Towards High-Bandwidth-Utilization SpMV on FPGAs via Partial Vector Duplication

ASP-DAC 2023

Bowen Liu, Dajiang Liu*

Chongqing University

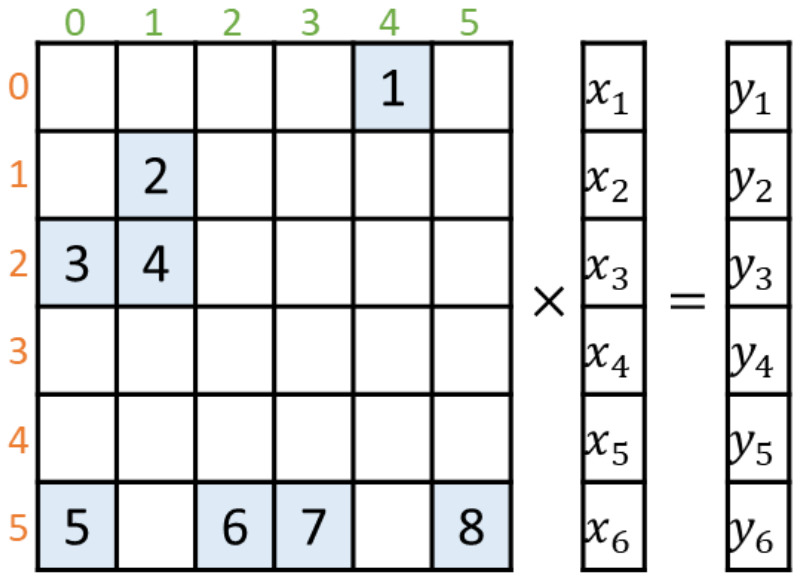
Outline

- Background
- Related works & Challenges
- Data preprocessing algorithm
- Hardware
- Experiment

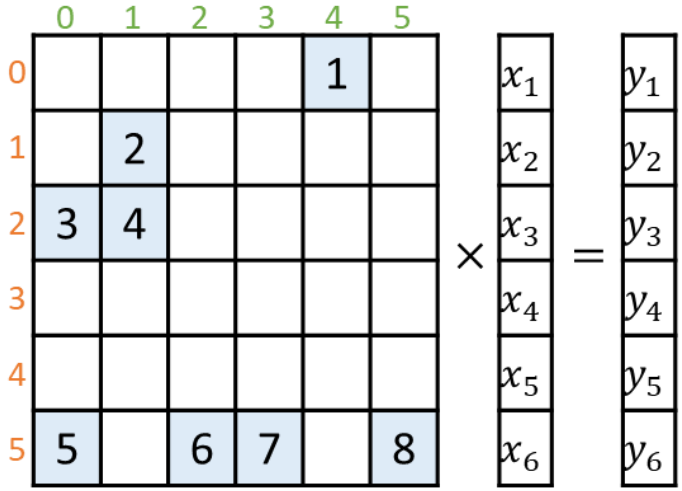
Sparse matrix-vector multiplication

SpMV multiplies every row in a sparse matrix by a corresponding element in the input vector to obtain every element in the output vector.

For example:



