%	19.73%	61M	
AlexNet Pruned	42.77%	19.67%	**6.7M**	**9**×
VGG-16 Ref	31.50%	11.32%	138M	
VGG-16 Pruned	31.34%	10.88%	**10.3M**	**13**×

- Large number of values(near to zero) can be discarded.
  - Without sacrificing much accuracy.

Compressed Row Storage(CRS)

- Example

0	1.2	0	0	0
0	0	0.6	0.7	0.5
0	0	0	0	1.1
0	0.5	1.5	0	0.7
0	0	0.8	0	0

2D Dense Matrix Representation

Values

1.2	0.6	0.7	0.5	1.1	0.5	1.5	0.7	0.8

Column Index

1	2	3	4	4	1	2	4	2

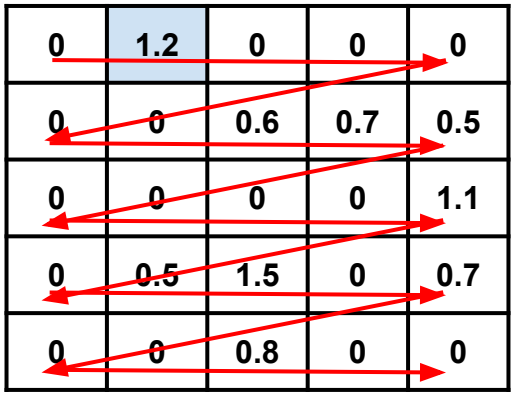
Row Pointer

0	1	4	5	8	9

- Comparison(storage saving)
  - 2D Dense Matrix(left): 4 bytes * 25 = 100 bytes
  - CRS(right): 4 bytes * 9 + 1 byte * (9+6) = 51 bytes

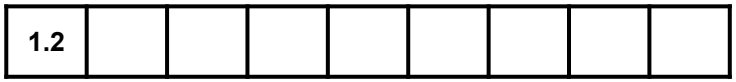
Compressed Row Storage(CRS)

- Example: given 2D matrix, converting to CRS



2D Dense Matrix Representation

Values



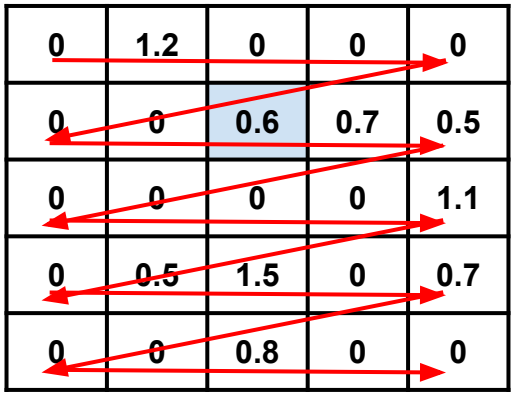
Column Index

Row Pointer

- Scanning in a Row-Major manner.
- Put the non-zero(NZ) value in the “Values” array
- Record the column index of NZ
- Initialize the array of “Row Pointer” of length “#Row” +1
  - First element is zero, no NZ before first row.

Compressed Row Storage(CRS)

