Survey on Sparse Matrix-Vector Multiplication

r13943052 黃秉璿
陳靖雯

Contents

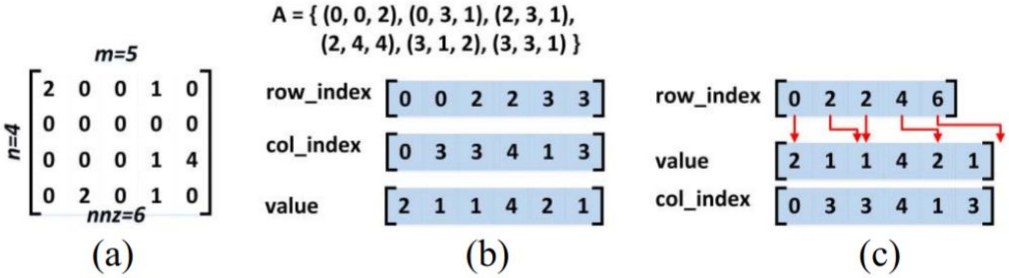
- Introduction of Sparse Matrix-Vector Multiplication (SpMV)
- Formats for SpMV
- Structure of Streaming Data Engine for SpMV
- Performance Model
- Algorithms
- Environment Setup and Experiment Results
- HiSparse: High-Performance Sparse Linear Algebra on HBM-Equipped FPGAs

Introduction of SpMV

- What SpMV is
  - Computes $y = A \cdot x$, where A is sparse ($\ll n^2$ non-zeros)
  - Memory-bound: arithmetic intensity is typically ≤ 1 FLOP / 12 B
  - Core of graph analytics, iterative solvers, recommendation engines
- Why it is hard to accelerate
  - Irregular memory access – non-zeros arrive with random column indices → cache/TLB misses on CPUs/GPUs
  - Bandwidth limited – moving the values & indices dominates run time; extra FLOPs don’t help if DDR is the bottleneck
  - Load imbalance – row lengths vary widely; SIMD lanes idle on short rows

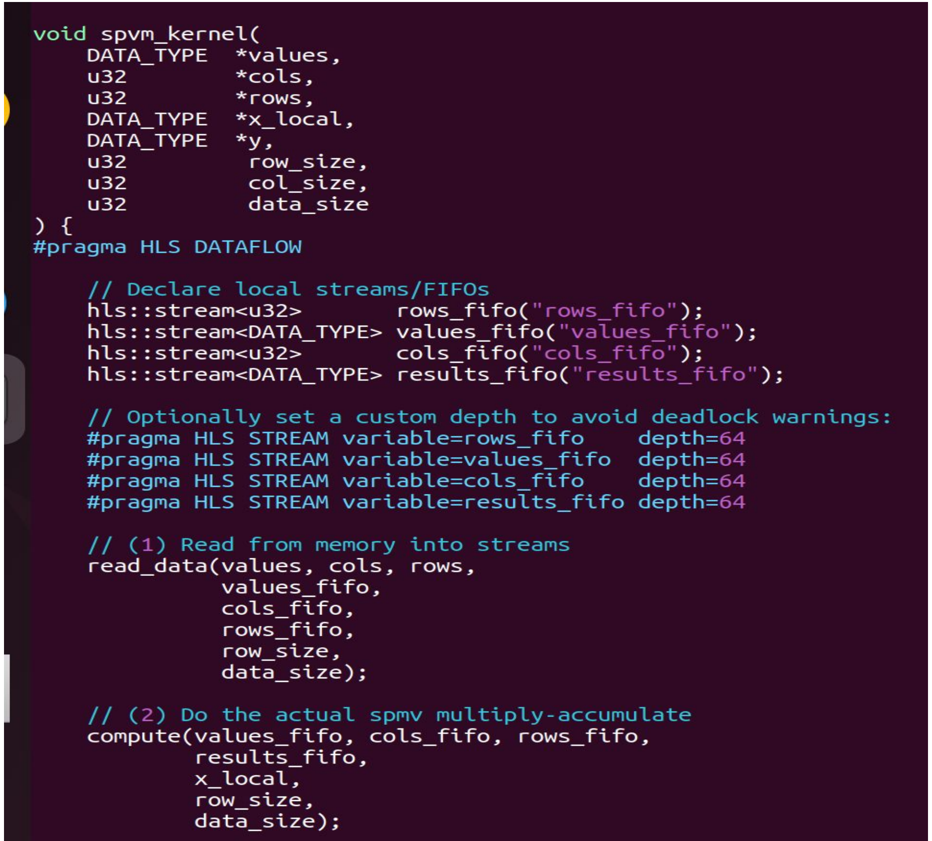
Formats for SpMV

- Various type of formats
  - CSR (Compressed Sparse Row)
  - COO (Coordinate list)
  - ELLPACK
  - BCSR (Block CSR)
- Choosing a format
  - CPU / FPGA streaming → CSR / BCSR (row-major access, dataflow friendly)
  - GPU kernels → ELL, HYB, JDS (warp-coalesced, uniform row length)
  - Stencil / banded PDE → DIA (few diagonals, minimal index storage)
  - Dynamic insertion → COO (simple append—later convert to CSR/ELL)
- Rule of thumb: balance indexing load vs. padding waste.
  - High hardware parallelism benefits from regular row length (ELL), while small FPGAs prefer minimal index traffic (CSR/DIA)

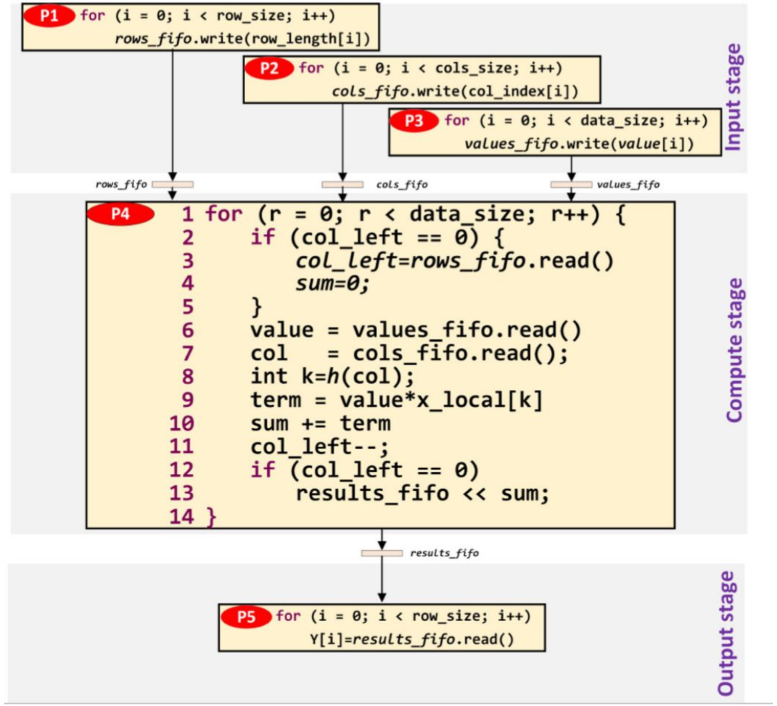


(a) (b) (c)

Structure of Streaming Data Engine for SpMV



```
void spvm_kernel(
    DATA_TYPE  *values,
    u32        *cols,
    u32        *rows,
    DATA_TYPE  *x_local,
    DATA_TYPE  *y,
    u32         row_size,
    u32         col_size,
    u32         data_size
) {
#pragma HLS DATAFLOW

    // Declare local streams/FIFOs
    hls::stream<u32>       rows_fifo("rows_fifo");
    hls::stream<DATA_TYPE> values_fifo("values_fifo");
    hls::stream<u32>       cols_fifo("cols_fifo");
    hls::stream<DATA_TYPE> results_fifo("results_fifo");

    // Optionally set a custom depth to avoid deadlock warnings:
    #pragma HLS STREAM variable=rows_fifo    depth=64
    #pragma HLS STREAM variable=values_fifo  depth=64
    #pragma HLS STREAM variable=cols_fifo    depth=64
    #pragma HLS STREAM variable=results_fifo depth=64

    // (1) Read from memory into streams
    read_data(values, cols, rows,
              values_fifo,
              cols_fifo,
              rows_fifo,
              row_size,
              data_size);

    // (2) Do the actual spmv multiply-accumulate
    compute(values_fifo, cols_fifo, rows_fifo,
            results_fifo,
            x_local,
            row_size,
            data_size);
```



```
for (i = 0; i < row_size; i++)
    rows_fifo.write(row_length[i])
```

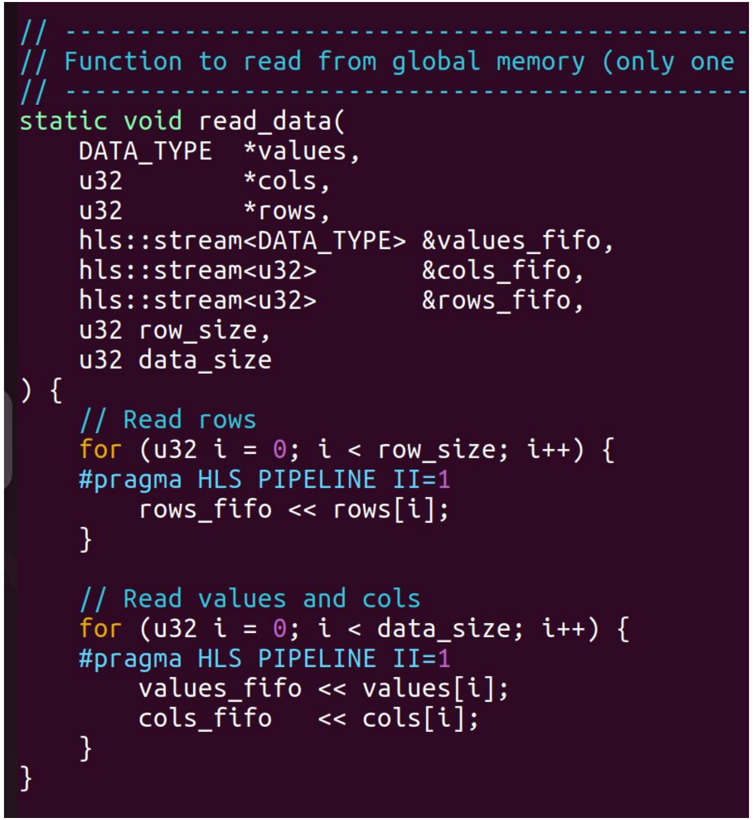
```
for (i = 0; i < cols_size; i++)
    cols_fifo.write(col_index[i])
```

```
for (i = 0; i < data_size; i++)
    values_fifo.write(value[i])
```