The compressed formats of the sparse matrix



COO

val	1	2	3	4	5	6	7	8
row	0	1	2	2	5	5	5	5
col	4	1	0	1	0	2	3	5

CSC

val	3	5	2	4	6	7	1	8
row	2	5	1	2	5	5	0	5
ptr	0	2	4	5	6	7	8	

CSR

val	1	2	3	4	5	6	7	8
col	4	1	0	1	0	2	3	5
ptr	0	3	4	6	7	9	10	

- **COOrdinate (COO)**

  values, column indexes, and row indexes

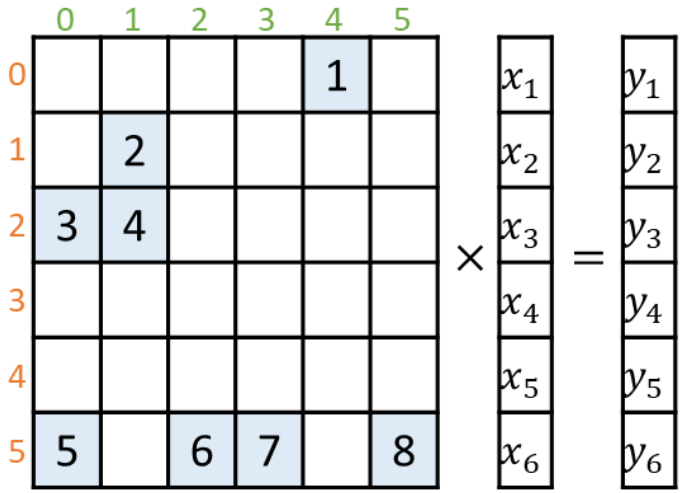
- **Compressed Sparse Column (CSC)**

  replaces column indexes with the ptr

- **Compressed Sparse Row (CSR)**

  replaces row indexes with the ptr

The compressed formats of the sparse matrix



COO

val	1	2	3	4	5	6	7	8
row	0	1	2	2	5	5	5	5
col	4	1	0	1	0	2	3	5

CSC

val	3	5	2	4	6	7	1	8
row	2	5	1	2	5	5	0	5
ptr	0	2	4	5	6	7	8	

CSR

val	1	2	3	4	5	6	7	8
col	4	1	0	1	0	2	3	5
ptr	0	3	4	6	7	9	10	

	Data size	Control logic
COO	large	simple
CSR & CSC	small	complex

Sparse matrix-vector multiplication(SpMV)

The SpMV can be expressed as Equation:

$$y_i = \sum_{j=0}^{cols} A_{i,j} * x_j \, (A_{i,j} \neq 0, 0 \leq i \leq rows)$$

For one non-zero elements $A_{i,j}$:

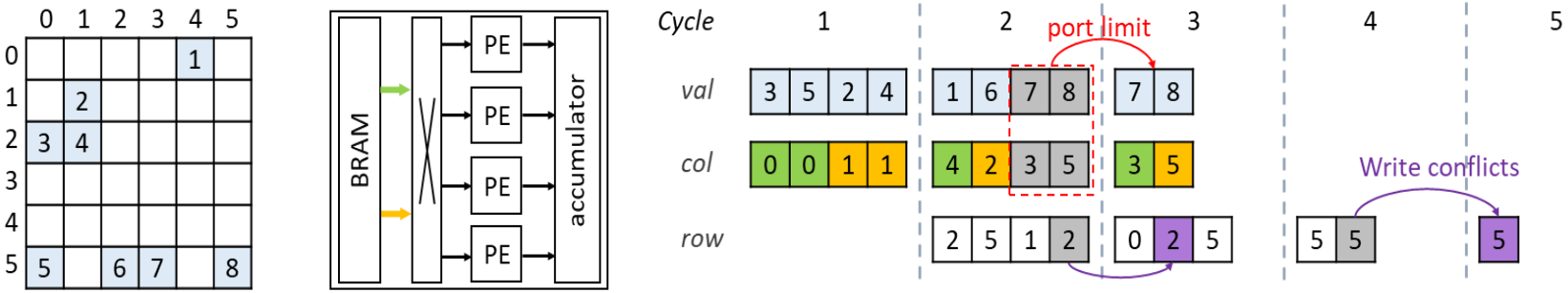
- Step1: access its input vector (x) irregular access
- Step2: multiply with x
- Step3: access its partial sum (y) irregular access
- Step4: add with y
- Step5: update y irregular access

3 irregular memory accesses

2 floating-point calculations

The challenge of the SpMV

An example of parallel SpMV. In the example, 4 non-zero elements are computed in a cycle:



Challenges of parallel SpMV:

- read port conflicts
- write conflicts

Irregular access

The related work of the SpMV

There are two different works to reduce irregular access conflicts:

- Replacing the column indices (**32 bits**) of the COO format with the corresponding input vector elements (**64 bits**)[1].

  eliminate port conflict & more data redundancy

- Using element-wise data reordering is carefully performed to form a data group referring to only two vector elements in a cycle[2].

  can not reduce the port limit in some matrices.

---

[1] Optimized Data Reuse via Reordering for Sparse Matrix-Vector Multiplication on FPGAs.

[2] ReDESK: A Reconfigurable Dataflow Engine for Sparse Kernels on Heterogeneous Platforms.

