Microsoft Research

An FPGA Drop-In Replacement for Universal Matrix-Vector Multiplication*

Eric S. Chung[1], John D. Davis[1], Srinidhi Kestur[2]

[1]Microsoft Research, Silicon Valley Lab
[2]The Pennsylvania State University

*Presented at FCCM'12

The Intersection of FPGAs & CPUs

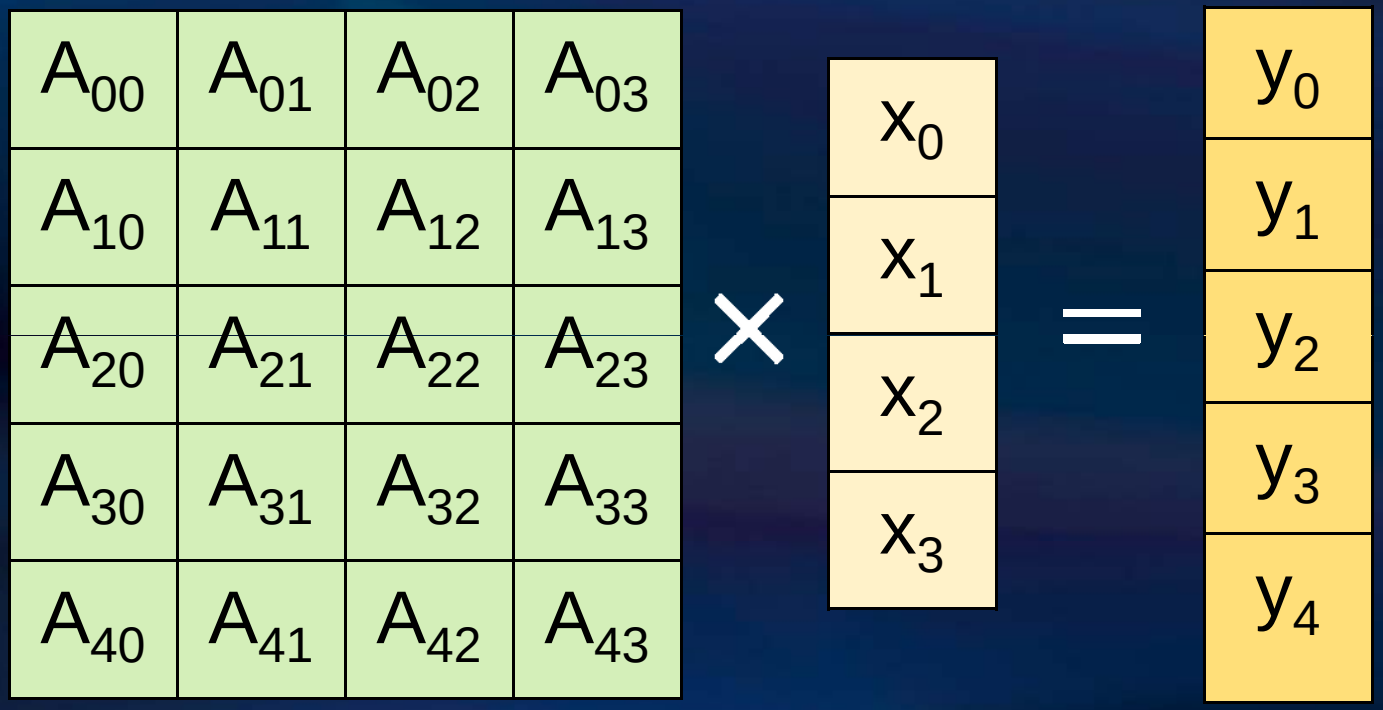
- Transistors are abundant, power is scarce
  - Utilize abundant silicon for FPGA fabrics
  - Energy efficient and post-silicon flexibility
- Challenges
  - FPGAs incur large compile-time and reconfiguration overheads
  - Must provide significant advantages over other architectures (many-core, GPGPU)
  - What are the right applications?

Matrix-Vector Multiply

Matrix-Vector Multiply on FPGA

Microsoft Research

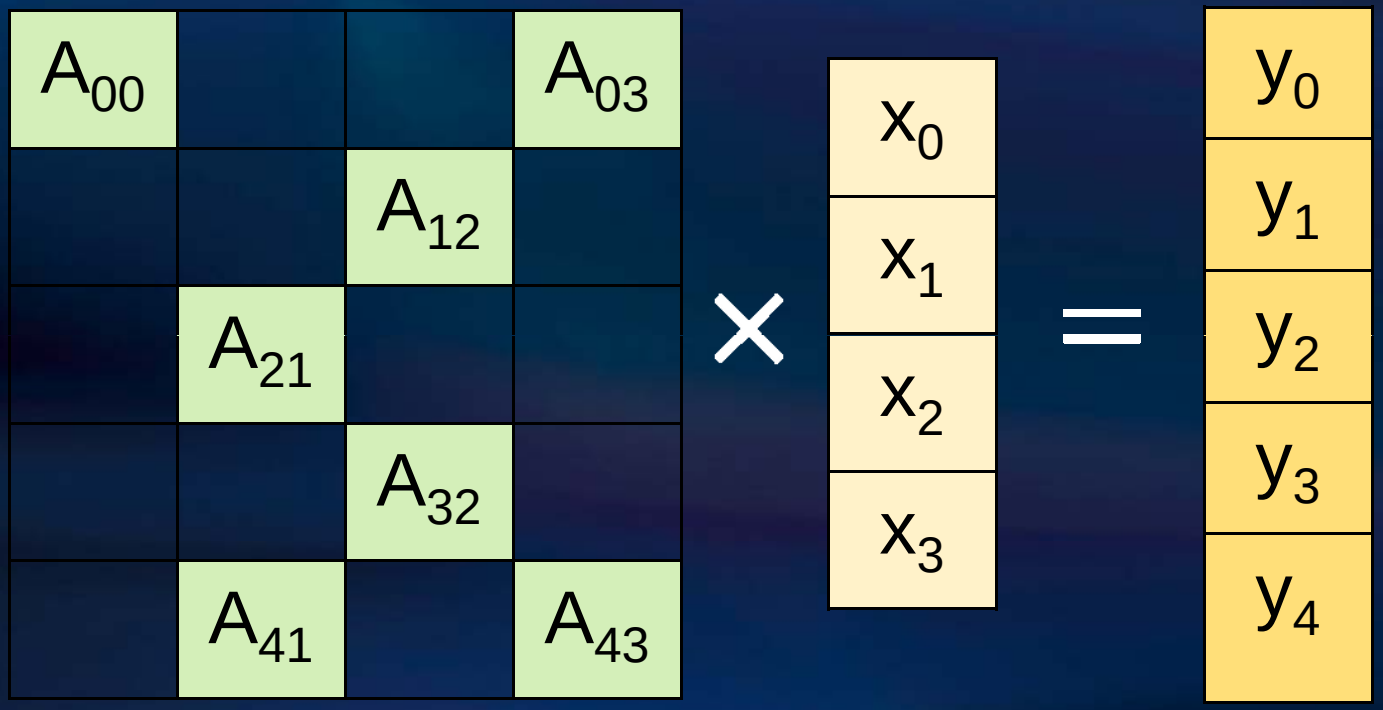
Matrix-Vector Multiply (Dense)



$$\vec{y} = A\vec{x}$$

Microsoft Research

Matrix-Vector Multiply (Sparse)