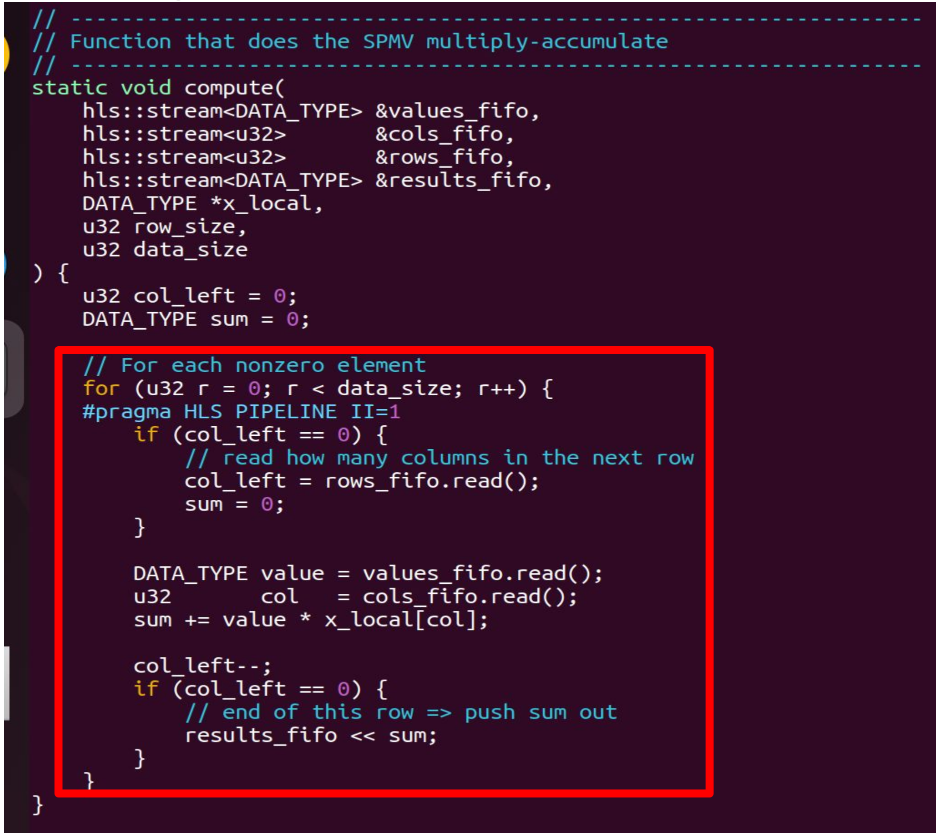
```
for (r = 0; r < data_size; r++) {
    if (col_left == 0) {
        col_left=rows_fifo.read()
        sum=0;
    }
    value = values_fifo.read()
    col   = cols_fifo.read();
    int k=h(col);
    term = value*x_local[k]
    sum += term
    col_left--;
    if (col_left == 0)
        results_fifo << sum;
}
```

```
for (i = 0; i < row_size; i++)
    Y[i]=results_fifo.read()
```

Structure of Streaming Data Engine for SpMV

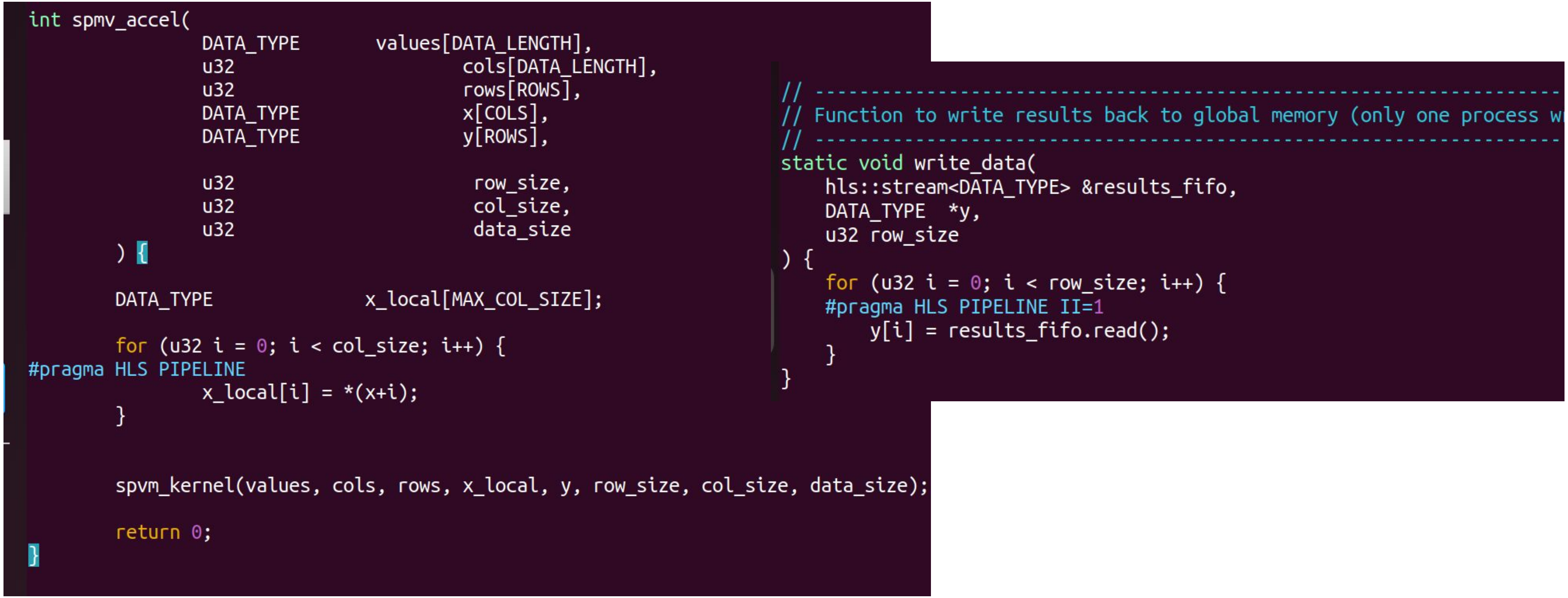


```
// -----------------------------------------------
// Function to read from global memory (only one
// -----------------------------------------------
static void read_data(
    DATA_TYPE  *values,
    u32        *cols,
    u32        *rows,
    hls::stream<DATA_TYPE> &values_fifo,
    hls::stream<u32>       &cols_fifo,
    hls::stream<u32>       &rows_fifo,
    u32 row_size,
    u32 data_size
) {
    // Read rows
    for (u32 i = 0; i < row_size; i++) {
    #pragma HLS PIPELINE II=1
        rows_fifo << rows[i];
    }

    // Read values and cols
    for (u32 i = 0; i < data_size; i++) {
    #pragma HLS PIPELINE II=1
        values_fifo << values[i];
        cols_fifo   << cols[i];
    }
}
```



```
// ----------------------------------------------------------------------
// Function that does the SPMV multiply-accumulate
// ----------------------------------------------------------------------
static void compute(
    hls::stream<DATA_TYPE> &values_fifo,
    hls::stream<u32>       &cols_fifo,
    hls::stream<u32>       &rows_fifo,
    hls::stream<DATA_TYPE> &results_fifo,
    DATA_TYPE *x_local,
    u32 row_size,
    u32 data_size
) {
    u32 col_left = 0;
    DATA_TYPE sum = 0;

    // For each nonzero element
    for (u32 r = 0; r < data_size; r++) {
    #pragma HLS PIPELINE II=1
        if (col_left == 0) {
            // read how many columns in the next row
            col_left = rows_fifo.read();
            sum = 0;
        }

        DATA_TYPE value = values_fifo.read();
        u32       col   = cols_fifo.read();
        sum += value * x_local[col];

        col_left--;
        if (col_left == 0) {
            // end of this row => push sum out
            results_fifo << sum;
        }
    }
}
```

Structure of Streaming Data Engine for SpMV



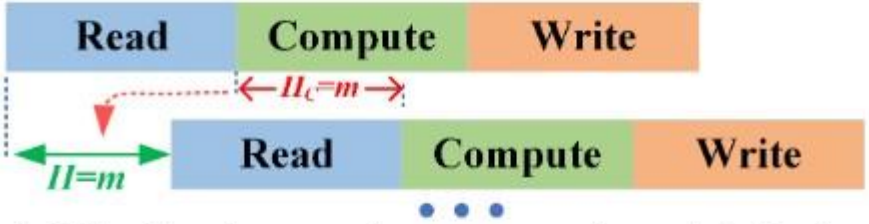
```
int spmv_accel(
                DATA_TYPE       values[DATA_LENGTH],
                u32                     cols[DATA_LENGTH],
                u32                     rows[ROWS],
                DATA_TYPE               x[COLS],
                DATA_TYPE               y[ROWS],

                u32                      row_size,
                u32                      col_size,
                u32                      data_size
        ) {

        DATA_TYPE               x_local[MAX_COL_SIZE];

        for (u32 i = 0; i < col_size; i++) {
#pragma HLS PIPELINE
                x_local[i] = *(x+i);
        }


        spvm_kernel(values, cols, rows, x_local, y, row_size, col_size, data_size);

        return 0;
}
```

```
// --------------------------------------------------------------------
// Function to write results back to global memory (only one process w
// --------------------------------------------------------------------
static void write_data(
    hls::stream<DATA_TYPE> &results_fifo,
    DATA_TYPE  *y,
    u32 row_size
) {
    for (u32 i = 0; i < row_size; i++) {
    #pragma HLS PIPELINE II=1
        y[i] = results_fifo.read();
    }
}
```

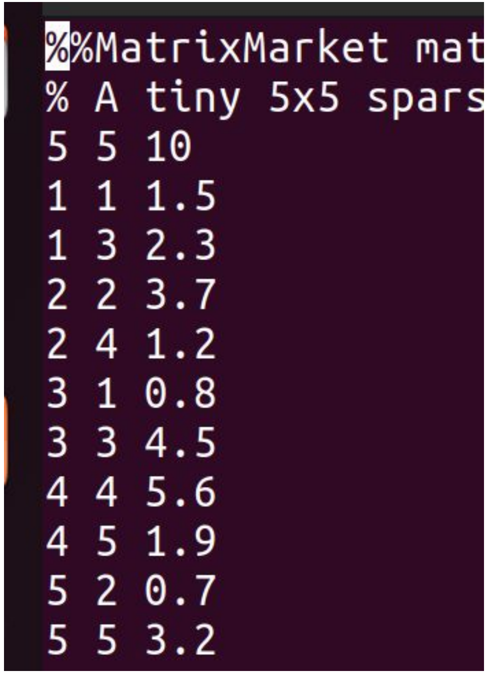
Performance Model & input matrix generation

