

P&S Heterogeneous Systems

Parallel Patterns: Sparse Matrices

Dr. Juan Gómez Luna

Prof. Onur Mutlu

ETH Zürich

Fall 2021

9 December 2021

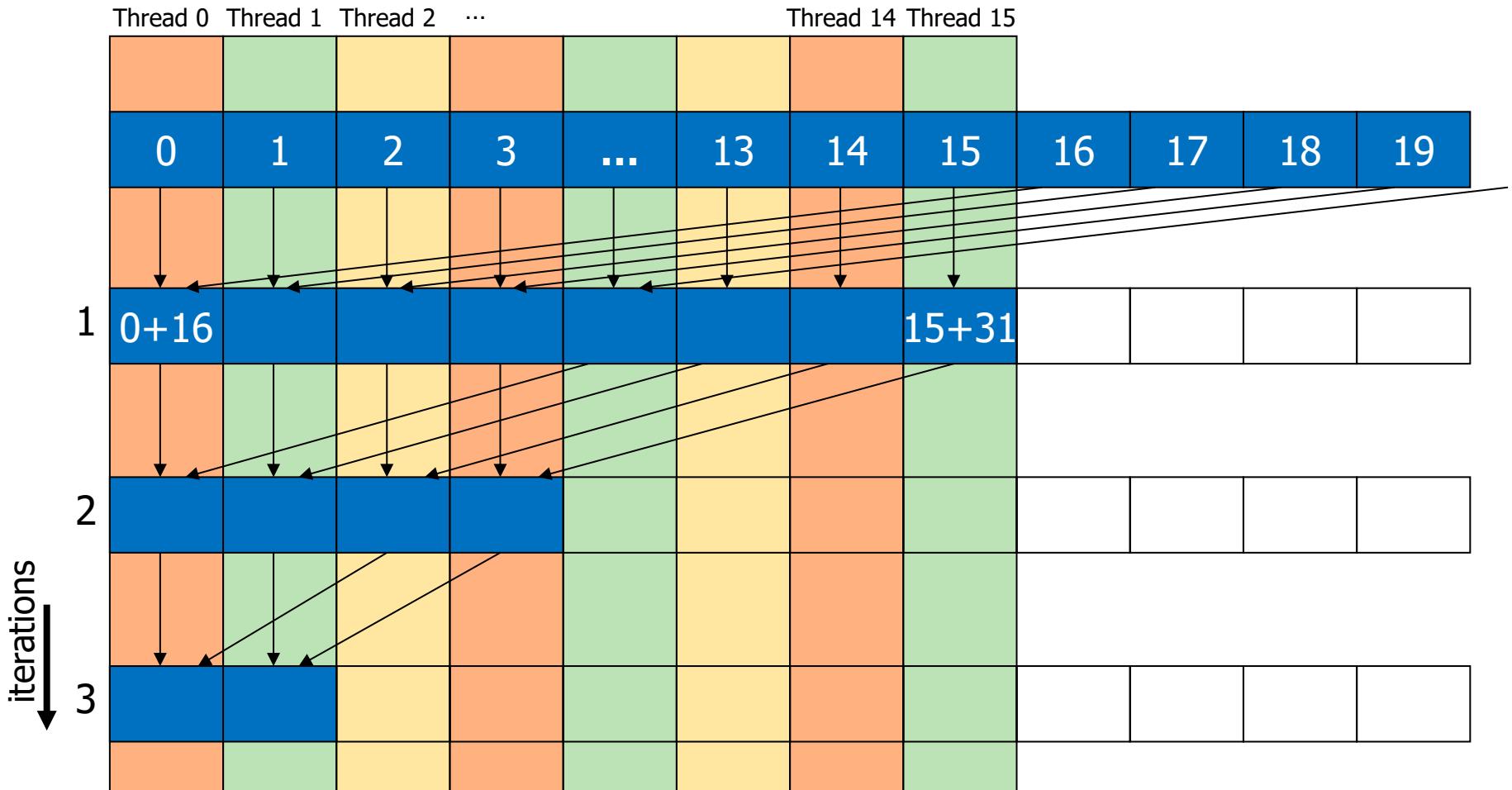
Parallel Patterns

Reduction Operation

- A reduction operation reduces a set of values to a single value
 - Sum, Product, Minimum, Maximum are examples
- Properties of reduction
 - Associativity
 - Commutativity
 - Identity value
- Reduction is a key primitive for parallel computing
 - E.g., MapReduce programming model

Divergence-Free Mapping (I)