- Example: given 2D matrix, converting to CRS



2D Dense Matrix Representation

Values

1.2	0.6							

Column Index

1	2							

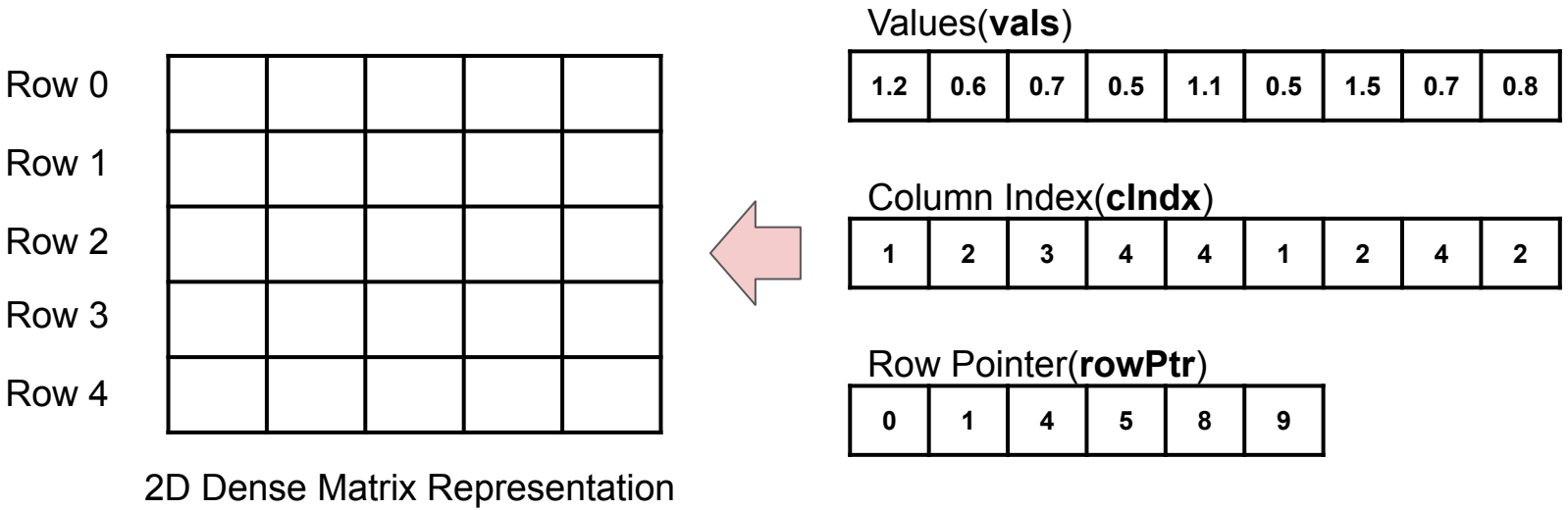
Row Pointer

0	1				

- Scanning in a Row-Major manner.
- Put the non-zero(NZ) value in the “Values” array
- Record the column index of NZ
- “**Row Pointer**”: 1 NZ before second row.

Compressed Row Storage(CRS)

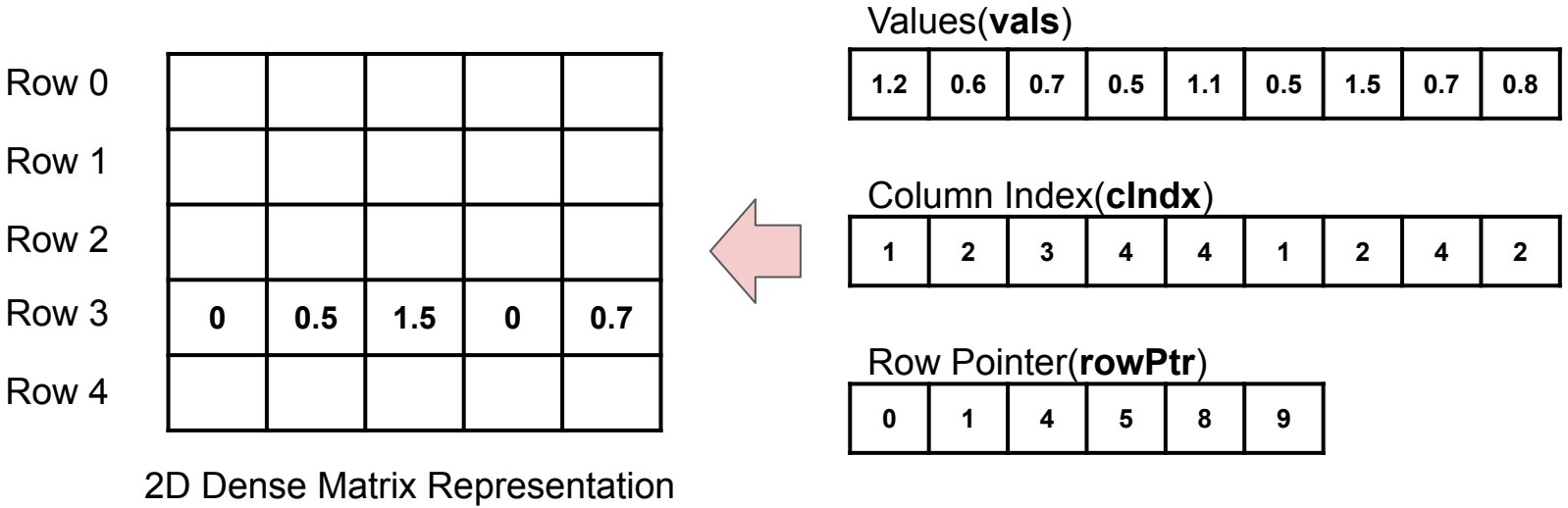
- Example: Given CRS, converting to 2D Matrix