$$\vec{y} = A\vec{x}$$

Microsoft Research

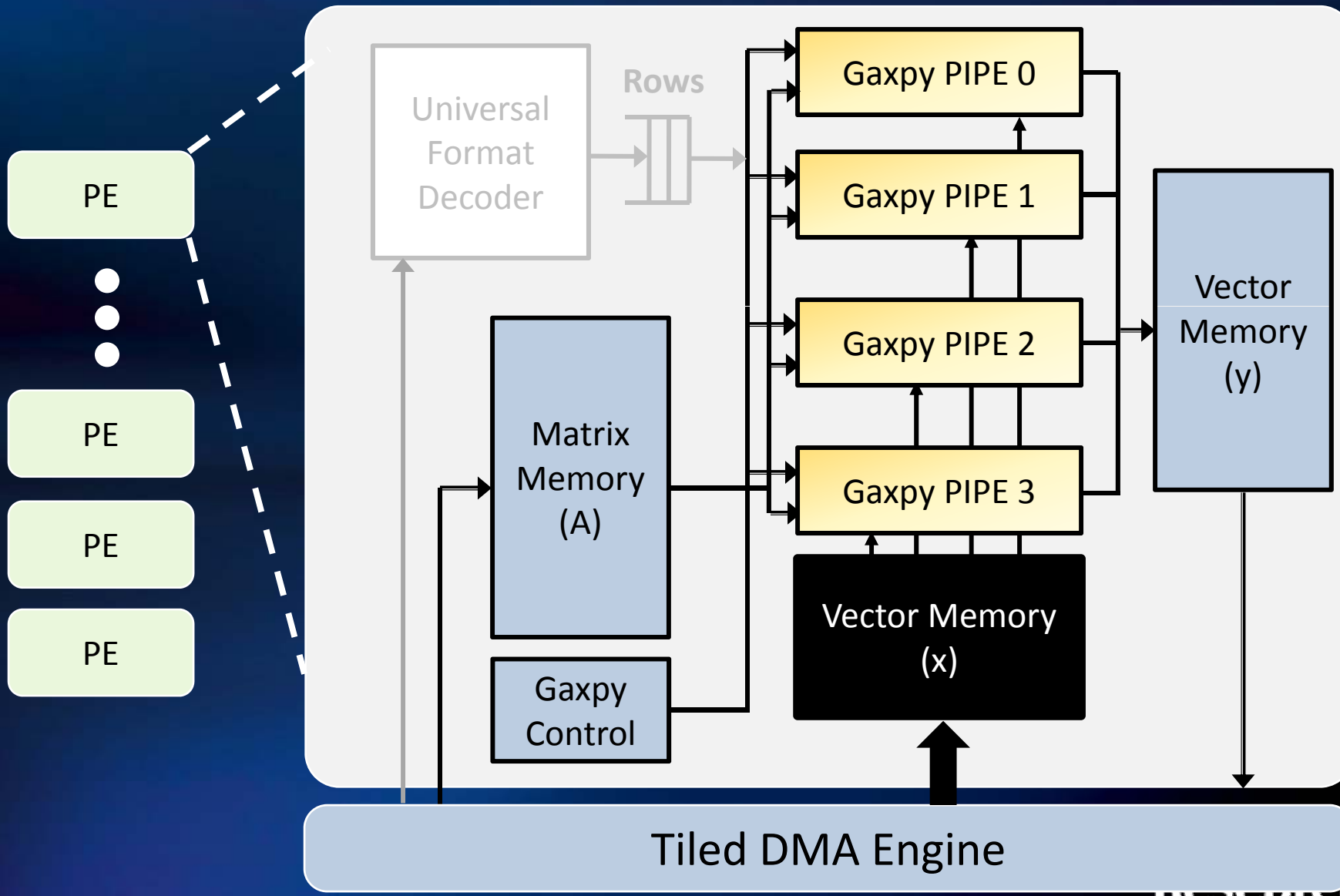
Accelerating $\vec{y} = A\vec{x}$ on FPGA

- Matrix-Vector-Multiply is Critical HPC Kernel
  - 10s of papers published/year on this topic
- Existing works on GPU/CPU/FPGA
  - Performance sensitive to matrix sparsity and formats
  - Processor-centric data formats
  - High power consumption (GPU/CPU)
- FPGA opportunities
  - Exploit custom variable-length formats
  - Low power, large memory configurations
  - Efficient, robust resource utilization

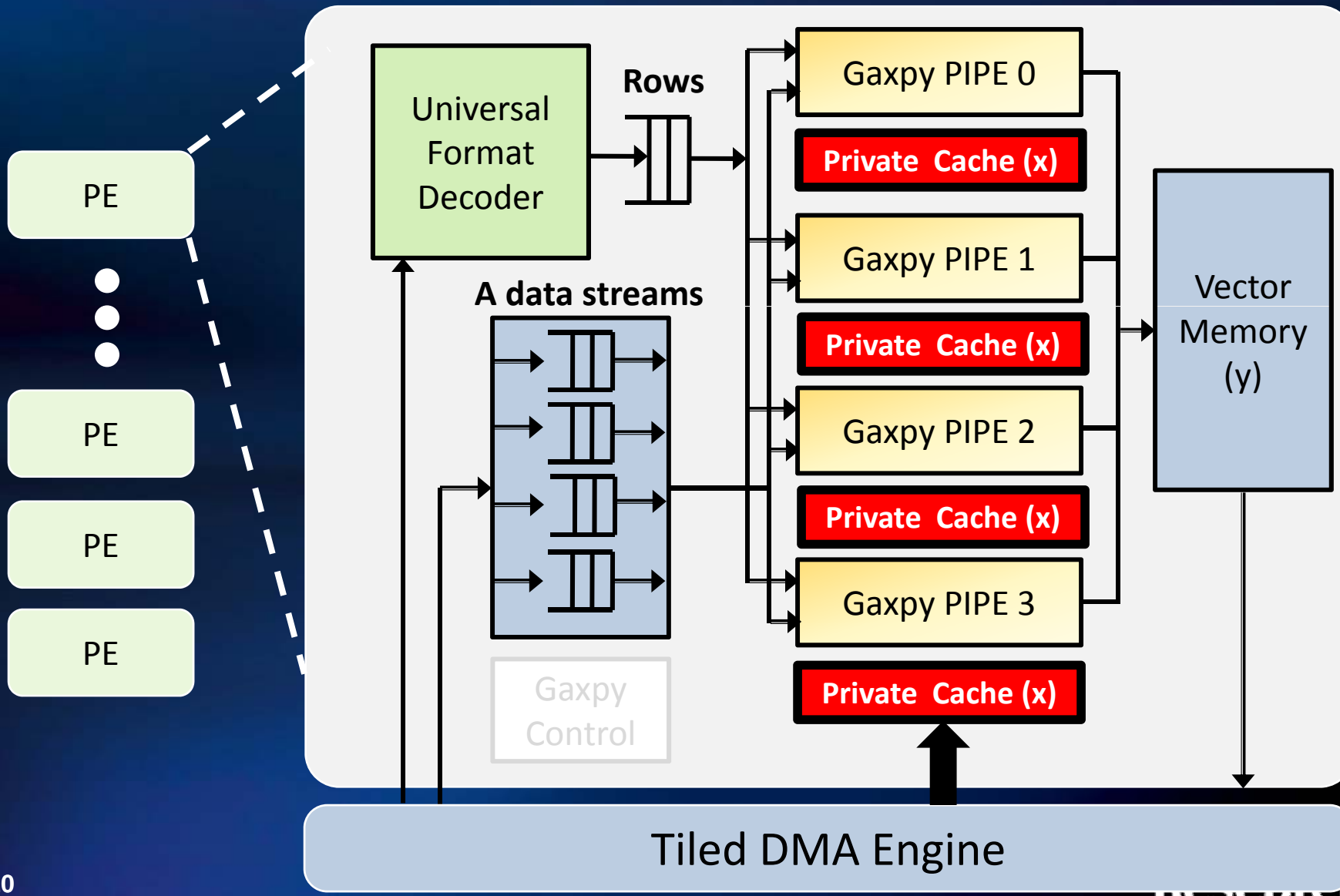
Objective: One Bitstream To Rule Them All