- T = time required by algorithm + platform overhead
- t^{alg} = (m + nnz) / f
- t^{plat} = (II - 1) nnz / f
- T = (m + II * nnz) / f

$$T = t^{\text{alg}} + t^{\text{plat}} = \left(t^{\text{alg}}_{\text{ideal}} + t^{\text{alg}}_{\text{over}}\right) + \left(t^{\text{plat}}_{\text{lib}} + t^{\text{plat}}_{\text{hard}}\right).$$



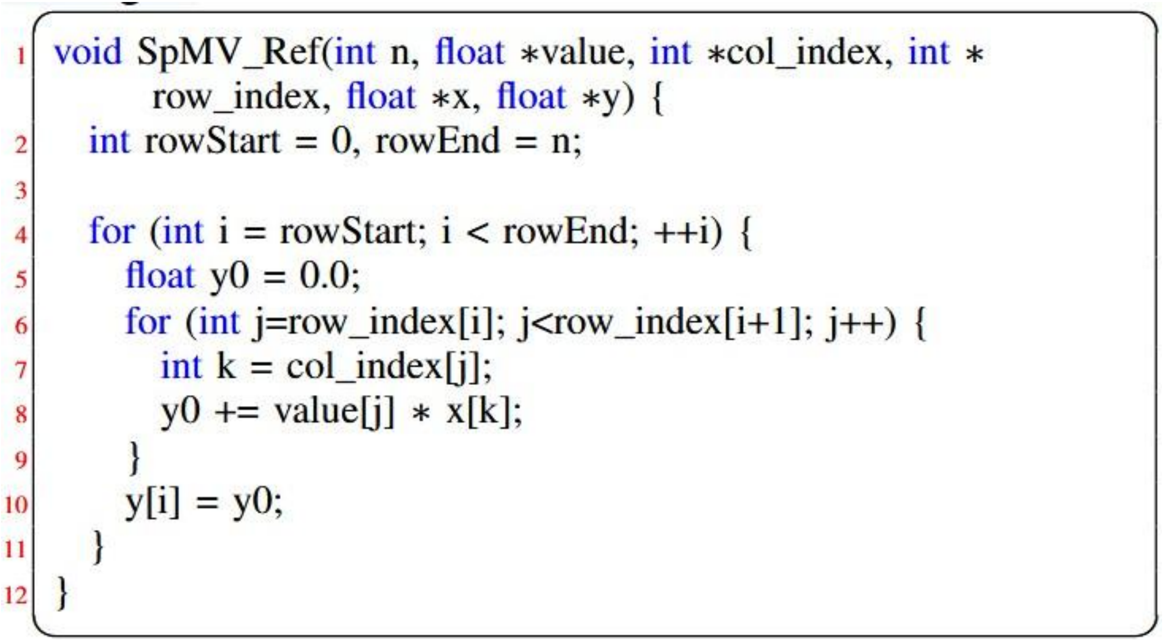
(c) Pipelined streaming computation with *II>1*



```
%%MatrixMarket mat
% A tiny 5x5 spars
5 5 10
1 1 1.5
1 3 2.3
2 2 3.7
2 4 1.2
3 1 0.8
3 3 4.5
4 4 5.6
4 5 1.9
5 2 0.7
5 5 3.2
```

Algorithm 1: Direct CSR SpMV

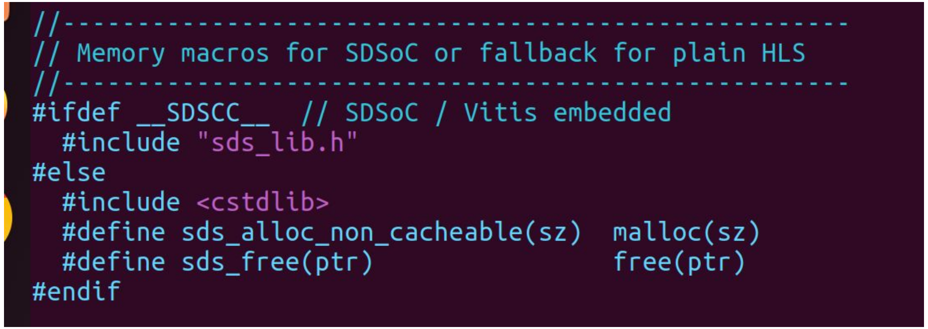
- Use software code as hls code
- The inner loop bounds are variables



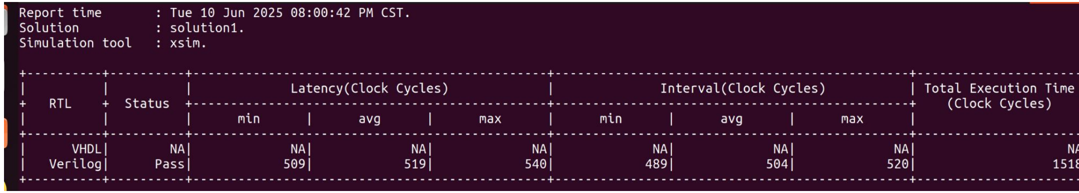
```
void SpMV_Ref(int n, float *value, int *col_index, int *
        row_index, float *x, float *y) {
  int rowStart = 0, rowEnd = n;

  for (int i = rowStart; i < rowEnd; ++i) {
    float y0 = 0.0;
    for (int j=row_index[i]; j<row_index[i+1]; j++) {
      int k = col_index[j];
      y0 += value[j] * x[k];
    }
    y[i] = y0;
  }
}
```

Algorithm 1: Naïve Stream Computing

- Problems for running csim
  - 1. Dataflow unavailable due to the potential deadlock
  - 2. SDS derepcated
- Problems for running cosim
  - 1. Segmentation Fault (buffer overflow)
- Solutions:
  - modulize the functions into load, compute and write
  - use cstdlib instead of sds_lib
  - When it comes to segmentation fault, situation becomes a bit tricky



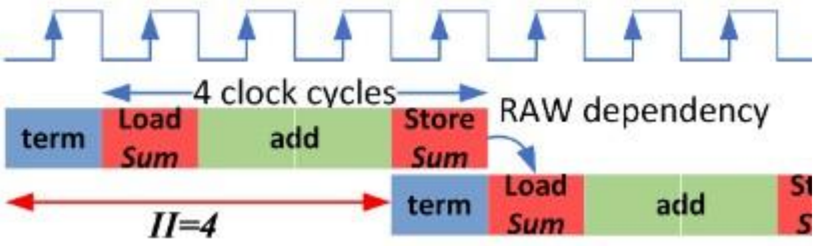
```
//------------------------------------------------------
// Memory macros for SDSoC or fallback for plain HLS
//------------------------------------------------------
#ifdef __SDSCC__   // SDSoC / Vitis embedded
  #include "sds_lib.h"
#else
  #include <cstdlib>
  #define sds_alloc_non_cacheable(sz)  malloc(sz)
  #define sds_free(ptr)                free(ptr)
#endif
```

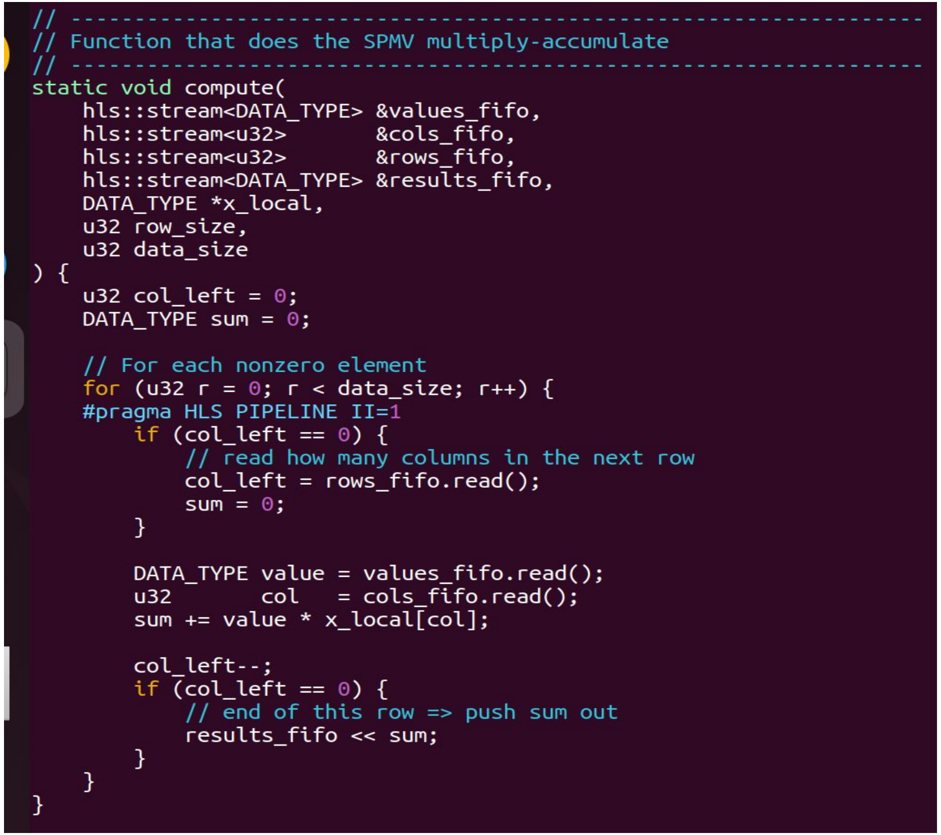


```
Report time        : Tue 10 Jun 2025 08:00:42 PM CST.
Solution           : solution1.
Simulation tool    : xsim.

+----------+----------+-----------------------------------------+-----------------------------------------+
|          |          |          Latency(Clock Cycles)          |          Interval(Clock Cycles)         | Total Execution Time
+   RTL    +  Status  +-----------------------------------------+-----------------------------------------+   (Clock Cycles)
|          |          |     min     |     avg     |     max     |     min     |     avg     |     max     |
+----------+----------+-----------------------------------------+-----------------------------------------+-------------------
|      VHDL|        NA|           NA|           NA|           NA|           NA|           NA|           NA|                 NA
|   Verilog|      Pass|          509|          519|          540|          489|          504|          520|               1518
+----------+----------+-----------------------------------------+-----------------------------------------+-------------------
```

Algorithm 2: Naïve Stream Computing

- data in rows_fifo need to be preprocessed to MCSR if input matrix is in CSR format
- Read after write hazard in sum and col_left -> II != 1
- Bug in the code: pure 0 row will cause error