- How many NZ in a row? rowPtr[i+1] - rowPrt[i]
- What is the column indexes of NZs in a row? cIndx[rowPtr[i]...(rowPtr[i+1]-1)]
- What is the values of NZs in a row? vals[rowPtr[i]...(rowPtr[i+1]-1)]

Compressed Row Storage(CRS)

- Example: Given CRS, converting to 2D Matrix



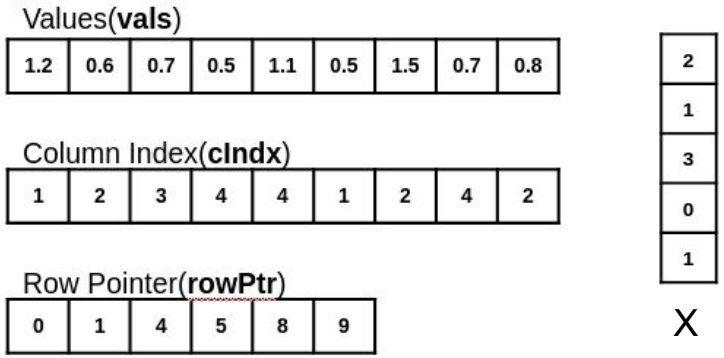
- How many NZ in a **Row 3**? rowPtr[3+1] - rowPrt[3] = 8 - 5 = 3
- What is the column indexes of the NZs? clndx[rowPtr[3]...(rowPtr[3+1]-1)] = clndx[5...7] = [1,2,4]
- What is the values of NZs in a row? vals[rowPtr[3]...(rowPtr[3+1]-1)] = vals[5...7] = [0.5, 1.5, 0.7]

Baseline Implementation

- Given CRS and vector **X[N]**
  - **Compute Y[N]**
- To compute Y[n], needs **Row n * X[N]**
- Previously, we have seen how to obtain “Row n” from CRS
  - With NZ column indexes and values
- Using the NZ column indexes to retrieve corresponding values in X[N]
- Obtaining result Y[n] by only looking at the operations of NZs.
- Refer lab for the HLS code.

Baseline Implementation

- Example: To compute **Y[3]**
- In **Row 3**
  - NZ values = [0.5, 1.5, 0.7]
  - NZ column index = [1, 2, 4]
- Corresponding operands in X[N]
  - X[1], X[2], X[4]
- Y[3] = 0.5*X[1] + 1.5*X[2] + 0.7*X[4]
  = 0.5*1 + 1.5*3 + 0.7*1 = 5.7
- Reduced Operations(mul and add)
  - Mul: 5 → 3, Add: 4 → 2



C/RTL Cosimulation

- In the “Synthesis” report, latency is unknown.
  - Since for every Y[n], loop length is unknown (depends on data)

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
?	?	?	?	?	?	none

- Two methods to evaluate performance,
  - Using Directive “loop_tripcount”
  - Doing C/RTL Cosimulation

Directive “loop_tripcount”

- To specifying the loop length manually.
- Needs to know the behaviors of program very well.