- Build single FPGA bitfile library for $\vec{y} = A\vec{x}$
- Handle large-scale inputs ($\geq$GB)
- Avoid costly run-time reconfiguration
- Exploit bit-level manipulation
- Dense and sparse inputs
- Process multiple sparse matrix formats
  - COO, CSR, Dense, DIA, ELL, etc.

Universal MVM (Dense Mode)



Universal MVM (Sparse Mode)



Sparse Matrix Formats

1	0	4	0
3	7	0	0
0	0	2	9
5	8	0	7

Data Array

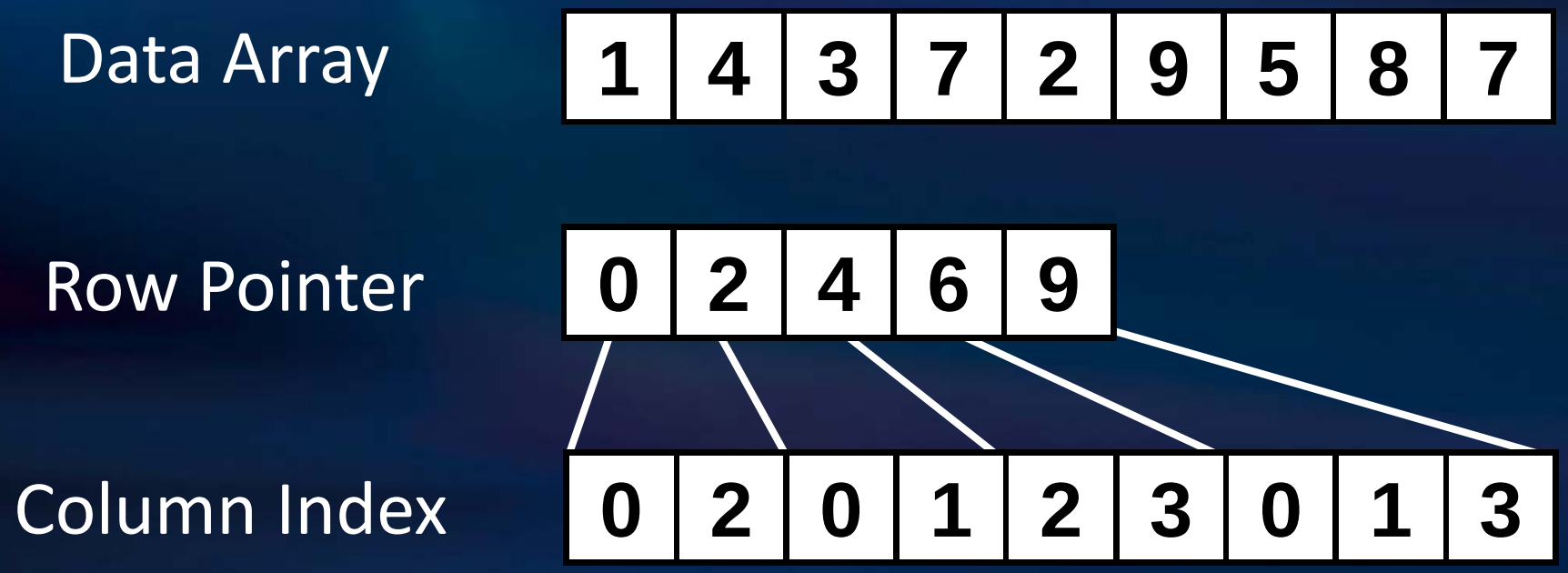
1	**4**	**3**	**7**	**2**	**9**	**5**	**8**	**7**

Coordinate (COO) Format

Data Array	1	4	3	7	2	9	5	8	7
Row Index	0	0	1	1	2	2	3	3	3
Column Index	0	2	0	1	2	3	0	1	3

COO Overhead = $\boldsymbol{Nonzeros} \times (\boldsymbol{4B} + \boldsymbol{4B})$

Compressed Sparse Row (CSR) Format



CSR Overhead =

$$Nonzeros \times 4B + Rows \times 4B$$

Microsoft Research

ELLPACK (ELL) Format

Data w/ Padding

1	4	*
3	7	*
2	9	*
5	8	7

Column Metadata

0	2	*
0	1	*
2	3	*
0	1	3

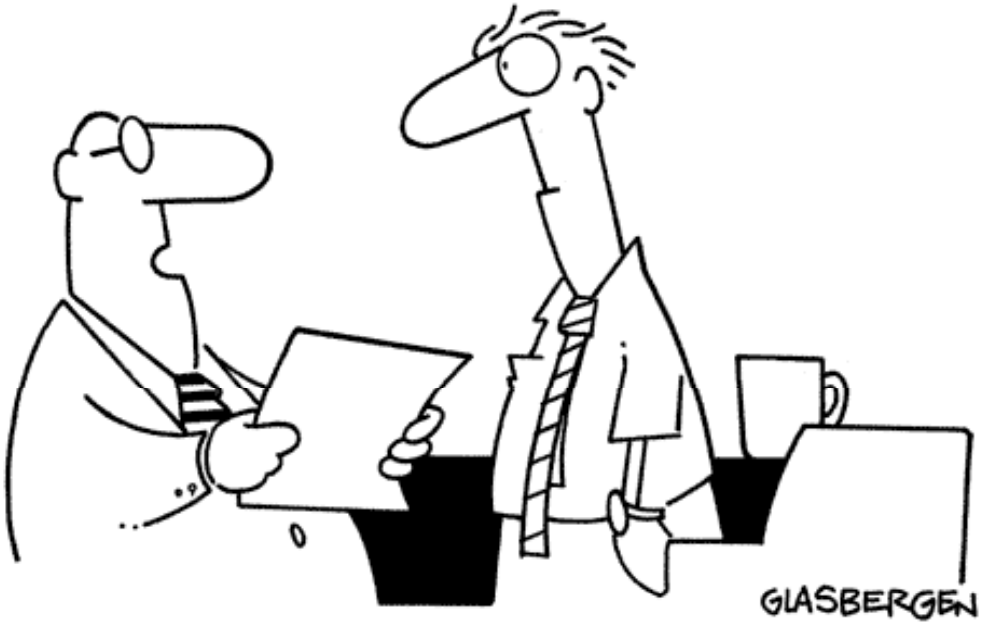
⟵ k=3 ⟶

ELL Overhead =

$$4B \times Rows \times k + DataPad \times 8B$$

Microsoft Research

What's wrong with these formats?