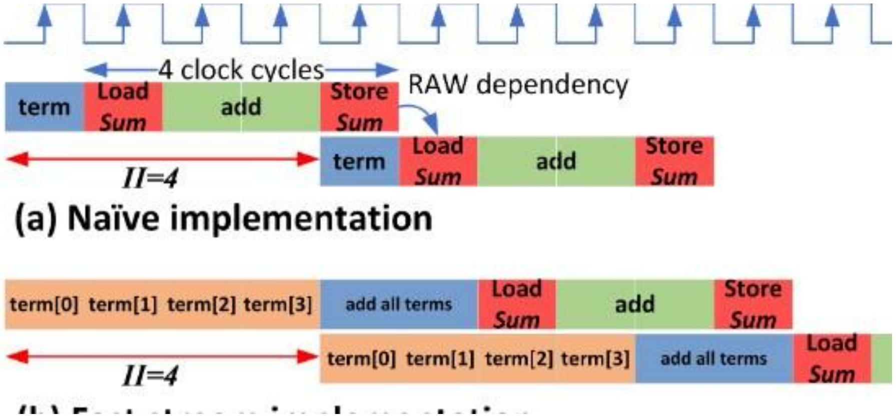




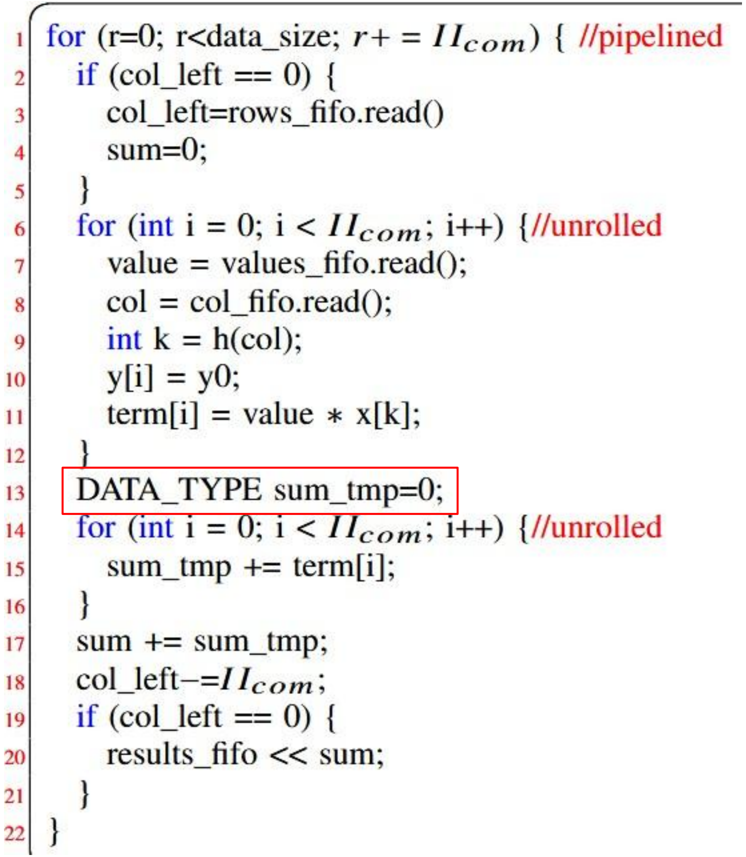
```
// ------------------------------------------------------------------
// Function that does the SPMV multiply-accumulate
// ------------------------------------------------------------------
static void compute(
    hls::stream<DATA_TYPE> &values_fifo,
    hls::stream<u32>       &cols_fifo,
    hls::stream<u32>       &rows_fifo,
    hls::stream<DATA_TYPE> &results_fifo,
    DATA_TYPE *x_local,
    u32 row_size,
    u32 data_size
) {
    u32 col_left = 0;
    DATA_TYPE sum = 0;

    // For each nonzero element
    for (u32 r = 0; r < data_size; r++) {
    #pragma HLS PIPELINE II=1
        if (col_left == 0) {
            // read how many columns in the next row
            col_left = rows_fifo.read();
            sum = 0;
        }

        DATA_TYPE value = values_fifo.read();
        u32       col   = cols_fifo.read();
        sum += value * x_local[col];

        col_left--;
        if (col_left == 0) {
            // end of this row => push sum out
            results_fifo << sum;
        }
    }
}
```

Algorithm 3: Fast Stream Computing

- Fast streaming : uroll the loop by II (need to pad each row to be a multiple of II)
- Process II terms in one iteration ~ II = 1 (Cons: II times of MAC circuit)



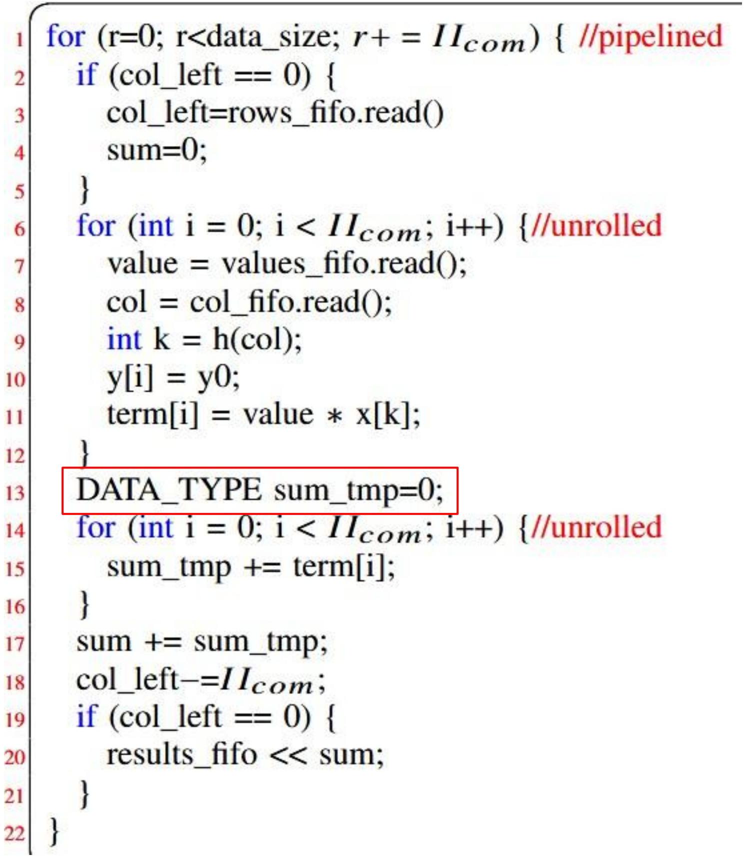
(a) Naïve implementation



```
for (r=0; r<data_size; r+ = II_com) { //pipelined
  if (col_left == 0) {
    col_left=rows_fifo.read()
    sum=0;
  }
  for (int i = 0; i < II_com; i++) {//unrolled
    value = values_fifo.read();
    col = col_fifo.read();
    int k = h(col);
    y[i] = y0;
    term[i] = value * x[k];
  }
  DATA_TYPE sum_tmp=0;
  for (int i = 0; i < II_com; i++) {//unrolled
    sum_tmp += term[i];
  }
  sum += sum_tmp;
  col_left−=II_com;
  if (col_left == 0) {
    results_fifo << sum;
  }
}
```

Algorithm 3: Fast Stream Computing

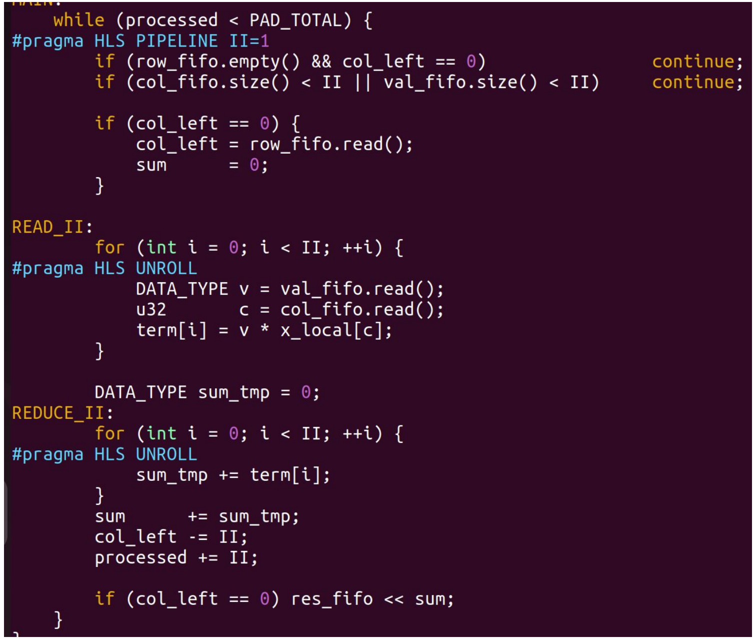
- What if uroll factor (UR) > II ?
  - The bottleneck will become fifo since only one element can be read/write in one cycle.
  - Using stream<vector> can solve the above issue, but large UR still cause timing violation or increas II. In conclusion, speeding up computation by increasing UR is not scalable



```
for (r=0; r<data_size; $r+ = II_{com}$) { //pipelined
  if (col_left == 0) {
    col_left=rows_fifo.read()
    sum=0;
  }
  for (int i = 0; i < $II_{com}$; i++) {//unrolled
    value = values_fifo.read();
    col = col_fifo.read();
    int k = h(col);
    y[i] = y0;
    term[i] = value $*$ x[k];
  }
  DATA_TYPE sum_tmp=0;
  for (int i = 0; i < $II_{com}$; i++) {//unrolled
    sum_tmp += term[i];
  }
  sum += sum_tmp;
  col_left−=$II_{com}$;
  if (col_left == 0) {
    results_fifo << sum;
  }
}
```

Reduced-Port Stream Computing

- Technique involved: combine the row and column into one array
- Cons: Time complexity = O(nnz + m) if data is processed on chip



```
MAIN:
    while (processed < PAD_TOTAL) {
#pragma HLS PIPELINE II=1
        if (row_fifo.empty() && col_left == 0)              continue;
        if (col_fifo.size() < II || val_fifo.size() < II)     continue;

        if (col_left == 0) {
            col_left = row_fifo.read();
            sum      = 0;
        }

READ_II:
        for (int i = 0; i < II; ++i) {
#pragma HLS UNROLL
            DATA_TYPE v = val_fifo.read();
            u32       c = col_fifo.read();
            term[i] = v * x_local[c];
        }

        DATA_TYPE sum_tmp = 0;
REDUCE_II:
        for (int i = 0; i < II; ++i) {
#pragma HLS UNROLL
            sum_tmp += term[i];
        }
        sum      += sum_tmp;
        col_left -= II;
        processed += II;

        if (col_left == 0) res_fifo << sum;
    }
```