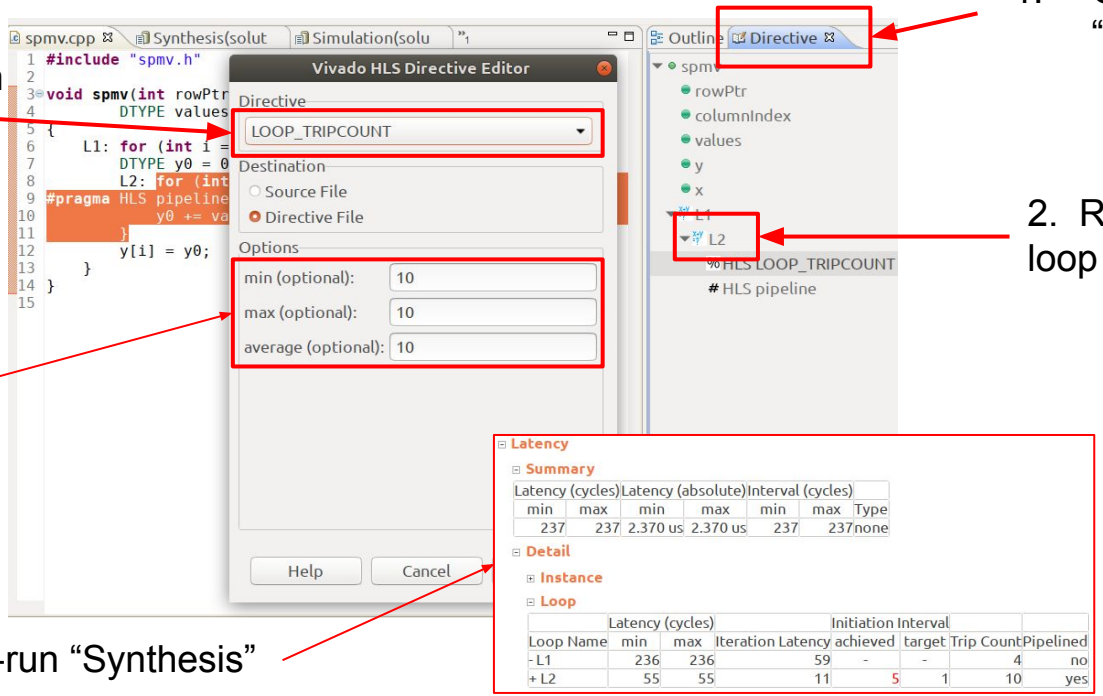
1. Open the “Directive” tab

2. Right click on L2 loop

3. Switch the drop down list to “LOOP_TRIPCOUNT”

4. Specifying the number of loop length manually

5. Re-run “Synthesis”



C/RTL Cosimulation

- Simulation using the converted RTL code
- Input data from test bench in C
- Performance evaluation and result validation

Test Bench

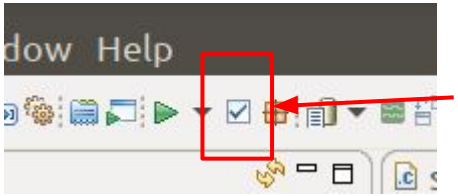
```
int main(){
    int fail = 0;
    DTYPE M[SIZE][SIZE] = {{3,4,0,0},{0,5,9,0},{2,0,3,1},{0,4,0,6}};
    DTYPE x[SIZE] = {1,2,3,4};
    DTYPE y_sw[SIZE];
    DTYPE values[] = {3,4,5,9,2,3,1,4,6};
    int columnIndex[] = {0,1,1,2,0,2,3,1,3};
    int rowPtr[] = {0,2,4,7,9};
    DTYPE y[SIZE];

    // hardware implementation
    spmv(rowPtr, columnIndex, values, y, x);
    matrixvector(M, y_sw, x);

    for(int i = 0; i < SIZE; i++)
        if(y_sw[i] != y[i])
            fail = 1;

    if(fail == 1)
        printf("FAILED\n");
    else
        printf("PASS\n");

    return fail;
}
```



Click to using C/RTL Cosimulation

Cosimulation Report for 'spmv'

Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	82	82	82	NA	NA	NA

Loop Optimization

- Case 1: Pipeline the loop L1
- Case 2: Pipeline the loop L2
- Case 3: Pipeline/Unroll the loop L2
- Case 4: Splitting L2 and partially unroll manually

```
void spmv(int rowPtr[NUM_ROWS+1], int columnIndex[NNZ],
        DTYPE values[NNZ], DTYPE y[SIZE], DTYPE x[SIZE])
{
    // Loop L1
    L1: for (int i = 0; i < NUM_ROWS; i++) {
        DTYPE y0 = 0;

        // Loop L2
        L2: for (int k = rowPtr[i]; k < rowPtr[i+1]; k++) {
            y0 += values[k] * x[columnIndex[k]];
        }
        y[i] = y0;
    }
}
```