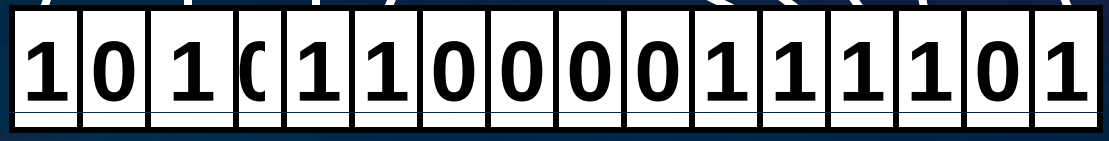


"I want you to find a bold and innovative way to do everything exactly the same way it's been done for 25 years."

FPGA Bit-Vector Format

Data Array

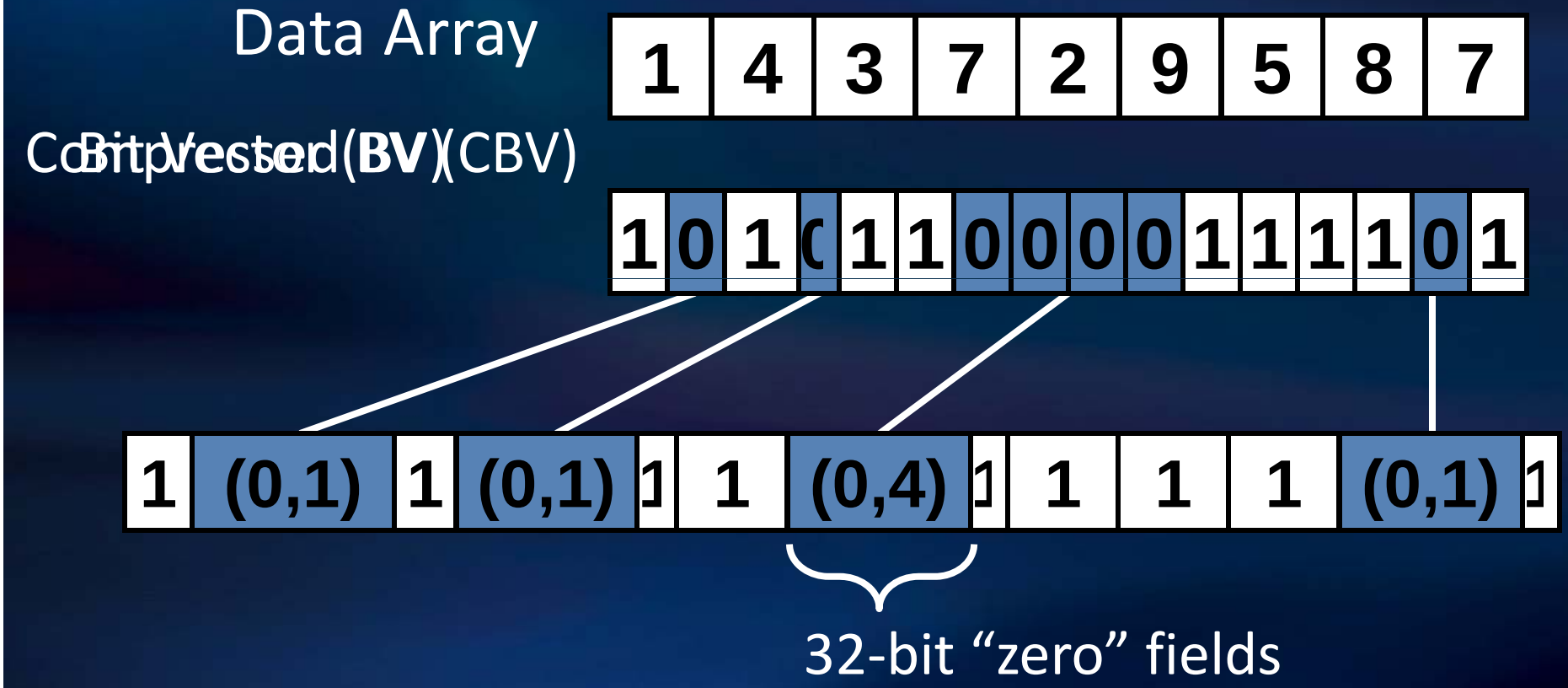
Bit Vector (BV)



BV Overhead = $\boldsymbol{Rows} \times \boldsymbol{Cols}\ \boldsymbol{x}\ \boldsymbol{1bit}$

Microsoft Research

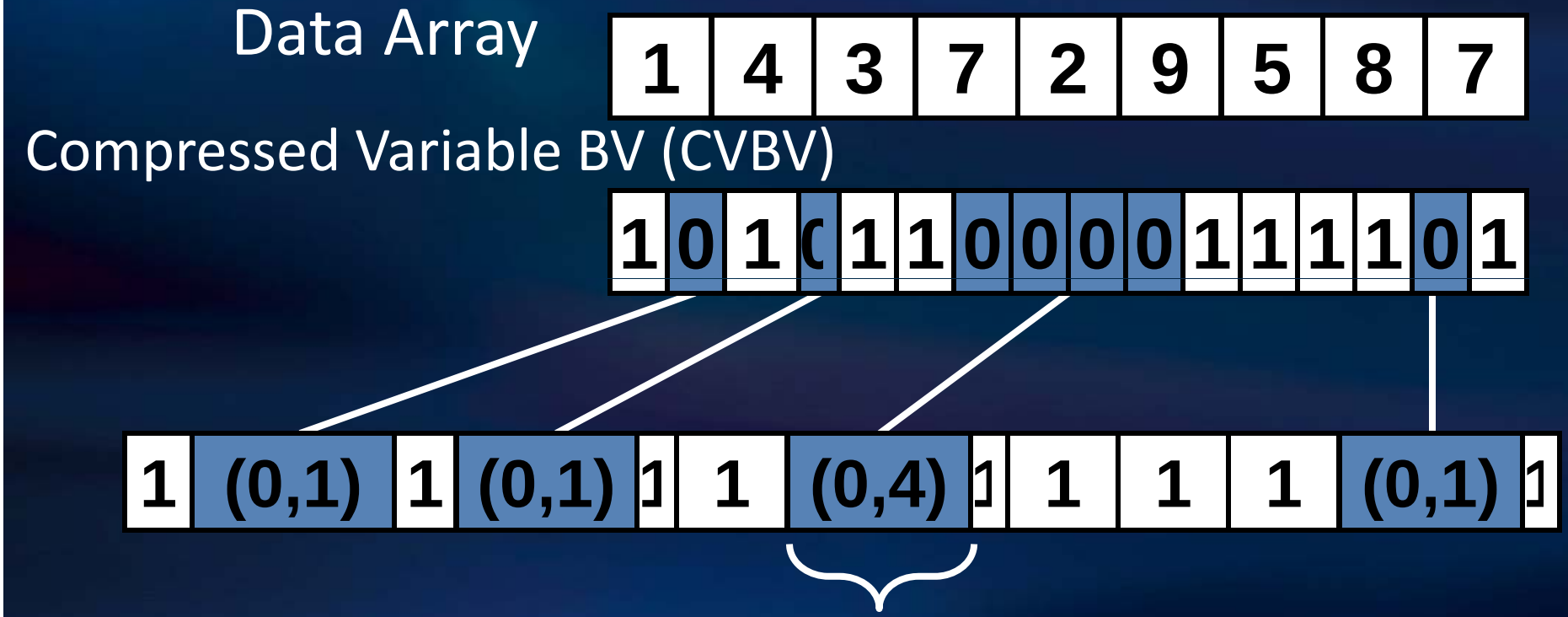
Compressed Bit Vector (CBV)