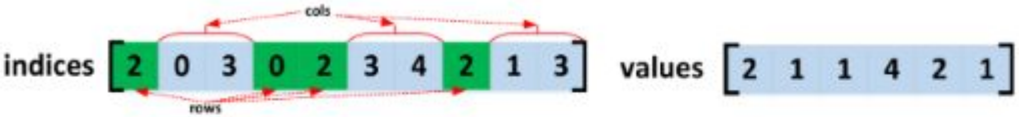


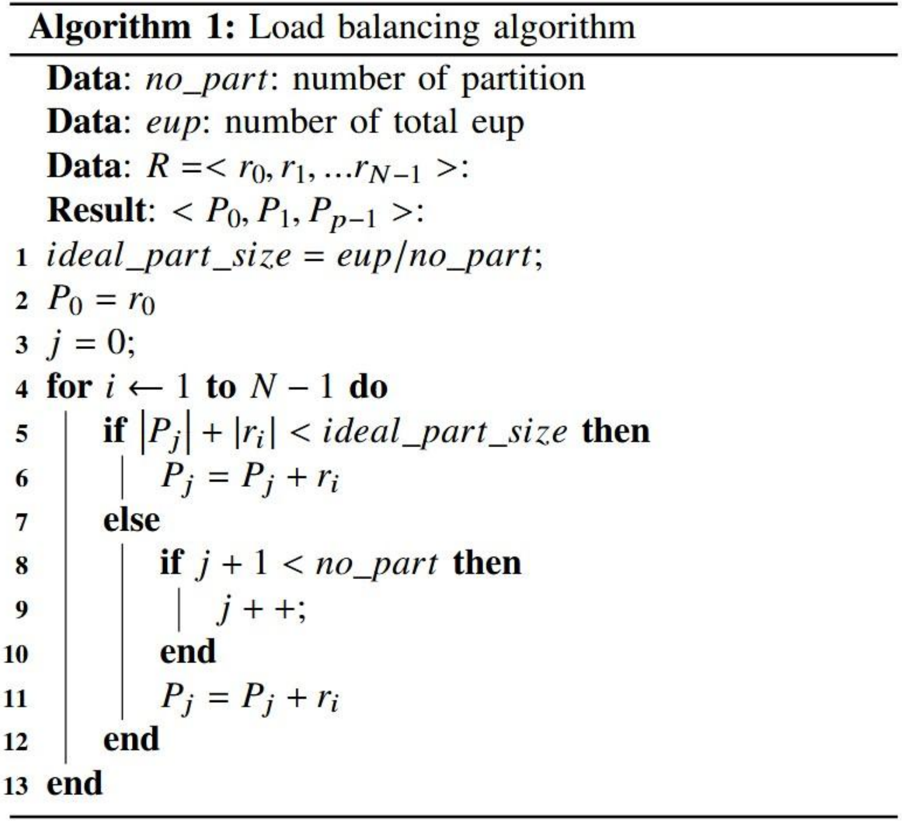
Fig. 7: Two-port streaming CSR

Multiport Stream Computing

- Suppose FPGA has P memory ports, each having B bits and value & indices have g and h bits
- The rows to be process at the same time $p <= P * B / (g + h)$

Algorithm 4: Load Balancing

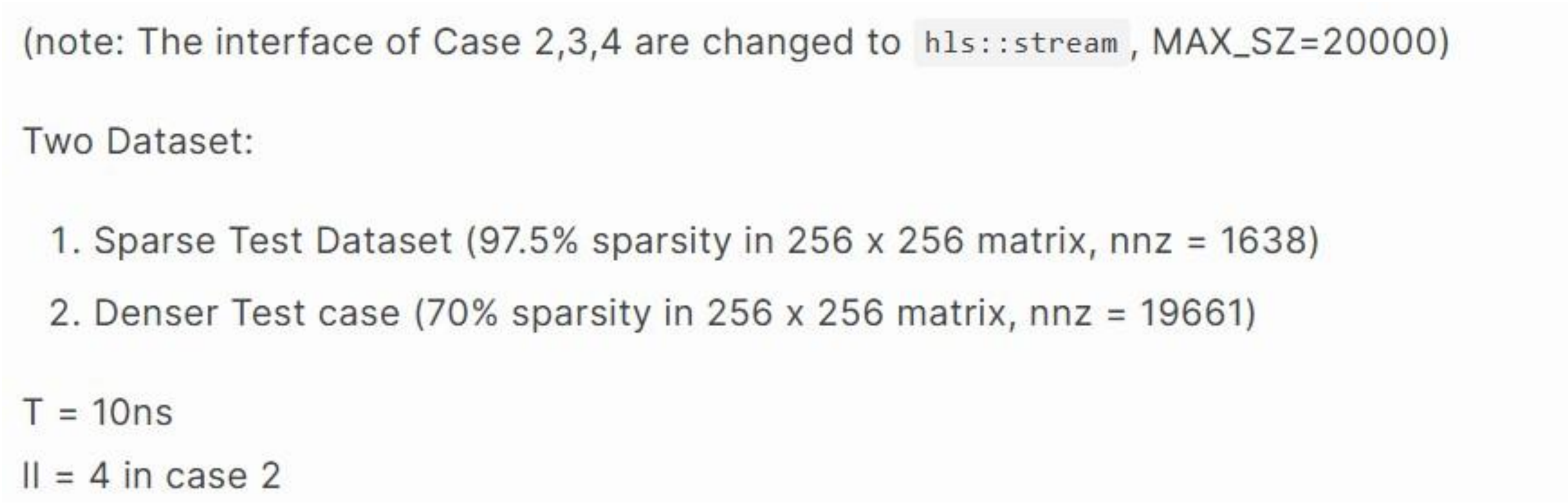
- eup : non-zero elements after padding
- Ideal partition : equally loaded to P computing unit
- Greedy partition : loaded rows to P computing units with same eup as equal as possible



Algorithm 1: Load balancing algorithm

```
Data: $no\_part$: number of partition
Data: $eup$: number of total eup
Data: $R =< r_0, r_1, ...r_{N-1} >$:
Result: $< P_0, P_1, P_{p-1} >$:
$ideal\_part\_size = eup/no\_part$;
$P_0 = r_0$
$j = 0$;
for $i \leftarrow 1$ to $N - 1$ do
    if $|P_j| + |r_i| < ideal\_part\_size$ then
        $P_j = P_j + r_i$
    else
        if $j + 1 < no\_part$ then
            $j++$;
        end
        $P_j = P_j + r_i$
    end
end
```

Experiment Setup and Results



(note: The interface of Case 2,3,4 are changed to `hls::stream`, MAX_SZ=20000)

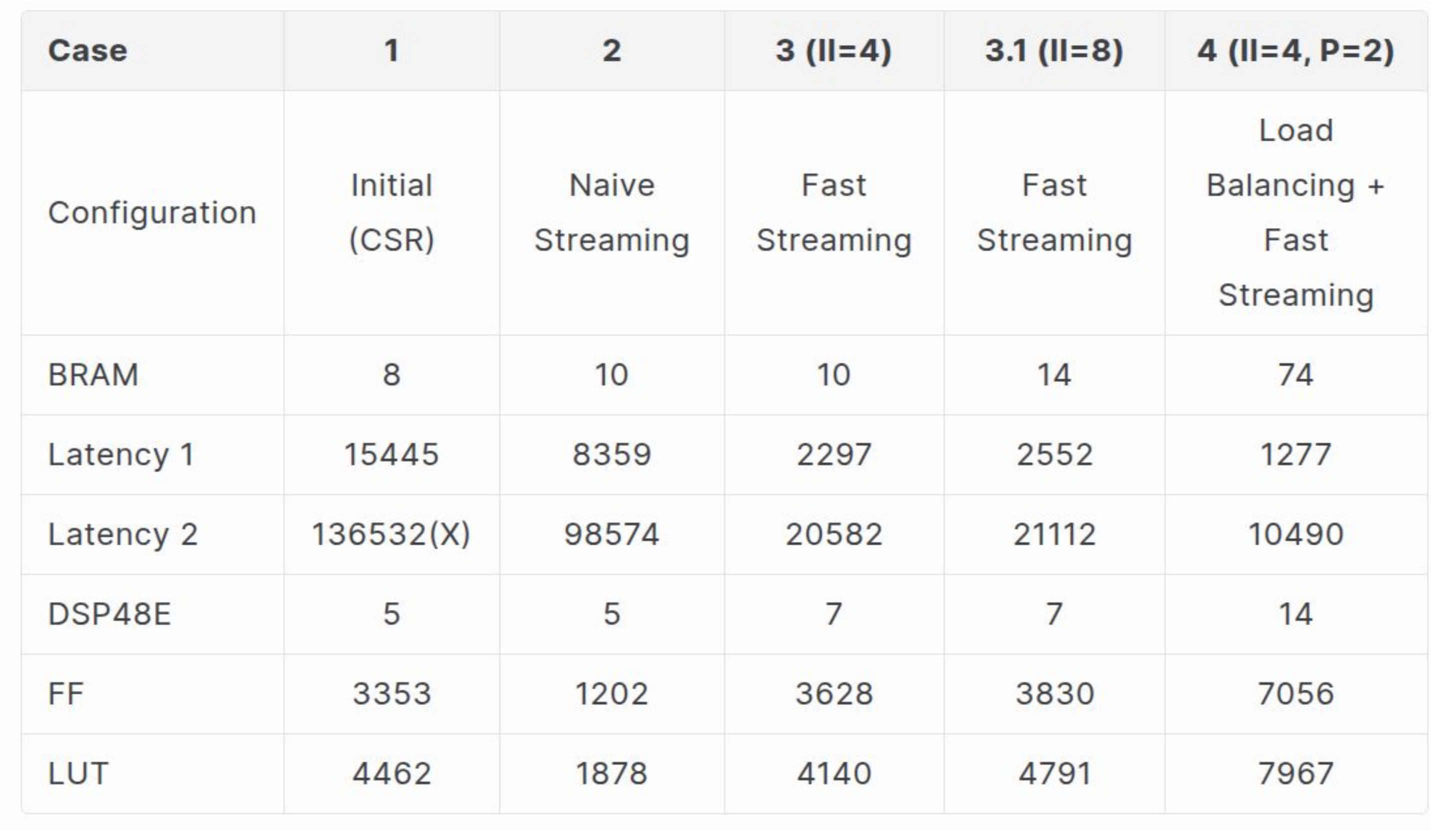
Two Dataset:

1. Sparse Test Dataset (97.5% sparsity in 256 x 256 matrix, nnz = 1638)
2. Denser Test case (70% sparsity in 256 x 256 matrix, nnz = 19661)