The challenge of the SpMV

Challenges:

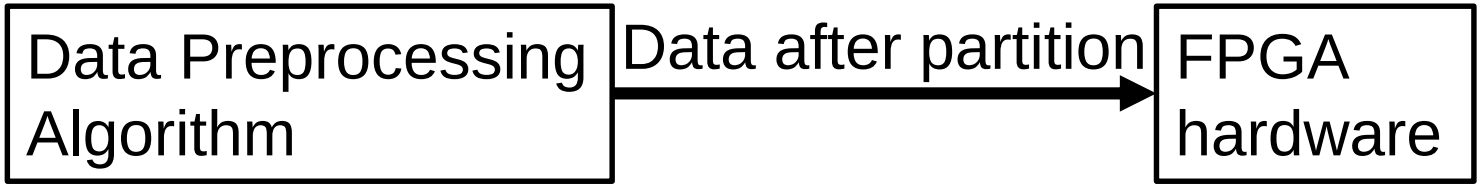
- port limit
- write conflict
- data redundancy

The measures in our work:

- Reading-Conflict-Free vector buffer

  Duplicating vector elements via partition

- Writing-Conflict-Free adder tree
- COO format



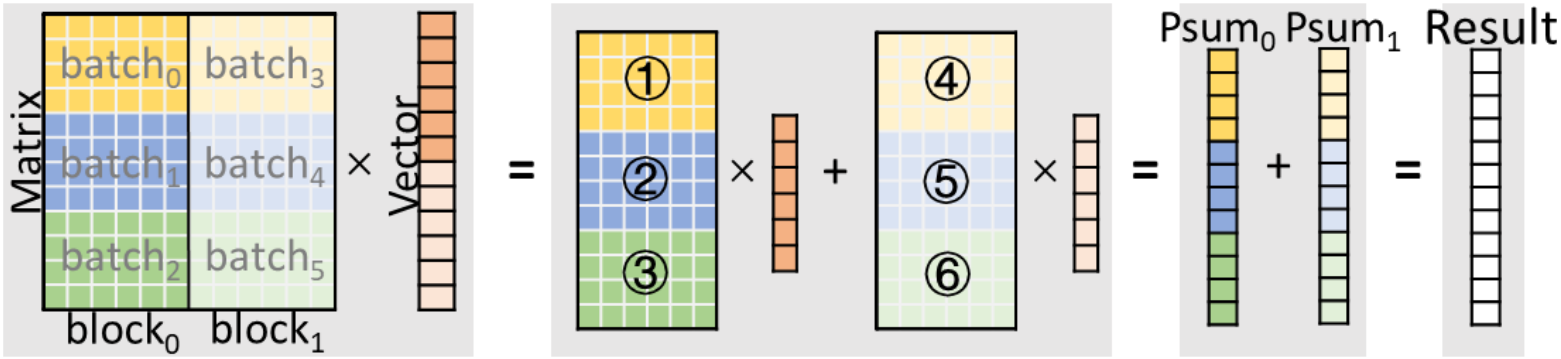
The reason for partitioning

The number of elements in the input vector = the number of columns in the matrix

The SpMV on big size matrix: the storage space for input vector **>** the size of BRAM

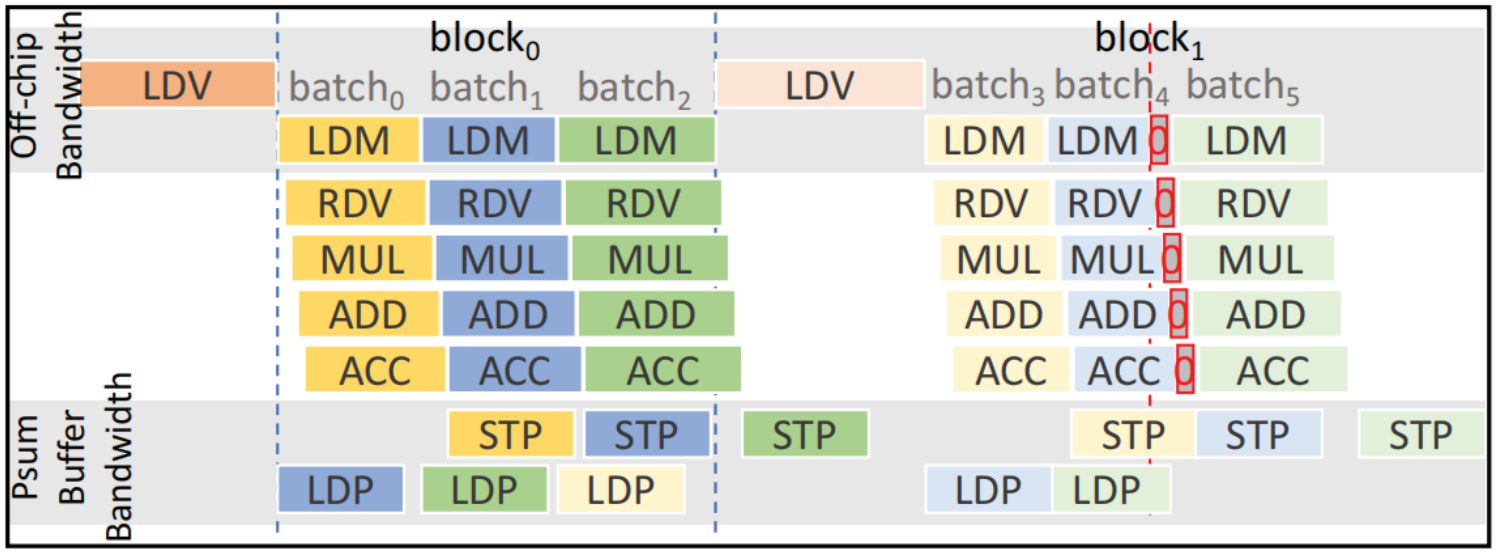
Partition the input vector and the matrix