CBV Overhead = $Nonzeros \times 1bit + ZeroClusters \times 32bit$

Research

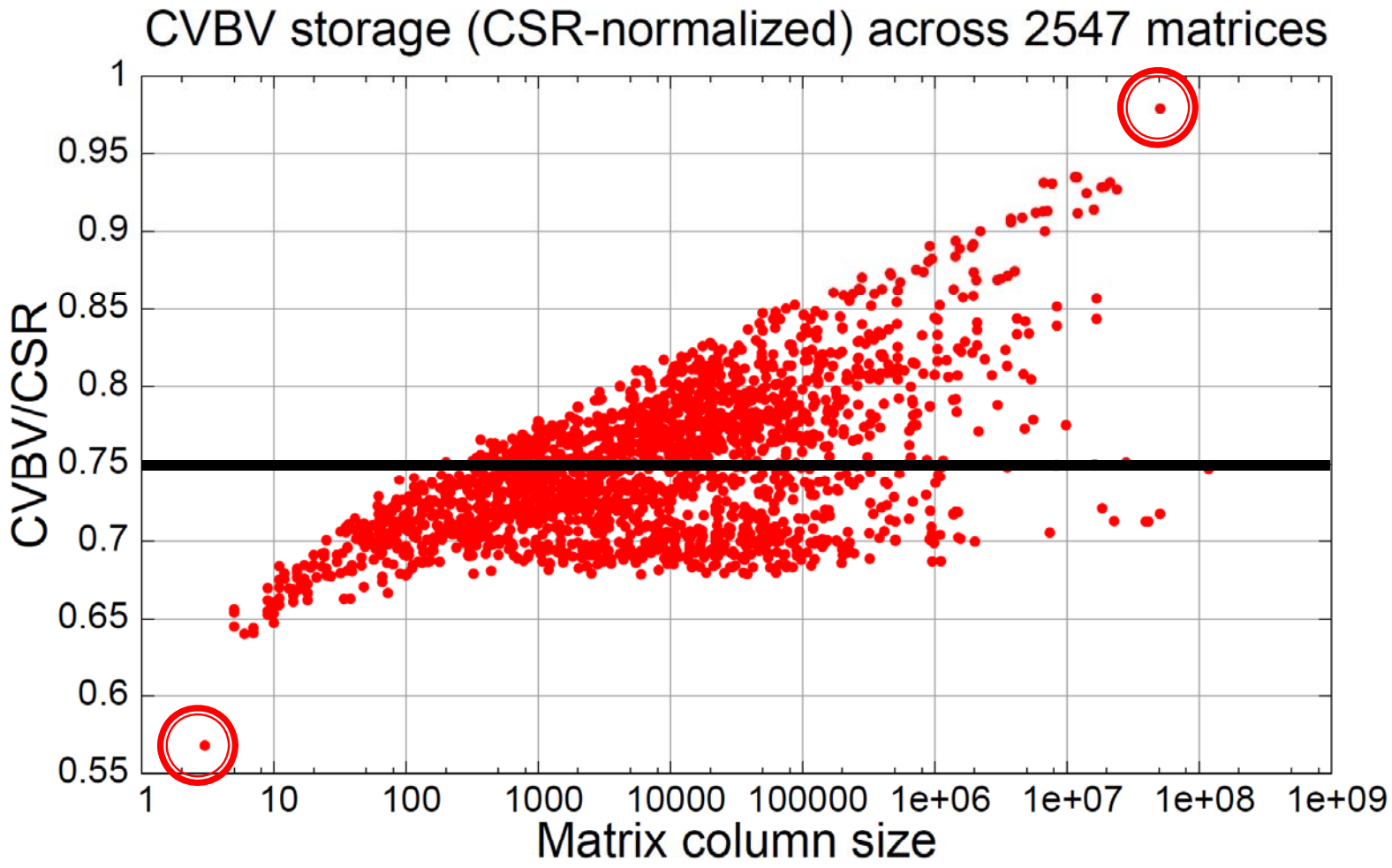
Compressed Variable BV (CVBV)