- All active threads belong to the same warp

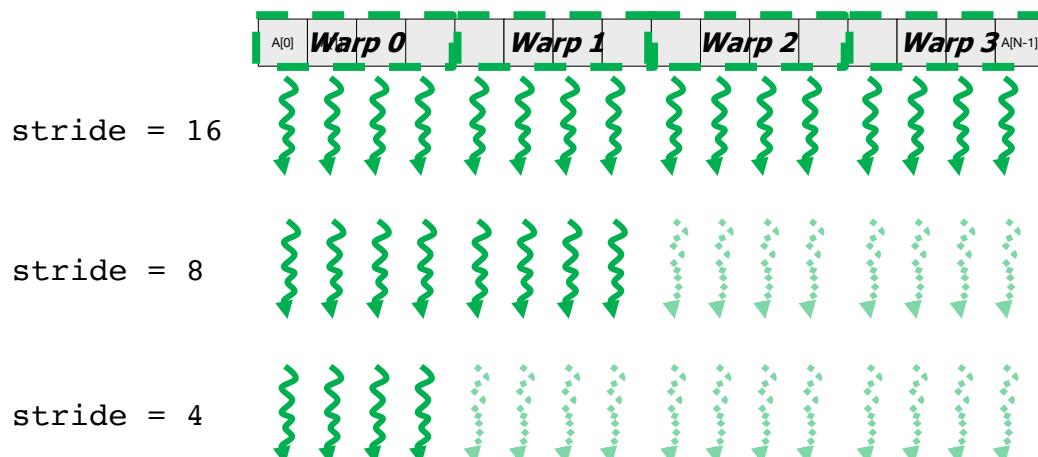


Divergence-Free Mapping (II)

■ Program with high SIMD utilization

```
__shared__ float partialSum[ ]  
  
unsigned int t = threadIdx.x;  
  
for(int stride = blockDim.x; stride > 0; stride >> 1){  
  
    __syncthreads();  
  
    if (t < stride)  
        partialSum[t] += partialSum[t + stride];  
  
}
```