How to do partitioning



- Matrix is partitioned into two block
- Vector is partitioned into two segments
- Each block is partitioned into three batches
- Partial sums of each block are added up to get the output result

How to execution after partitioning



LDV-load vector LDM-load matrix RDV-read vector

MUL-multiplication ADD-addition ACC-accumulation

STP-store partial sum LDP-load partial sum

- batch-by-batch within a block
- block-by-block among blocks

Formulation

To eliminate the impact of bandwidth in different designs, related works use normalized bandwidth utilization as the metric, which is the ratio of performance (GFLOPs) to off-chip memory bandwidth (GB/s) in the unit GFLOP/GB:

$$BU = \frac{2 * N_{nonzero}}{Datawidth_{byte} * C_{total}}$$

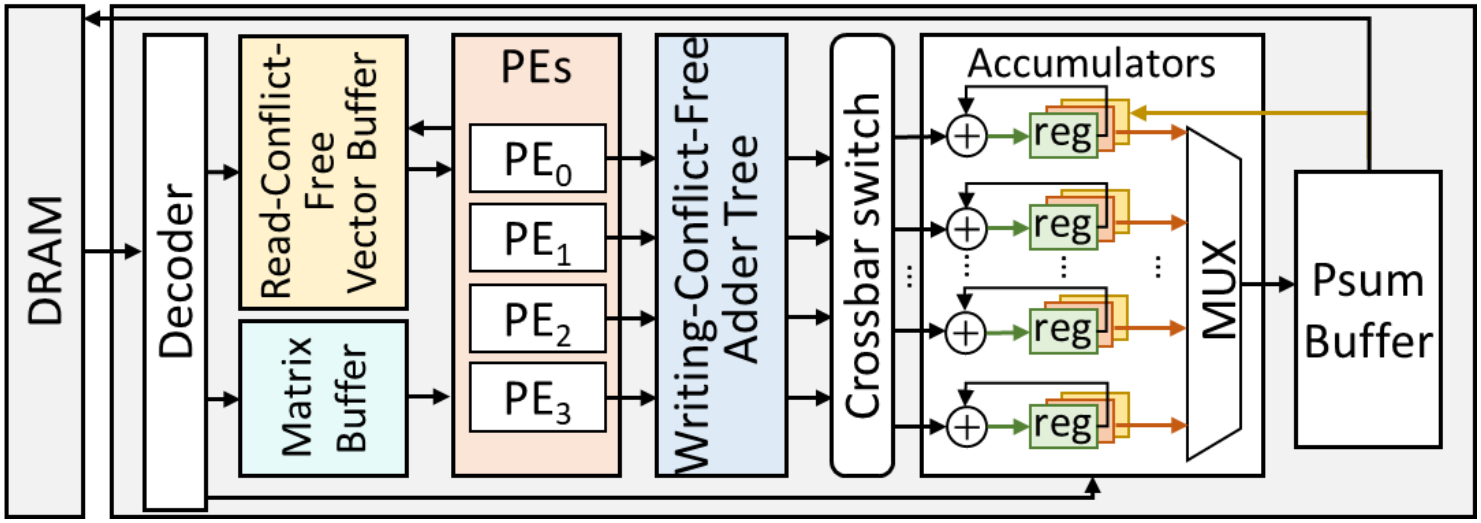
- $N_{nonzero}$: the number of non-zero elements
- C_{total} : the total execution time in cycles
- $Datawidth_{byte}$: the data width in bytes of the off-chip memory interface