CVBV Overhead ~ input-dependent

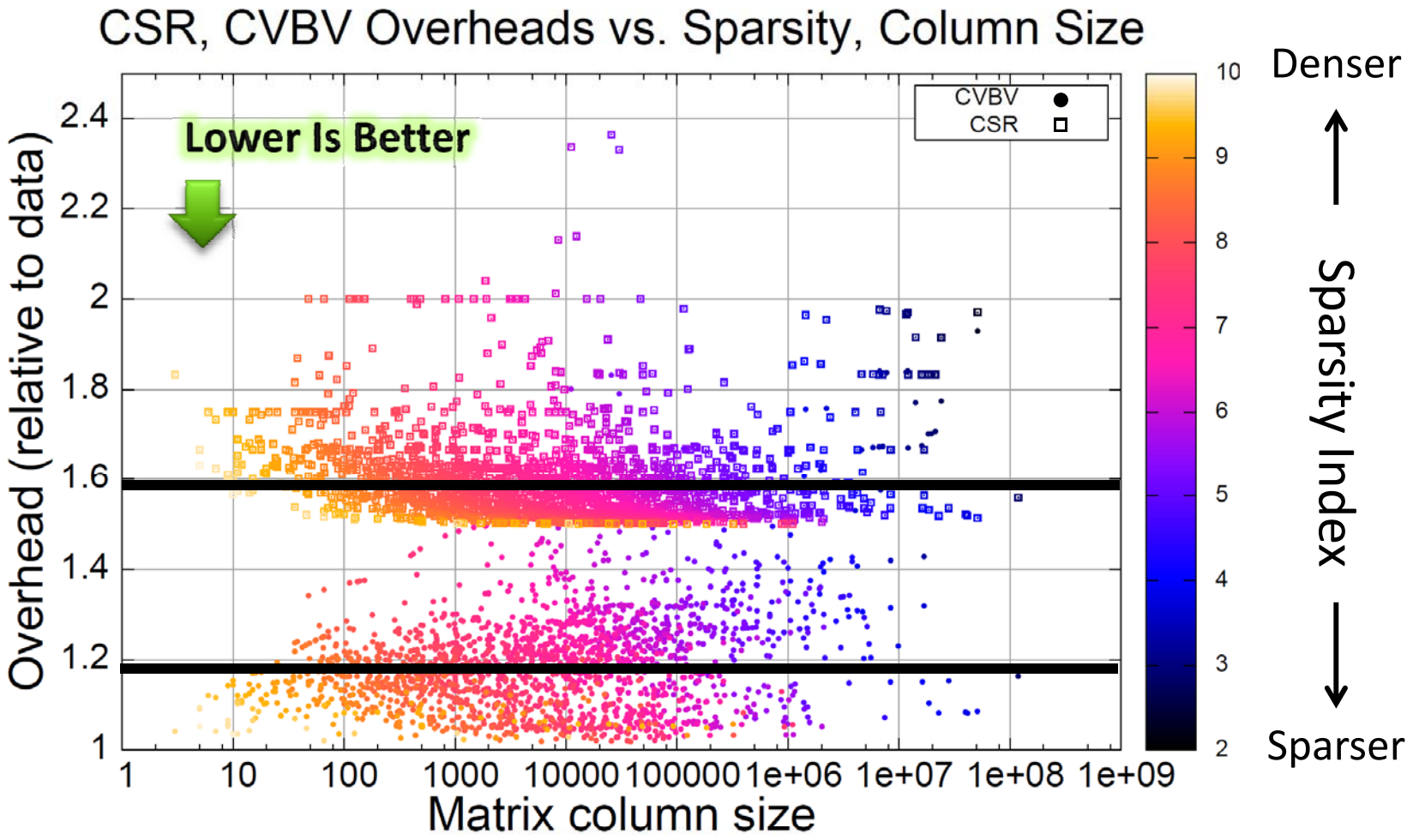
Microsoft Research

Sparse Format Comparison



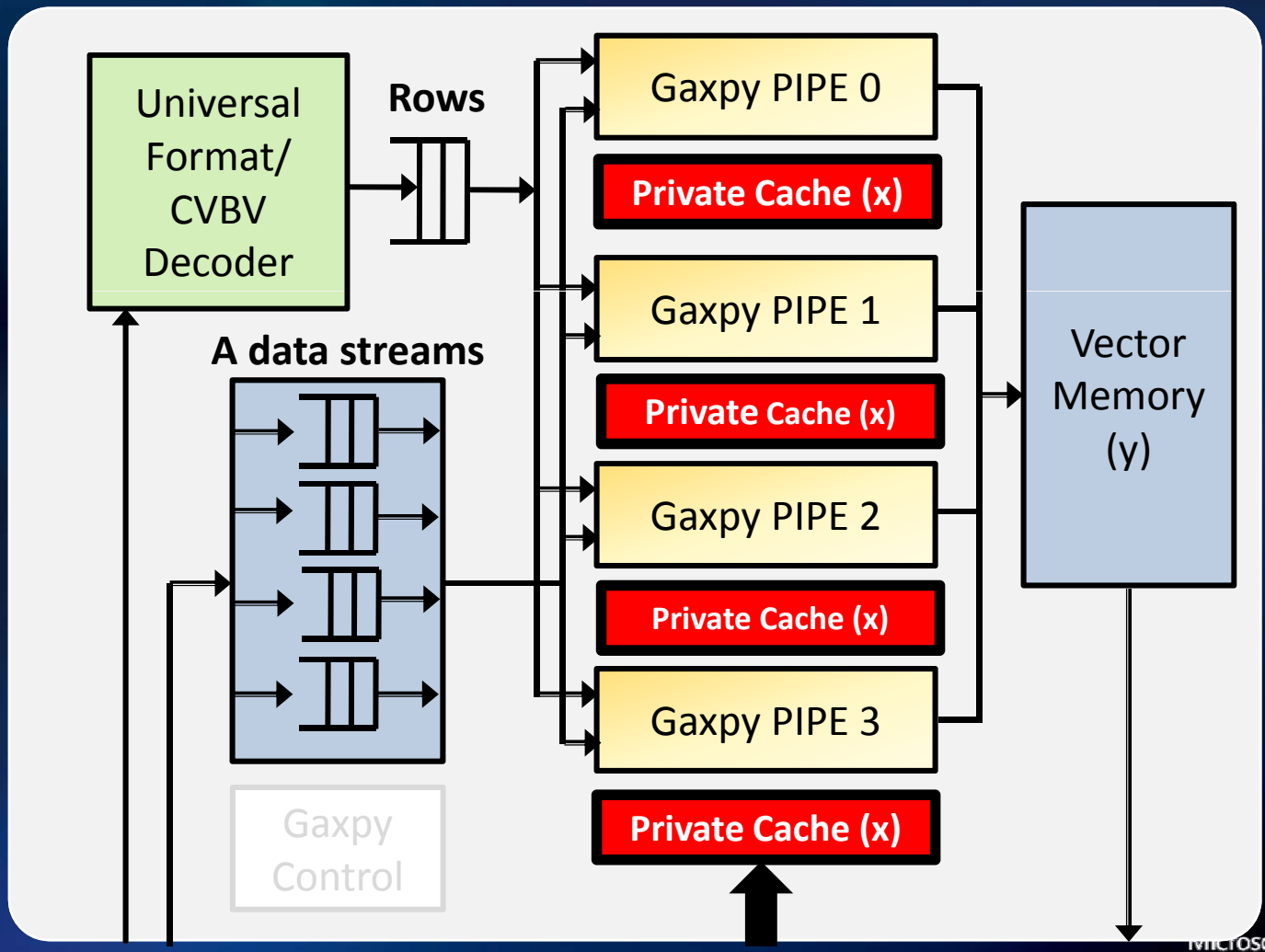
Microsoft Research

Sparse Matrix Format Overhead



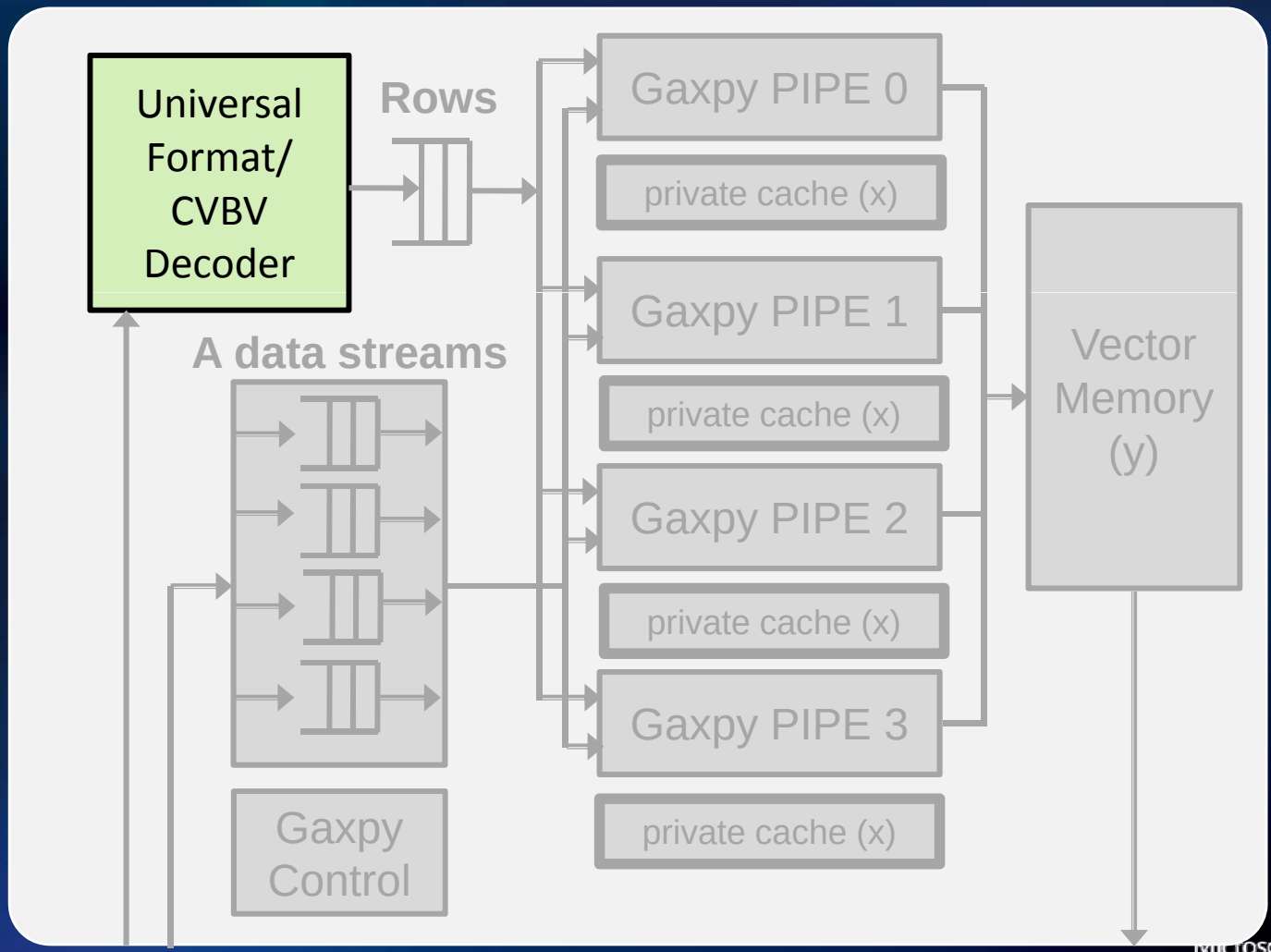
Microsoft Research

Universal Matrix Format Decoder



Microsoft Research

Universal Matrix Format Decoder



Microsoft Research

Universal Matrix Format Decoder

- Specify matrix format descriptors
  - Fixed/variable length, padding, index/ptr, etc.
- Translates row/column into sequence #
- Generate *BV (reduce storage/BW)
- Generate modified COO (consumed by PEs)
  - Row index, nonzero count per row, col indices

Universal Matrix Format Decoder

Algorithm 1 Universal Matrix Format Decoder.

Input: queue $streams[3]$
Output: Compressed data, rows, columns

```
 1: data = streams[0].head
 2: cx = streams[1].head
 3: rx = streams[2].head

 4: rowStream = FixedLenRows? stream[0] : stream[2]
 5: for i = 0 → NNZ − 1 do
 6:     r = RowAddress? rx : ((rowStream.idx-1)/K)
 7:     c = cx + ((pivot == -1) ? r : pivot)
 8:     stream[0].dequeue()
 9:     if stream[1] then
10:         stream[1].dequeue()
11:     end if
12:     if RowAddress then
13:         stream[2].dequeue()
14:     else if (rx − streams[1].idx) > 1 then
15:         stream[2].dequeue()
16:     end if
17: end for
```

Performance/Area Results

	PEs	LUT (% area)	RAM (% area)	DSP (% area)	GFLOPs (Peak)	GFLOPs (off-chip)	BW (% peak)
Dense V5-LX155T	16	72%	86%	88%	3.1	0.92	64.7