Loop Optimization

- Case 1: Pipeline loop L1
  - Fully unrolling the loop L2
  - However, the length of loop L2 is unknown and variable loop is not supported in Vivado HLS.

```
INFO: [XFORM 203-502] Unrolling all sub-loops inside loop 'L1' (baseline/spmv.cpp:7) in function 'spmv' for pipelining.
WARNING: [XFORM 203-503] Cannot unroll loop 'L2' (baseline/spmv.cpp:12) in function 'spmv' completely: variable loop bound.
INFO: [HLS 200-111] Finished Pre-synthesis Time (s): cpu = 00:00:07 ; elapsed = 00:00:07 . Memory (MB): peak = 1676.246 ; g
```

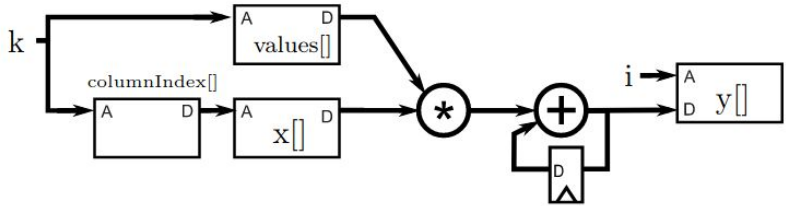
  - This optimization does not have any effect.

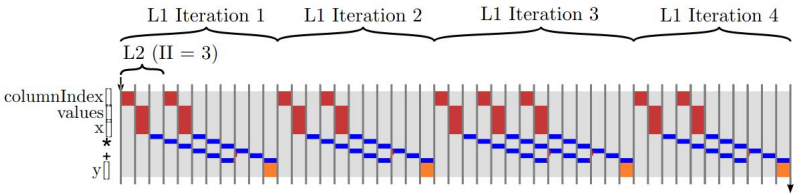
Loop Optimization

- Case 2: Pipeline the loop L2

```
void spmv(int rowPtr[NUM_ROWS+1], int columnIndex[NNZ],
        DTYPE values[NNZ], DTYPE y[SIZE], DTYPE x[SIZE])
{
    // Loop L1
    L1: for (int i = 0; i < NUM_ROWS; i++) {
        DTYPE y0 = 0;

        // Loop L2
        L2: for (int k = rowPtr[i]; k < rowPtr[i+1]; k++) {
#pragma HLS PIPELINE
            y0 += values[k] * x[columnIndex[k]];
        }
        y[i] = y0;
    }
}
```





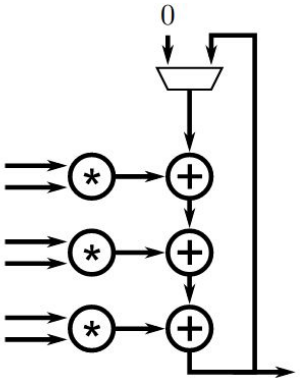
- Sequentially perform mul and add

Loop Optimization

- Case 3: Pipeline/Unroll the loop L2
  - Increasing the parallelism by unrolling L2

```
void spmv(int rowPtr[NUM_ROWS+1], int columnIndex[NNZ],
        DTYPE values[NNZ], DTYPE y[SIZE], DTYPE x[SIZE])
{
    // Loop L1
    L1: for (int i = 0; i < NUM_ROWS; i++) {
        DTYPE y0 = 0;

        // Loop L2
        L2: for (int k = rowPtr[i]; k < rowPtr[i+1]; k++) {
#pragma HLS PIPELINE
#pragma HLS UNROLL factor=3
            y0 += values[k] * x[columnIndex[k]];
        }
        y[i] = y0;
    }
}
```