Warp utilization
is maximized

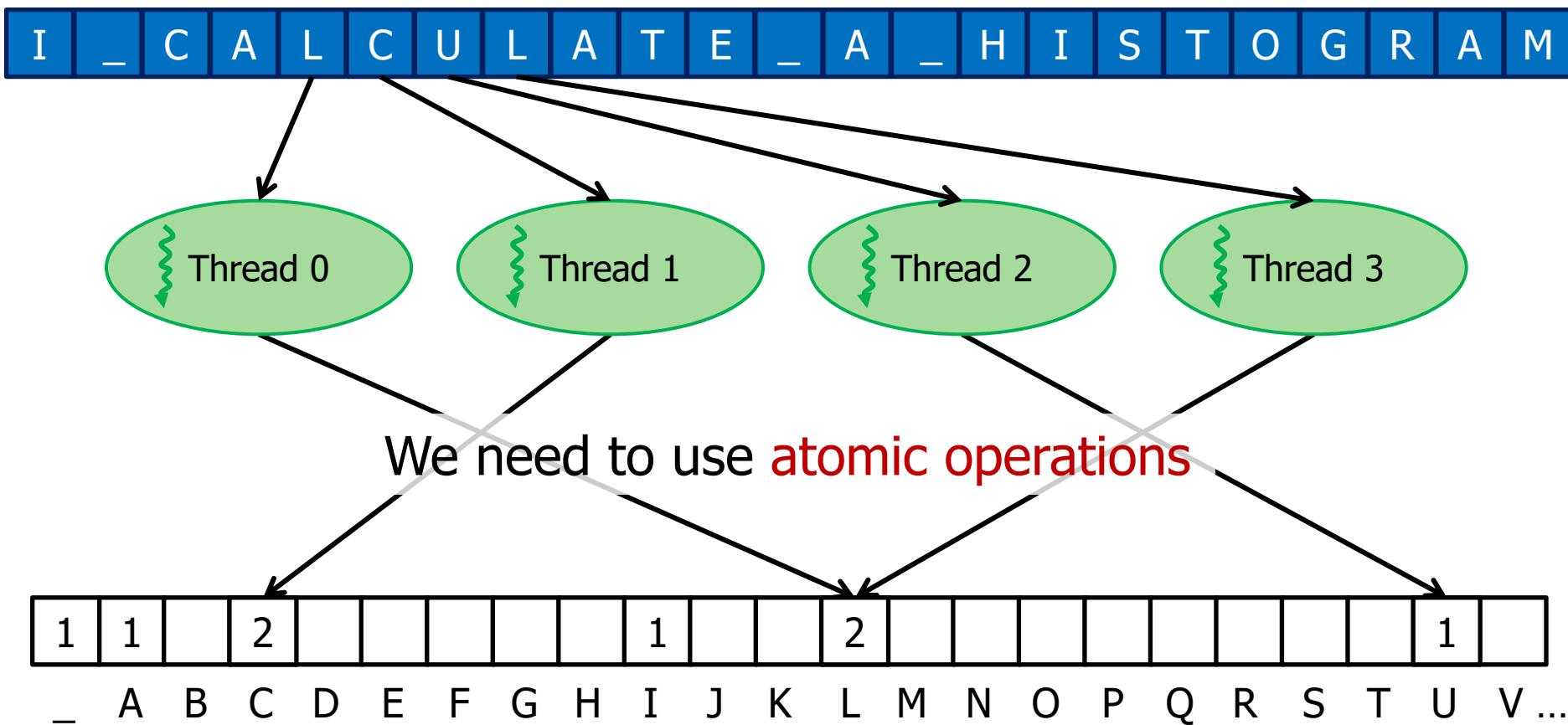


Histogram Computation

- Histogram is a frequently used computation for **reducing the dimensionality and extracting notable features** and patterns from large data sets
 - Feature extraction for object recognition in images
 - Fraud detection in credit card transactions
 - Correlating heavenly object movements in astrophysics
 - ...
- Basic histograms - for **each element in the data set, use the value to identify a “bin” to increment**
 - Divide possible input value range into “bins”
 - Associate a counter to each bin
 - For each input element, examine its value and determine the bin it falls into and increment the counter for that bin

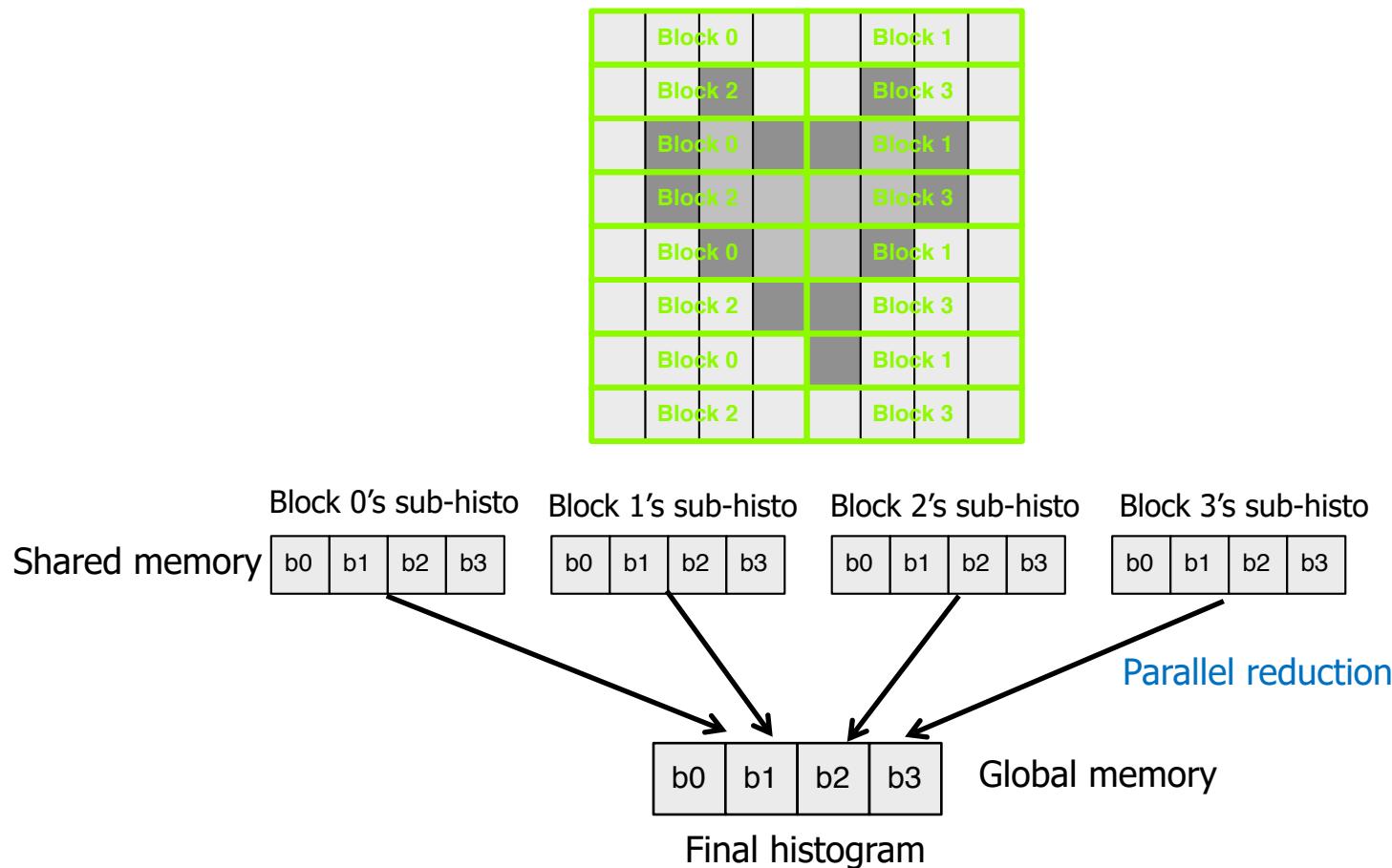
Parallel Histogram Computation: Iteration 2