Formulation

$$C_{total} = C_{ldv} + C_{exe} + C_{pipe}$$

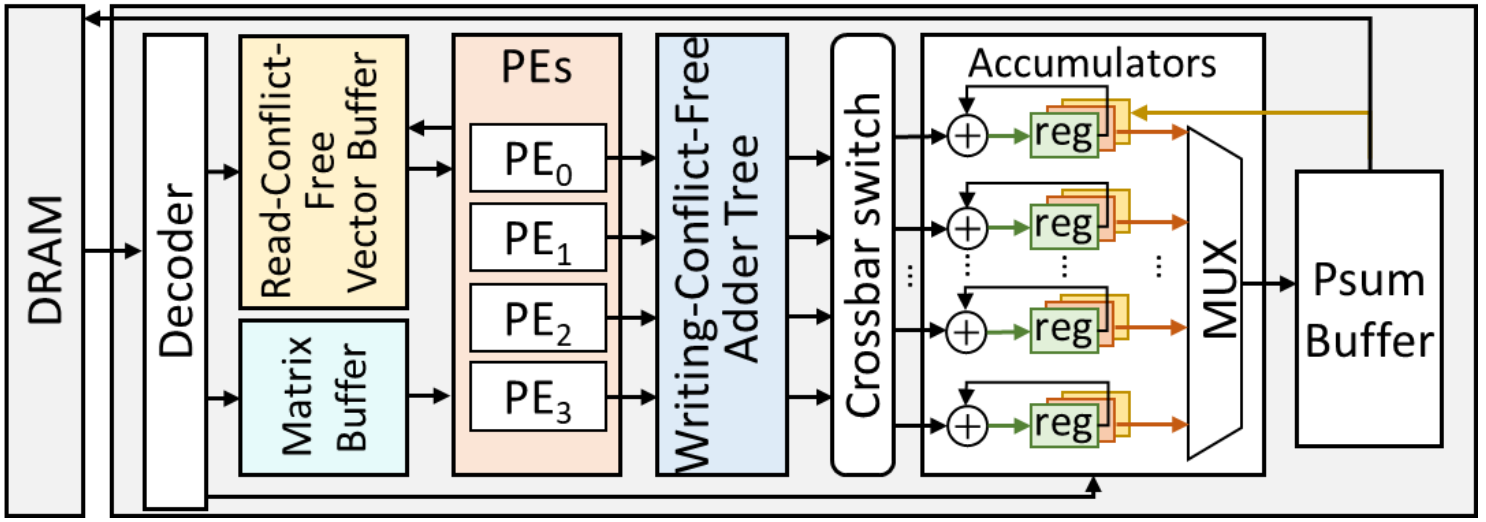
- C_{ldv} : the cycles for processing the data in all batches
- C_{exe}: the total execution time in cycles
- C_{pipe}: the cycles for starting the pipeline

Overall Architecture



- Decoder loads data.
- Decoder sends data to the read-conflict-free vector buffer, accumulator, or matrix buffer.
- PEs read matrix elements and vector elements.
- Pes perform multiplication operations.
- Send the multiplication results to the writing-conflict-free adder tree.

Overall Architecture



- Adder tree performs addition operations.
- Accumulators add all partial sums.
- Write results to the on-chip partial sum.

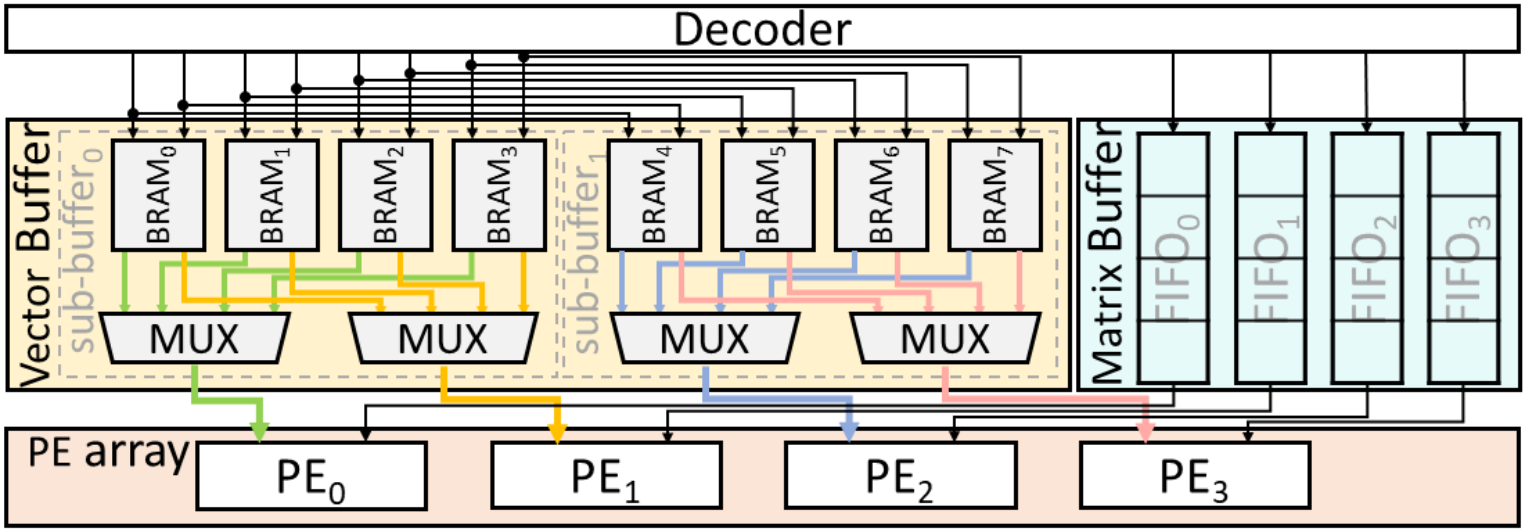
The reason for increasing the port of the vector buffer