T = 10ns

II = 4 in case 2

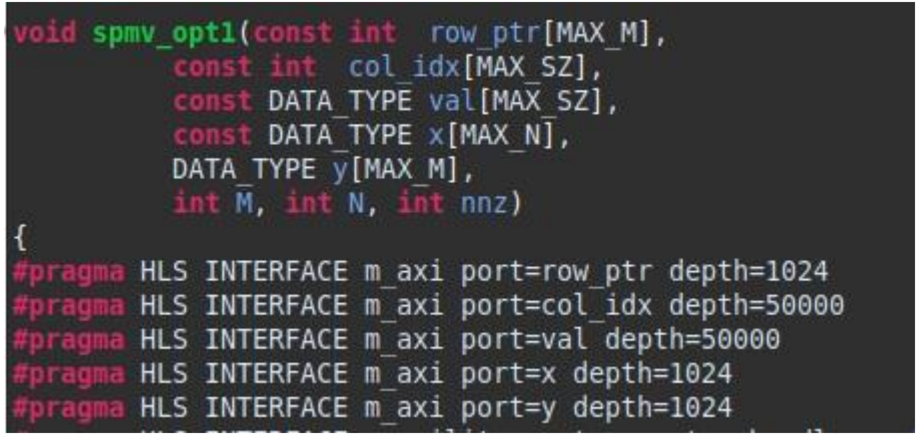
Experiment Setup and Results



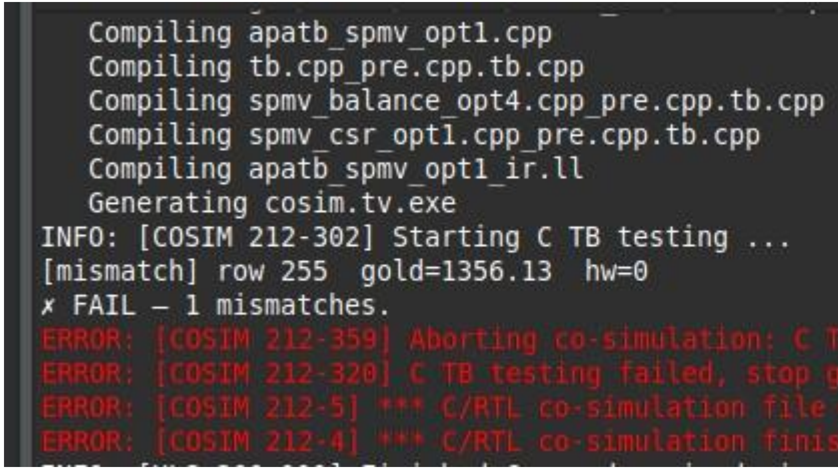
Case	1	2	3 (II=4)	3.1 (II=8)	4 (II=4, P=2)
Configuration	Initial (CSR)	Naive Streaming	Fast Streaming	Fast Streaming	Load Balancing + Fast Streaming
BRAM	8	10	10	14	74
Latency 1	15445	8359	2297	2552	1277
Latency 2	136532(X)	98574	20582	21112	10490
DSP48E	5	5	7	7	14
FF	3353	1202	3628	3830	7056
LUT	4462	1878	4140	4791	7967

Experiment Setup and Results

- Case 1 co-sim mismatched at last row in larger dataset
- Array as hls function parameter
    - Hard to meet dataflow requirement : need to bind each port to different memory port
    - Use ap_fifo as interface

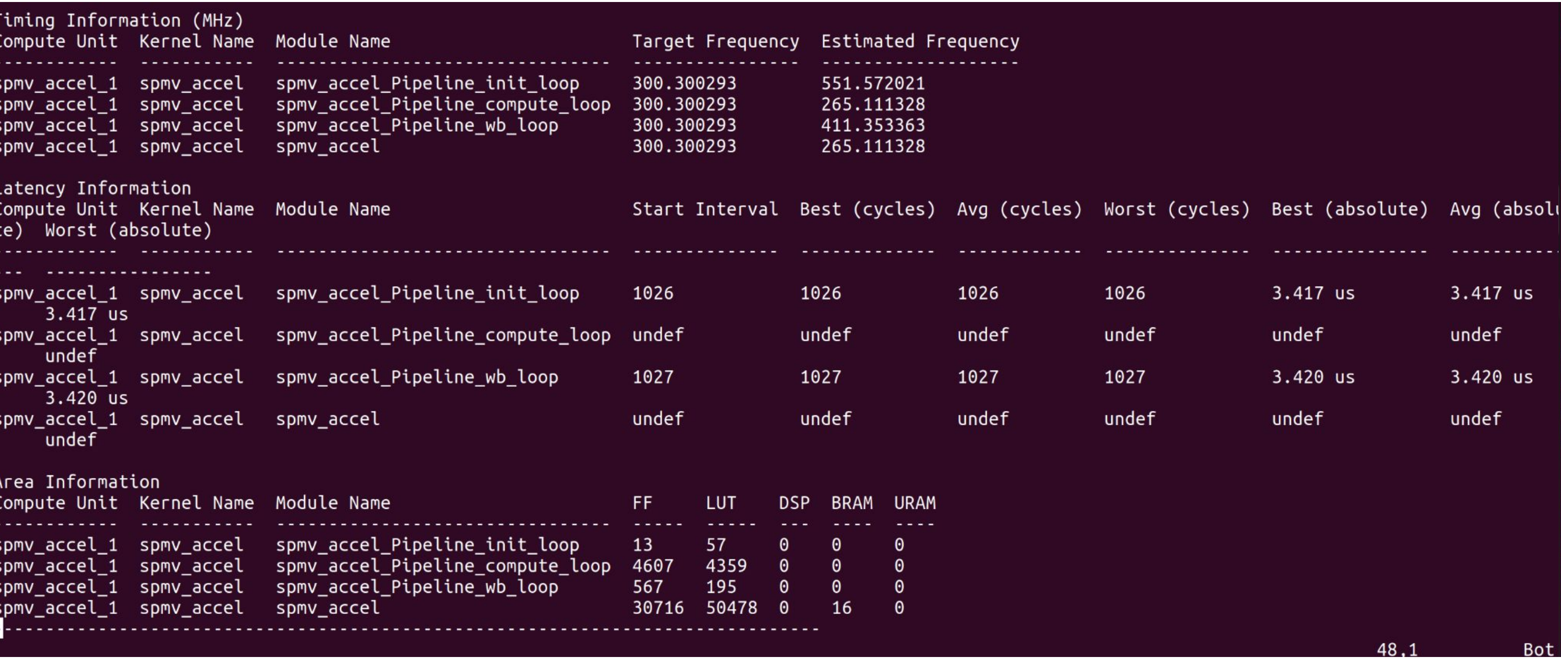


```
void spmv_opt1(const int  row_ptr[MAX_M],
          const int  col_idx[MAX_SZ],
          const DATA_TYPE val[MAX_SZ],
          const DATA_TYPE x[MAX_N],
          DATA_TYPE y[MAX_M],
          int M, int N, int nnz)
{
#pragma HLS INTERFACE m_axi port=row_ptr depth=1024
#pragma HLS INTERFACE m_axi port=col_idx depth=50000
#pragma HLS INTERFACE m_axi port=val depth=50000
#pragma HLS INTERFACE m_axi port=x depth=1024
#pragma HLS INTERFACE m_axi port=y depth=1024
```

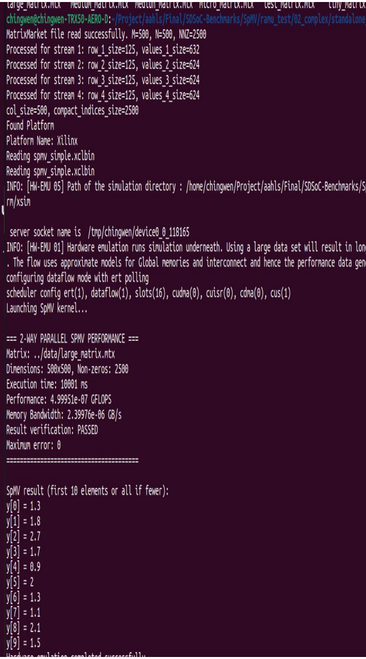


```
   Compiling apatb_spmv_opt1.cpp
   Compiling tb.cpp_pre.cpp.tb.cpp
   Compiling spmv_balance_opt4.cpp_pre.cpp.tb.cpp
   Compiling spmv_csr_opt1.cpp_pre.cpp.tb.cpp
   Compiling apatb_spmv_opt1_ir.ll
   Generating cosim.tv.exe
INFO: [COSIM 212-302] Starting C TB testing ...
[mismatch] row 255  gold=1356.13  hw=0
✗ FAIL — 1 mismatches.
ERROR: [COSIM 212-359] Aborting co-simulation: C T
ERROR: [COSIM 212-320] C TB testing failed, stop g
ERROR: [COSIM 212-5] *** C/RTL co-simulation file 
ERROR: [COSIM 212-4] *** C/RTL co-simulation finis
```

Hardware Emulation Result for simple SpMV



```
iming Information (MHz)
ompute Unit  Kernel Name  Module Name                        Target Frequency  Estimated Frequency
-----------  -----------  ---------------------------------  ----------------  -------------------
pmv_accel_1  spmv_accel   spmv_accel_Pipeline_init_loop      300.300293        551.572021
pmv_accel_1  spmv_accel   spmv_accel_Pipeline_compute_loop   300.300293        265.111328
pmv_accel_1  spmv_accel   spmv_accel_Pipeline_wb_loop        300.300293        411.353363
pmv_accel_1  spmv_accel   spmv_accel                         300.300293        265.111328

atency Information
ompute Unit  Kernel Name  Module Name                        Start Interval  Best (cycles)  Avg (cycles)  Worst (cycles)  Best (absolute)  Avg (absolu
te)  Worst (absolute)
-----------  -----------  ---------------------------------  --------------  -------------  ------------  --------------  ---------------  ----------
--  ----------------
pmv_accel_1  spmv_accel   spmv_accel_Pipeline_init_loop      1026            1026           1026          1026            3.417 us         3.417 us
     3.417 us
pmv_accel_1  spmv_accel   spmv_accel_Pipeline_compute_loop   undef           undef          undef         undef           undef            undef
     undef
pmv_accel_1  spmv_accel   spmv_accel_Pipeline_wb_loop        1027            1027           1027          1027            3.420 us         3.420 us
     3.420 us
pmv_accel_1  spmv_accel   spmv_accel                         undef           undef          undef         undef           undef            undef
     undef

rea Information
ompute Unit  Kernel Name  Module Name                        FF     LUT    DSP  BRAM  URAM
-----------  -----------  ---------------------------------  -----  -----  ---  ----  ----
pmv_accel_1  spmv_accel   spmv_accel_Pipeline_init_loop      13     57     0    0     0
pmv_accel_1  spmv_accel   spmv_accel_Pipeline_compute_loop   4607   4359   0    0     0
pmv_accel_1  spmv_accel   spmv_accel_Pipeline_wb_loop        567    195    0    0     0
pmv_accel_1  spmv_accel   spmv_accel                         30716  50478  0    16    0
---------------------------------------------------------------------------------------------
                                                                                          48,1          Bot
```



```
large_matrix.mtx   medium_matrix.mtx  medium_matrix.mtx  micro_matrix.mtx   test_matrix.mtx    tiny_matrix
chingwen@chingwen-TRX50-AERO-D:~/Project/aahls/Final/SDSoC-Benchmarks/SpMV/ramu_test/02_complex/standalone
MatrixMarket file read successfully. M=500, N=500, NNZ=2500
Processed for stream 1: row_1_size=125, values_1_size=632
Processed for stream 2: row_2_size=125, values_2_size=624
Processed for stream 3: row_3_size=125, values_3_size=624
Processed for stream 4: row_4_size=125, values_4_size=624
col_size=500, compact_indices_size=2500
Found Platform
Platform Name: Xilinx
Reading spmv_simple.xclbin
Reading spmv_simple.xclbin
INFO: [HW-EMU 05] Path of the simulation directory : /home/chingwen/Project/aahls/Final/SDSoC-Benchmarks/S
rm/xsim

 server socket name is  /tmp/chingwen/device0_0_118165
INFO: [HW-EMU 01] Hardware emulation runs simulation underneath. Using a large data set will result in lon
. The flow uses approximate models for Global memories and interconnect and hence the performance data gen
configuring dataflow mode with ert polling
scheduler config ert(1), dataflow(1), slots(16), cudma(0), cuisr(0), cdma(0), cus(1)
Launching SpMV kernel...

=== 2-WAY PARALLEL SPMV PERFORMANCE ===
Matrix: ../data/large_matrix.mtx
Dimensions: 500x500, Non-zeros: 2500
Execution time: 10001 ms
Performance: 4.99951e-07 GFLOPS
Memory Bandwidth: 2.39976e-06 GB/s
Result verification: PASSED
Maximum error: 0
========================================


SpMV result (first 10 elements or all if fewer):
y[0] = 1.3
y[1] = 1.8
y[2] = 2.7
y[3] = 1.7
y[4] = 0.9
y[5] = 2
y[6] = 1.3
y[7] = 1.1
y[8] = 2.1
y[9] = 1.5
Hardware emulation completed successfully
```