Dense V6-LX240T	32	71%	63%	56%	6.4	1.14	80
Dense+Sparse V5	16	74%	87%	91%	3.1	-	-

Sparse Inputs	V5-LX155T (Ours)	HC-1 (32 PE)[1]	Tesla S1070[2]
	GFLOPS / BWUsed	GFLOPS / BWUsed	GFLOPS / BWUsed
dw8192	0.10 / 10.3%	1.7 / 13.2%	0.5 / 3.1%
t2d_q9	0.15 / 14.4%	2.5 / 19.3%	0.9 / 5.7%
epb1	0.17 / 17.1%	2.6 / 20.2%	0.8 / 4.9%
raefsky1	0.20 / 18.5%	3.9 / 29.0%	2.6 / 15.3%
psmigr_2	0.20 / 18.6%	3.9 / 29.6%	2.8 / 16.7%
torso2	0.04 / 4.0%	1.2 / 9.1%	3.0 / 18.3%

[1] Nagar et al., A Sparse Matrix Personality for the Convey HC-1, FCCM'11
[2] Bell et al., Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors, SC'09

Conclusions

- We propose CVBV/CBV sparse format
  - 25% reduction in storage/bandwidth compared to well-known CSR
  - Exploits bit-level manipulation of FPGA
- Single bit file for dense AND sparse MVM
  - Universal matrix format decoder
  - DMA and caches for memory management
  - Stall-free accumulator
  - Scalable design, implemented on multiple platforms

Future Work / Discussion

- Performance and energy efficiency compared to CPUs and GPGPUs
- ASIC core integration
  - Sparse *BV load/store engine for future CPU/GPGPU
- Pushing bits over wires is dominant energy cost
  - 256bits across 10 mm;
    Today @ 310pJ/bit ➔ 2017 @ 200pJ/bit
    (vs. 7x reduction in pJ/FMADD) [Dally, IEEE MICRO'11]
- Will 'bit-level' memory systems make sense in future?
  - Bit-level encoding/compaction/efficiency
  - Sparse, bit-level load-store engine
  - Applications?

Microsoft Research

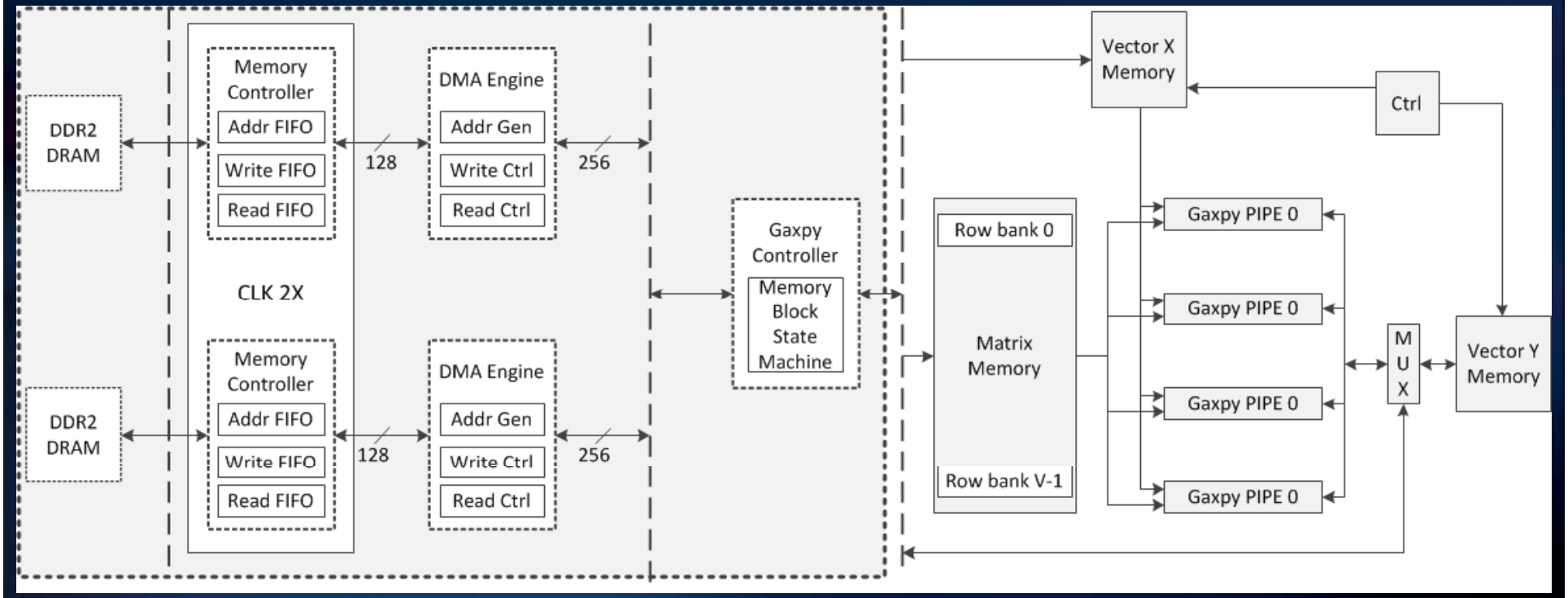
Questions?