For a non-zero element, COO includes a 64-bit double float value, a 32-bit row index, and a 32-bit column index. The 512-bit off-chip bandwidth can load 4 matrix elements from DRAM to the decoder per cycle.

4 irregular accesses

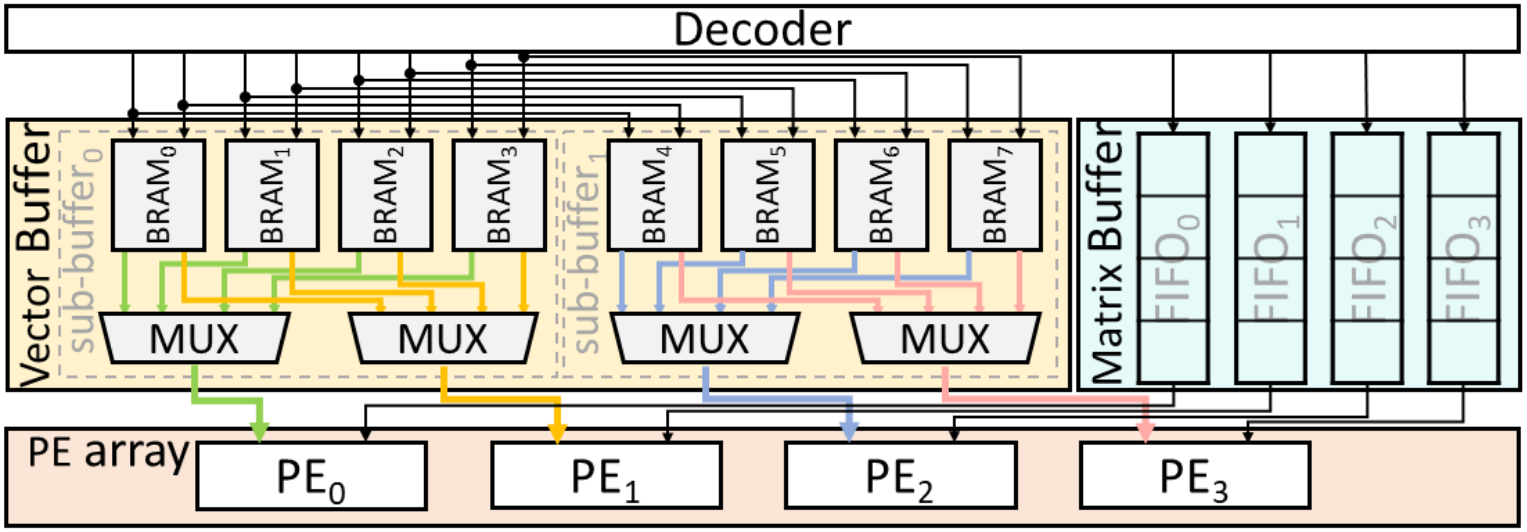
4 independent read ports

The architecture of the vector buffer



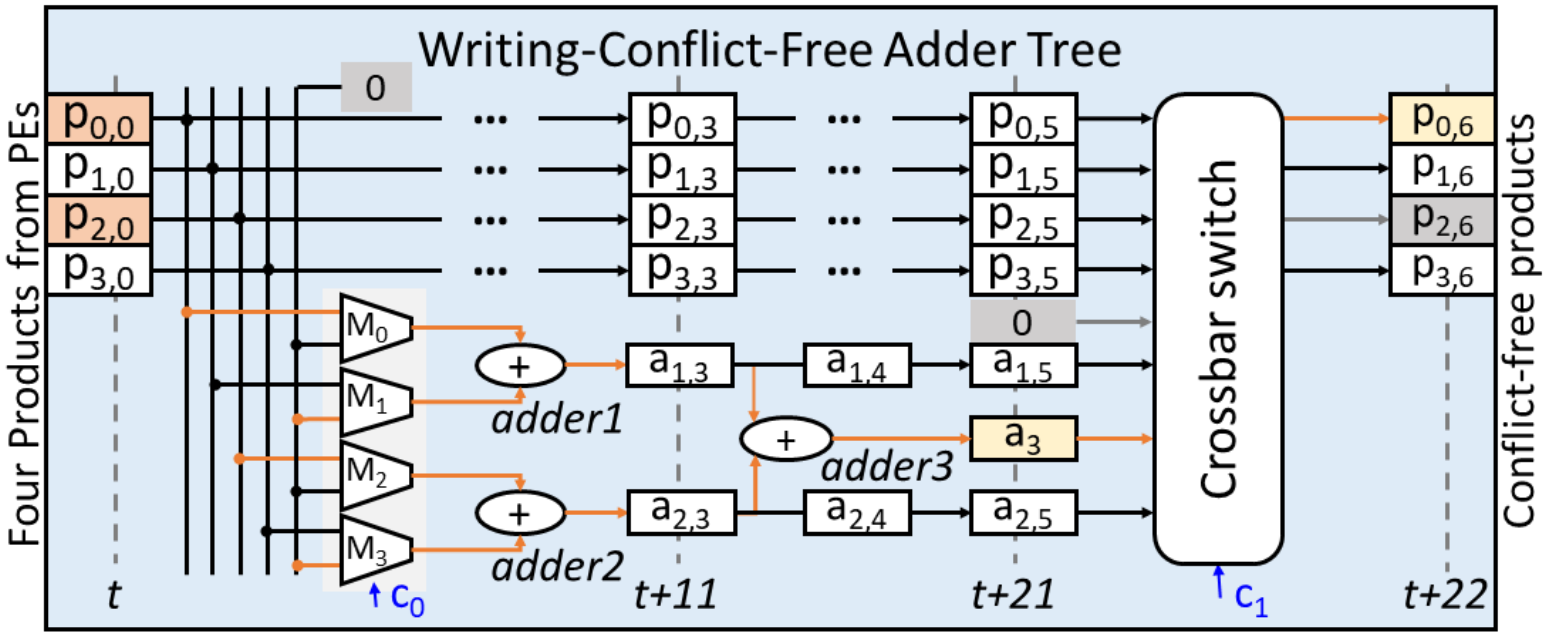
- Each sub-buffer include 4 BRAMs.
- In each sub-buffer, two 4-to-1 multiplexers (MUXs) are added to select vector elements from different BRAMs to 2 PEs.

Time-division multiplexing of the vector buffer



- For loading vector elements, the ports work as writing ports.
- For performing multiplication operations, the ports work as reading ports.

The architecture of adder tree



- Taking the multiplication result as inputs.
- Two adders are used to add the inputs and store the addition results to register.
- Another adder is used to add the previous addition results up and store the sum to register.
- Crossbar switch to select four partial sums as outputs.

Experiment setup

Benchmark

- Xilinx Zynq UltraScale ZCU106 (XCZU7EV-2FFVC1156 FPGA)
- University of Florida sparse matrix

Matrix	Rows/Cols	Nonzero	Density
raefsky1	3242	293409	2.79156%
memplus	17758	99147	0.03144%
stanford	281903	2312497	0.00291%
s3dkt3m2	90449	3686223	0.04506%
psmigr_2	3140	540022	5.47712%
rma10	46835	2329092	0.10618%
t2d_q9	9801	87025	0.09060%
epb1	14734	95053	0.04379%
lns_3937	3937	25407	0.04379%
mac_econ	206500	1273389	0.00299%
dw8192	8192	41746	0.06180%
pwtk	217918	11524432	0.02427%