Experiment Setup and Results

- Configurations:
- Challenges:
- Results:

	Numbers of cycles	Cycle time	Resource utilization

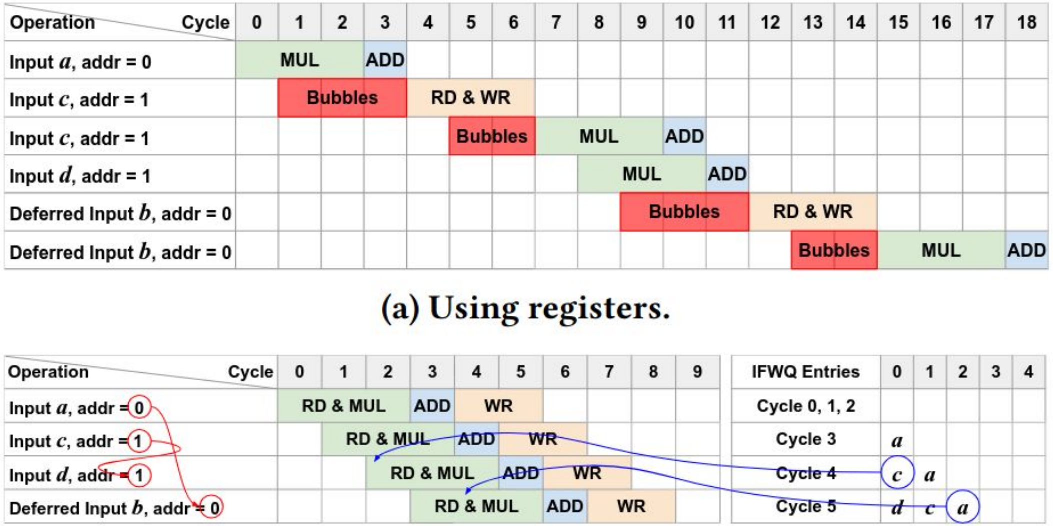
HiSparse: High-Performance Sparse Linear Algebra on HBM-Equipped FPGAs

- Challenges desired to solve
    - Timing closure on a multi-die FPGA
    - Irregularity in the compute pattern of SpMV causes bank conflicts and carried data dependencie
    - Multi-cycle for floating-point number in addition stage complicates the case of RAW hazard
- Proposed solutions
    - Split kernel + relay unit
    - load-store forwarding mechanism
    - Row interleaving + stall unit

HiSparse: Shared Vector Buffer with Shuffle Unit

HiSparse: Pipelined PE with Load-Store Forwarding

- In-flight write queue (IFWQ)
- Dependence resolution logic



Operation \ Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Input a, addr = 0	MUL			ADD															
Input c, addr = 1		Bubbles			RD & WR														
Input c, addr = 1						Bubbles		MUL			ADD								
Input d, addr = 1									MUL			ADD							
Deferred Input b, addr = 0										Bubbles			RD & WR						
Deferred Input b, addr = 0														Bubbles		MUL			ADD

(a) Using registers.

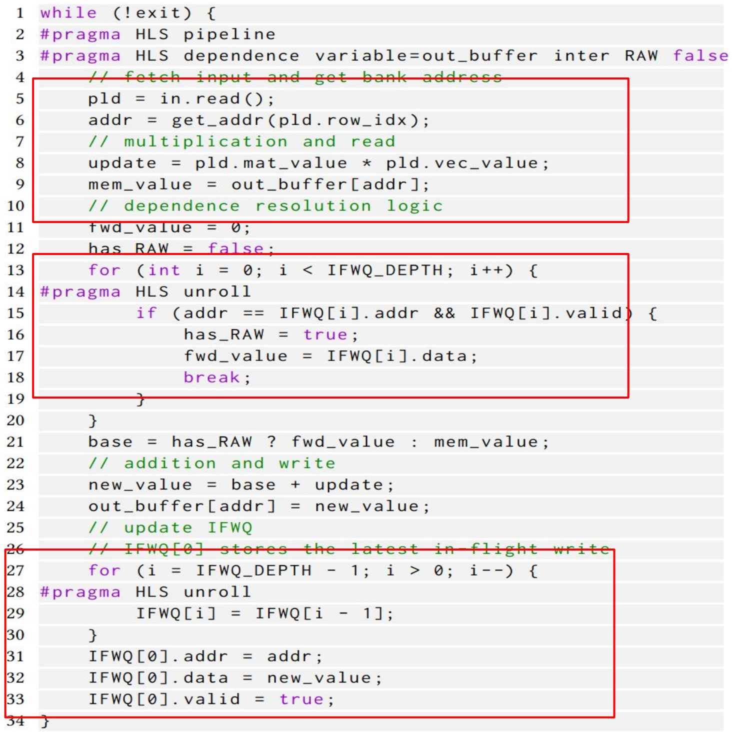
Operation \ Cycle	0	1	2	3	4	5	6	7	8	9
Input a, addr = 0	RD & MUL			ADD	WR					
Input c, addr = 1		RD & MUL			ADD	WR				
Input d, addr = 1			RD & MUL			ADD	WR			
Deferred Input b, addr = 0				RD & MUL			ADD	WR		

IFWQ Entries	0	1	2	3	4
Cycle 0, 1, 2					
Cycle 3	a				
Cycle 4	c	a			
Cycle 5	d	c	a		

(b) Using load-store forwarding — Red arrows indicate the RAW dependencies. Blue arrows indicate the data forwarding to resolve dependencies.

HiSparse: Pipelined PE with Load-Store Forwarding

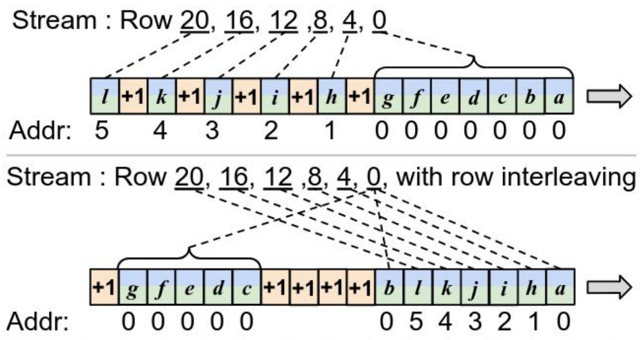
- Stage 1: Get address from the playload
- Stage 2: Check the RAW hazard
- Stage 3: Update the IFWQ
- Usable scenario: when number of cycle of the addition logic is equal to one



```
while (!exit) {
#pragma HLS pipeline
#pragma HLS dependence variable=out_buffer inter RAW false
    // fetch input and get bank address
    pld = in.read();
    addr = get_addr(pld.row_idx);
    // multiplication and read
    update = pld.mat_value * pld.vec_value;
    mem_value = out_buffer[addr];
    // dependence resolution logic
    fwd_value = 0;
    has_RAW = false;
    for (int i = 0; i < IFWQ_DEPTH; i++) {
#pragma HLS unroll
        if (addr == IFWQ[i].addr && IFWQ[i].valid) {
            has_RAW = true;
            fwd_value = IFWQ[i].data;
            break;
        }
    }
    base = has_RAW ? fwd_value : mem_value;
    // addition and write
    new_value = base + update;
    out_buffer[addr] = new_value;
    // update IFWQ
    // IFWQ[0] stores the latest in-flight write
    for (i = IFWQ_DEPTH - 1; i > 0; i--) {
#pragma HLS unroll
        IFWQ[i] = IFWQ[i - 1];
    }
    IFWQ[0].addr = addr;
    IFWQ[0].data = new_value;
    IFWQ[0].valid = true;
}
```

HiSparse: Floating-Point Implementation

- Inter-iteration carried dependencies
- Naive way: duplicate the output buffer to multiple partial buffer
- Suggested way: Row interleaving



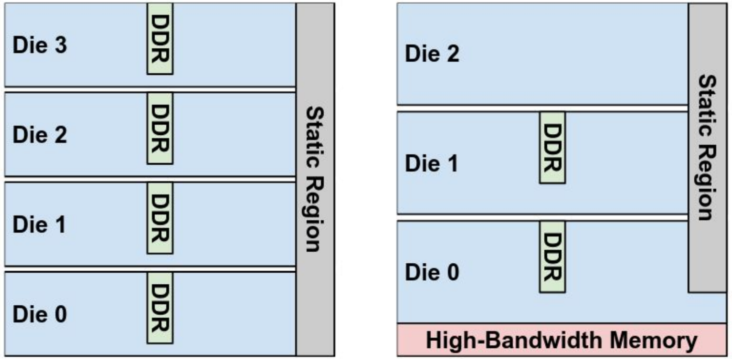
Stream : Row 20, 16, 12 ,8, 4, 0
l +1 k +1 j +1 i +1 h +1 g f e d c b a
Addr: 5 4 3 2 1 0 0 0 0 0 0 0
Stream : Row 20, 16, 12 ,8, 4, 0, with row interleaving
+1 g f e d c +1 +1 +1 +1 b l k j i h a
Addr: 0 0 0 0 0 0 5 4 3 2 1 0

Operation Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Input a, addr = 0	RD	ADD			WR													
Input h, addr = 1		RD	ADD			WR												
Input i, addr = 2			RD	ADD			WR											
Input j, addr = 3				RD	ADD			WR										
Input k, addr = 4					RD	ADD			WR									
Input l, addr = 5						RD	ADD			WR								
Input b, addr = 0							RD	ADD			WR							
Input c, addr = 0								Bubbles					RD	ADD			WR	

Figure 10: PE with row interleaving — Blue and red arrows indicate the dependencies resolved by row interleaving and stalling, respectively. "Addr" indicates the output buffer bank address.