Microsoft Research

Back-up

Universal MVM Engine

Dense



Microsoft Research

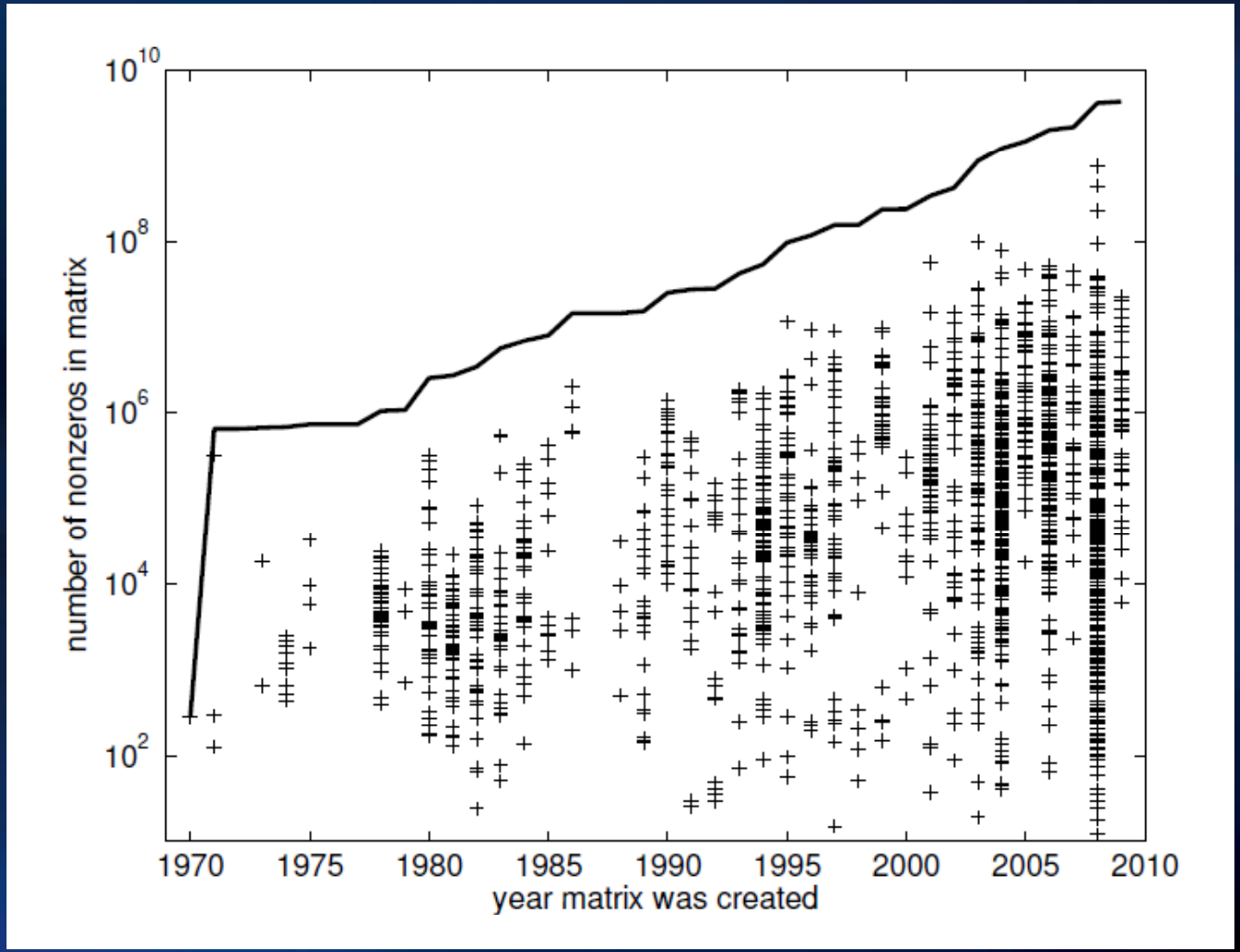
Format Summary

matrix	rows	cols	nonzeros	% nonzeros	CSR	COO	ELL	BV	CBV	CVBV
conf5_0-4x4-10	3072	3072	119808	1.27%	1.0	1.32	0.99	1.47	0.70	0.76
dw8192	8192	8192	41746	0.06%	1.0	1.25	1.47	16	0.69	0.83
psmigr_2	3140	3140	540022	5.48%	1.0	1.33	13.31	0.86	0.77	0.92
scircuit	170998	170998	958936	0.00%	1.0	1.26	59.42	300	0.71	0.83
t2d_q9	9801	9801	87025	0.09%	1.0	1.29	0.98	12	0.68	0.76
epb1	14734	14734	95053	0.04%	1.0	1.27	1.03	23	0.68	0.77
raefsky1	3242	3242	294276	2.80%	1.0	1.33	1.19	1.03	0.69	0.71
torso2	115967	115967	1033473	0.01%	1.0	1.29	1.08	131	0.70	0.76
Mean	361922	361132	4448472	1.82%	1.0	1.26	390	928	0.89	0.75
Median	4182	5300	40424	0.23%	1.0	1.27	2.77	5.03	0.91	0.74
Stdev	3339878	3333546	55947269	6.10%	1.0	0.06	6092	10526	0.09	0.04
Min	2	3	3	0.000002%	1.0	0.85	0.55	0.55	0.64	0.57
Max	118142142	118142155	1949412601	76.0%	1.0	1.33	250108	380151	1.01	0.98

MVM Engine Resource Usage

		PEs	LUT/RAM/DSP	GFLOPS		BW
		#	% Area	Peak	Sustained	% Peak
Dense	V5-LX155T	16	72 / 86 / 88	3.1	0.92	64.7
	V6-LX240T	32	71 / 63 / 56	6.4	1.14	80
	V7-LX550T	64	53 / 43 / 24	12.8	N/A	N/A
	Unified V5	16	74 / 87 / 91	3.1	Table III	

Sparse Matrix Growth



Microsoft Research

Results

	GFLOPS / % Peak Bandwidth Used		
	BEE3	HC-1 (32 PE) [14]	Tesla S1070 [14]
dw8192	0.10 / **10.3%**	1.7 / **13.2%**	0.5 / **3.1%**
t2d_q9	0.15 / **14.4%**	2.5 / **19.3%**	0.9 / **5.7%**
epb1	0.17 / **17.1%**	2.6 / **20.2%**	0.8 / **4.9%**
raefsky1	0.20 / **18.5%**	3.9 / **29.0%**	2.6 / **15.3%**
psmigr_2	0.20 / **18.6%**	3.9 / **29.6%**	2.8 / **16.7%**
torso2	0.04 / **4.0%**	1.2 / **9.1%**	3.0 / **18.3%**