Experiment setup

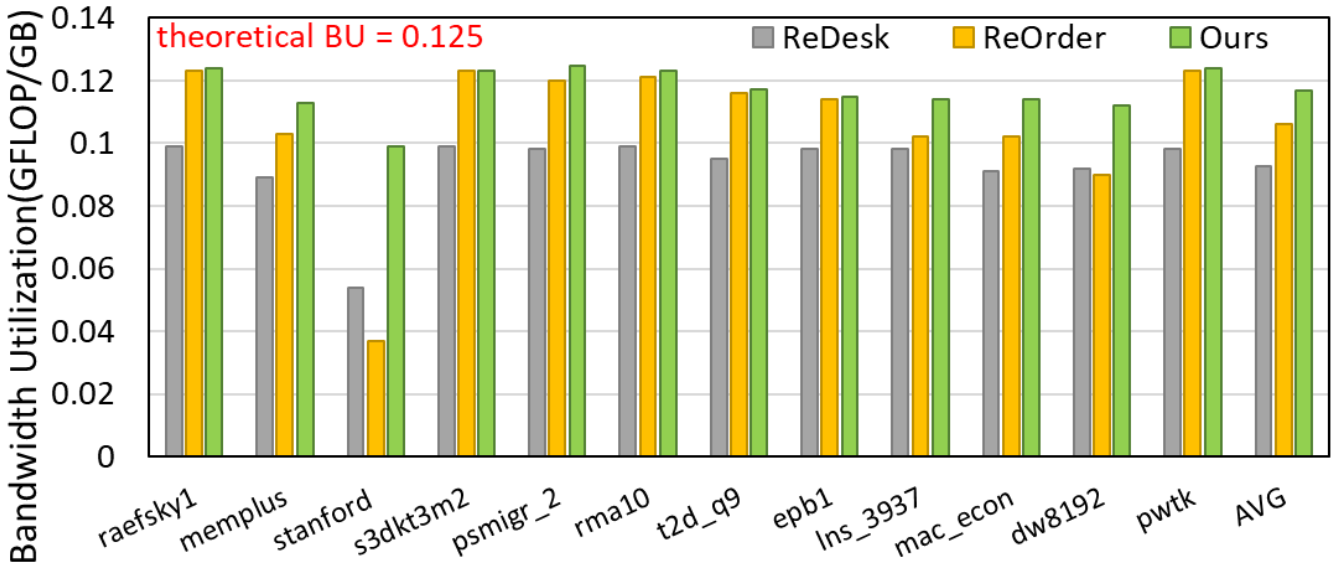
Parameter Setup

- 100 MHz
- block width is set to 2^{14}
- batch height is set to 64

Baseline

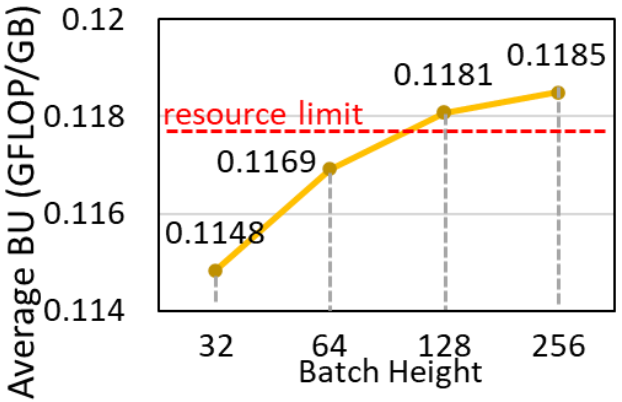
- ReDESK
- ReOrder

Bandwidth utilization



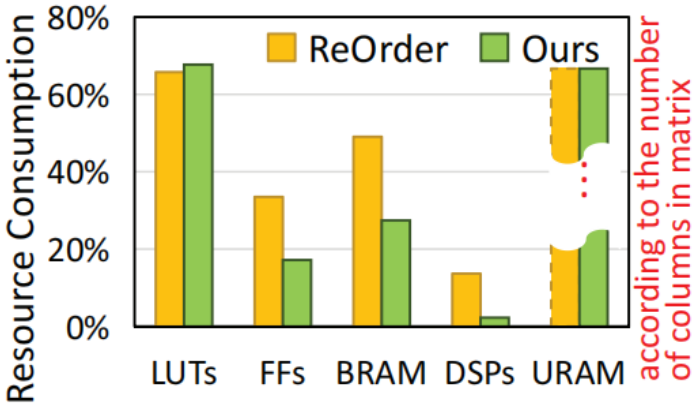
- Theoretical peak BU is 0.125.
- Our work Achieve 1.26x and 1.10x improvement on BU compared to ReDESK and ReOrder.

Batch height



- The average BU increases with the increase in batch height.
- A higher batch height means that the accumulators require more LUT resources.

Resource



- ReOrder has the same hardware as ours
- More LUTs
- Fewer FFs¥BRAMs¥DSPs
- Same URAM

Conclusion

- partition matrix and vectors
- read-conflict-free vector buffer
- writing-conflict-free adder tree

THANKS