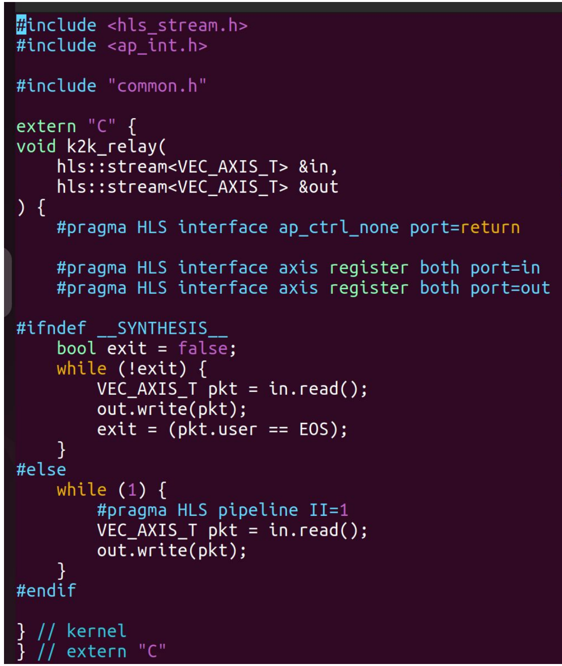
HiSparse: Timing Closure on Multi-Die FPGAS

- Concepts of die and SLR (Super logic regions)
- Split kernel vs monolithic kernel
- Relay Unit
- create_pblock <k0>; resize_pblock … -add_slrs {SLR0}



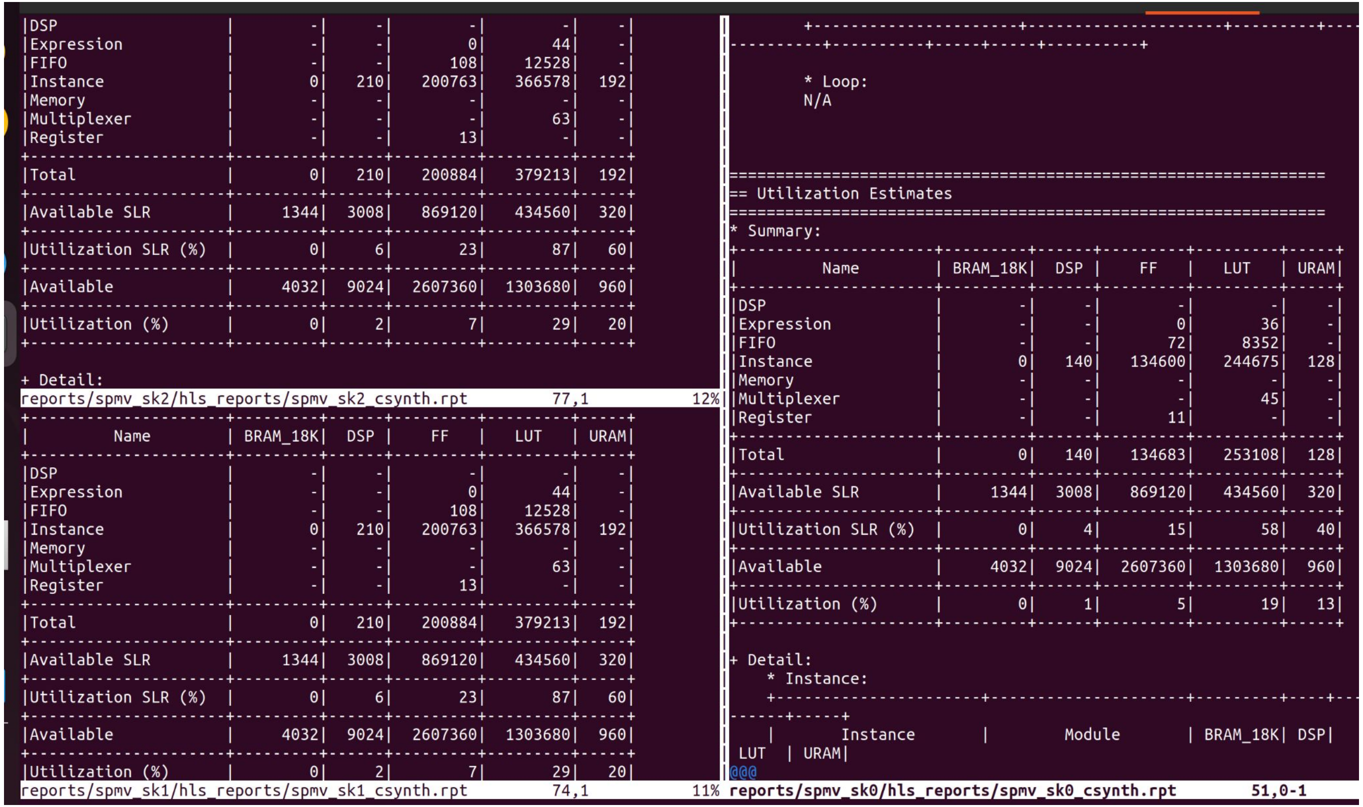
(a) U250. (b) U280.

Figure 1: Comparison between Xilinx Alveo U250 and U280.



```
#include <hls_stream.h>
#include <ap_int.h>

#include "common.h"

extern "C" {
void k2k_relay(
    hls::stream<VEC_AXIS_T> &in,
    hls::stream<VEC_AXIS_T> &out
) {
    #pragma HLS interface ap_ctrl_none port=return

    #pragma HLS interface axis register both port=in
    #pragma HLS interface axis register both port=out

#ifndef __SYNTHESIS__
    bool exit = false;
    while (!exit) {
        VEC_AXIS_T pkt = in.read();
        out.write(pkt);
        exit = (pkt.user == EOS);
    }
#else
    while (1) {
        #pragma HLS pipeline II=1
        VEC_AXIS_T pkt = in.read();
        out.write(pkt);
    }
#endif

} // kernel
} // extern "C"
```

Reports for Hardware emulation and Synthesis



```
|DSP                  |        -|    -|        -|        -|    -|
|Expression           |        -|    -|        0|       44|    -|
|FIFO                 |        -|    -|      108|    12528|    -|
|Instance             |        0|  210|   200763|   366578|  192|
|Memory               |        -|    -|        -|        -|    -|
|Multiplexer          |        -|    -|        -|       63|    -|
|Register             |        -|    -|       13|        -|    -|
+---------------------+---------+-----+---------+---------+-----+
|Total                |        0|  210|   200884|   379213|  192|
+---------------------+---------+-----+---------+---------+-----+
|Available SLR        |     1344| 3008|   869120|   434560|  320|
+---------------------+---------+-----+---------+---------+-----+
|Utilization SLR (%)  |        0|    6|       23|       87|   60|
+---------------------+---------+-----+---------+---------+-----+
|Available            |     4032| 9024|  2607360|  1303680|  960|
+---------------------+---------+-----+---------+---------+-----+
|Utilization (%)      |        0|    2|        7|       29|   20|
+---------------------+---------+-----+---------+---------+-----+

+ Detail:
reports/spmv_sk2/hls_reports/spmv_sk2_csynth.rpt          77,1          12%
```

```
+---------------------+---------+-----+---------+---------+-----+
|         Name        | BRAM_18K| DSP |    FF   |   LUT   | URAM|
+---------------------+---------+-----+---------+---------+-----+
|DSP                  |        -|    -|        -|        -|    -|
|Expression           |        -|    -|        0|       44|    -|
|FIFO                 |        -|    -|      108|    12528|    -|
|Instance             |        0|  210|   200763|   366578|  192|
|Memory               |        -|    -|        -|        -|    -|
|Multiplexer          |        -|    -|        -|       63|    -|
|Register             |        -|    -|       13|        -|    -|
+---------------------+---------+-----+---------+---------+-----+
|Total                |        0|  210|   200884|   379213|  192|
+---------------------+---------+-----+---------+---------+-----+
|Available SLR        |     1344| 3008|   869120|   434560|  320|
+---------------------+---------+-----+---------+---------+-----+
|Utilization SLR (%)  |        0|    6|       23|       87|   60|
+---------------------+---------+-----+---------+---------+-----+
|Available            |     4032| 9024|  2607360|  1303680|  960|
+---------------------+---------+-----+---------+---------+-----+
|Utilization (%)      |        0|    2|        7|       29|   20|
reports/spmv_sk1/hls_reports/spmv_sk1_csynth.rpt          74,1          11%
```

```
        +----------------------+----------------------+---------+---
----------+---------+----+----+----------+

        * Loop:
        N/A



================================================================
== Utilization Estimates
================================================================
* Summary:
+---------------------+---------+-----+---------+---------+-----+
|         Name        | BRAM_18K| DSP |    FF   |   LUT   | URAM|
+---------------------+---------+-----+---------+---------+-----+
|DSP                  |        -|    -|        -|        -|    -|
|Expression           |        -|    -|        0|       36|    -|
|FIFO                 |        -|    -|       72|     8352|    -|
|Instance             |        0|  140|   134600|   244675|  128|
|Memory               |        -|    -|        -|        -|    -|
|Multiplexer          |        -|    -|        -|       45|    -|
|Register             |        -|    -|       11|        -|    -|
+---------------------+---------+-----+---------+---------+-----+
|Total                |        0|  140|   134683|   253108|  128|
+---------------------+---------+-----+---------+---------+-----+
|Available SLR        |     1344| 3008|   869120|   434560|  320|
+---------------------+---------+-----+---------+---------+-----+
|Utilization SLR (%)  |        0|    4|       15|       58|   40|
+---------------------+---------+-----+---------+---------+-----+
|Available            |     4032| 9024|  2607360|  1303680|  960|
+---------------------+---------+-----+---------+---------+-----+
|Utilization (%)      |        0|    1|        5|       19|   13|
+---------------------+---------+-----+---------+---------+-----+

+ Detail:
    * Instance:
    +----------------------+----------------------+---------+----+---
------+-----+
    |       Instance       |        Module        | BRAM_18K| DSP|
 LUT   | URAM|
@@@
reports/spmv_sk0/hls_reports/spmv_sk0_csynth.rpt          51,0-1
```