Comparing Case 2 and Case 3

- Latency for Case 2 = 82 cycles
- Latency for Case 3 = 96 cycles

Long chain of operation dependencies

Loop Optimization

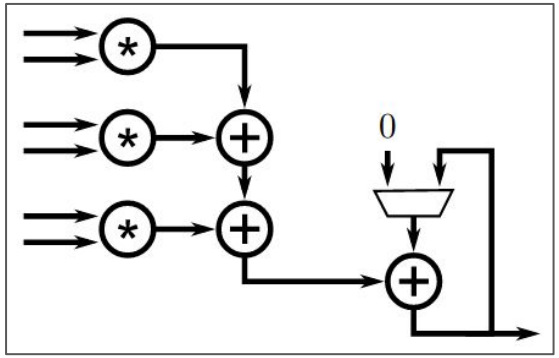
- Case 4: Splitting L2 and partially unroll manually

```
const static int S = 3;

void spmv(int rowPtr[NUM_ROWS+1], int columnIndex[NNZ],
          DTYPE values[NNZ], DTYPE y[SIZE], DTYPE x[SIZE])
{
  L1: for (int i = 0; i < NUM_ROWS; i++) {
      DTYPE y0 = 0;

      L2_1: for (int k = rowPtr[i]; k < rowPtr[i+1]; k += S) {
#pragma HLS pipeline II=S
          DTYPE yt = values[k] * x[columnIndex[k]];
          L2_2: for(int j = 1; j < S; j++) {
                  if(k+j < rowPtr[i+1]) {
                      yt += values[k+j] * x[columnIndex[k+j]];
                  }
          }
        y0 += yt;
      }

    y[i] = y0;
  }
}
```



Comparing to **Case 3**:

- **(Pros)**Recurrence only on final accumulation
- **(Pros)**Less Initial Interval for case 4
  - Case 3: 15; case 4: 5
- **(Cons)** Deeper pipeline
  - Longer latency for single task

Loop Optimization

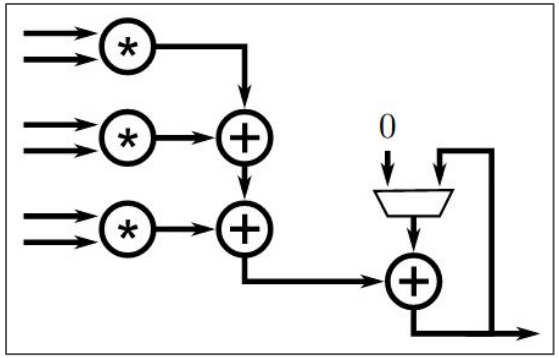
- Case 4: Splitting L2 and partially unroll manually

```
const static int S = 3;

void spmv(int rowPtr[NUM_ROWS+1], int columnIndex[NNZ],
          DTYPE values[NNZ], DTYPE y[SIZE], DTYPE x[SIZE])
{
  L1: for (int i = 0; i < NUM_ROWS; i++) {
      DTYPE y0 = 0;

      L2_1: for (int k = rowPtr[i]; k < rowPtr[i+1]; k += S) {
#pragma HLS pipeline II=S
          DTYPE yt = values[k] * x[columnIndex[k]];
          L2_2: for(int j = 1; j < S; j++) {
                  if(k+j < rowPtr[i+1]) {
                      yt += values[k+j] * x[columnIndex[k+j]];
                  }
          }
        y0 += yt;
      }

    y[i] = y0;
  }
}
```



Comparing to **Case 2**:

- **(Pros)** More parallelism
- **(Cons)** resources are tend to be wasted in very sparse cases
  - Ex. for a row with only one NZ, 2 multiplier are wasted.

Summarize

- Exploit sparsity in matrix operation
  - Reduce # of operations
  - Reduce required storages
- Loop optimization for a spatial(irregular) data structure
  - Serialize and pipeline the calculations of Y[n], that is case 2: pipeline L2
    - More suitable for very sparse case
  - Partially parallelize(unroll) the calculations of Y[n], that is case 3 and 4: partially unroll for L2
    - More suitable for the cases with more NZ elements.

Labs

- Goal: find a case of sparse matrix, such that “Case 4” can perform better than “Case 2”
  - Download baseline and case 4 files
  - C simulation, validating using C
  - C/RTL cosimulation, performance evaluation using given test bench
  - Using loop optimization, Case 2 and Case 4
  - Modifying the test bench and using C/RTL cosimulation again