- All threads move to the next section of the input
 - Each thread moves to element $\text{threadID} + \#\text{threads}$



Histogram Privatization

- **Privatization:** Per-block sub-histograms in shared memory
 - Threads use atomic operations in shared memory



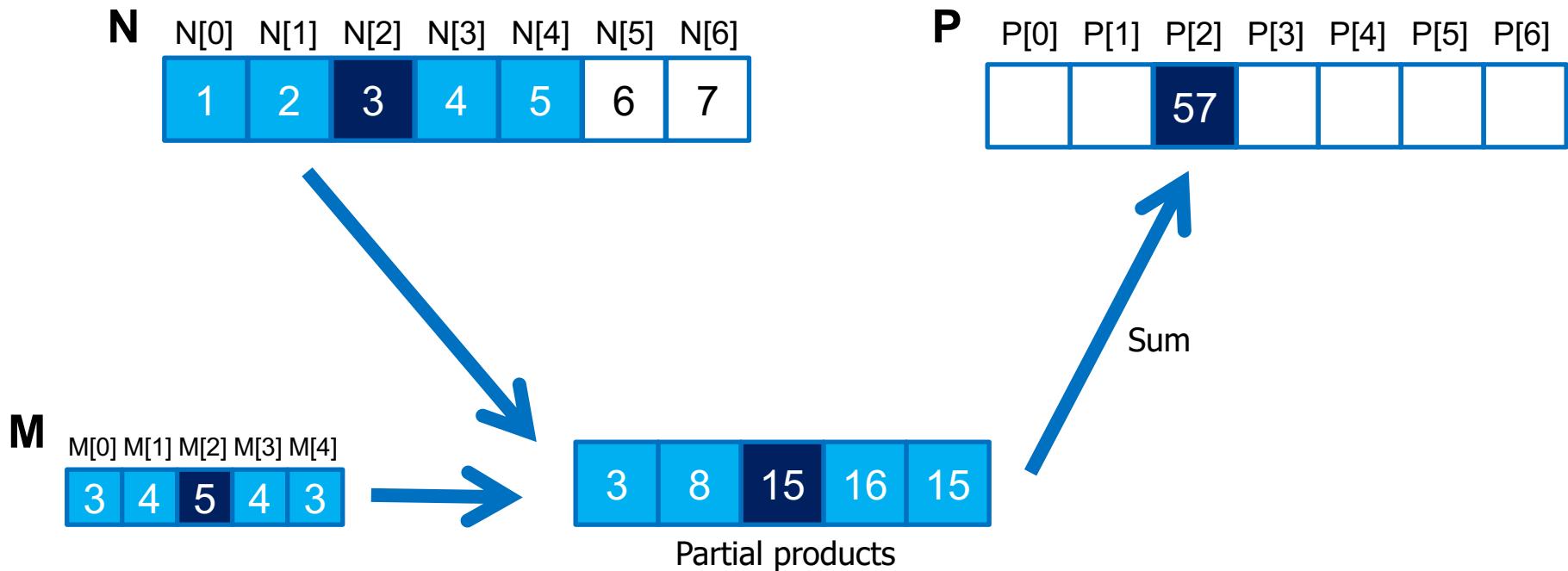
Convolution Applications

- Convolution is a widely-used operation in signal processing, image processing, video processing, and computer vision
- Convolution applies a filter or mask or kernel* on each element of the input (e.g., a signal, an image, a frame) to obtain a new value, which is a weighted sum of a set of neighboring input elements
 - Smoothing, sharpening, or blurring an image
 - Finding edges in an image
 - Removing noise, etc.
- Applications in machine learning and artificial intelligence
 - Convolutional Neural Networks (CNN or ConvNets)

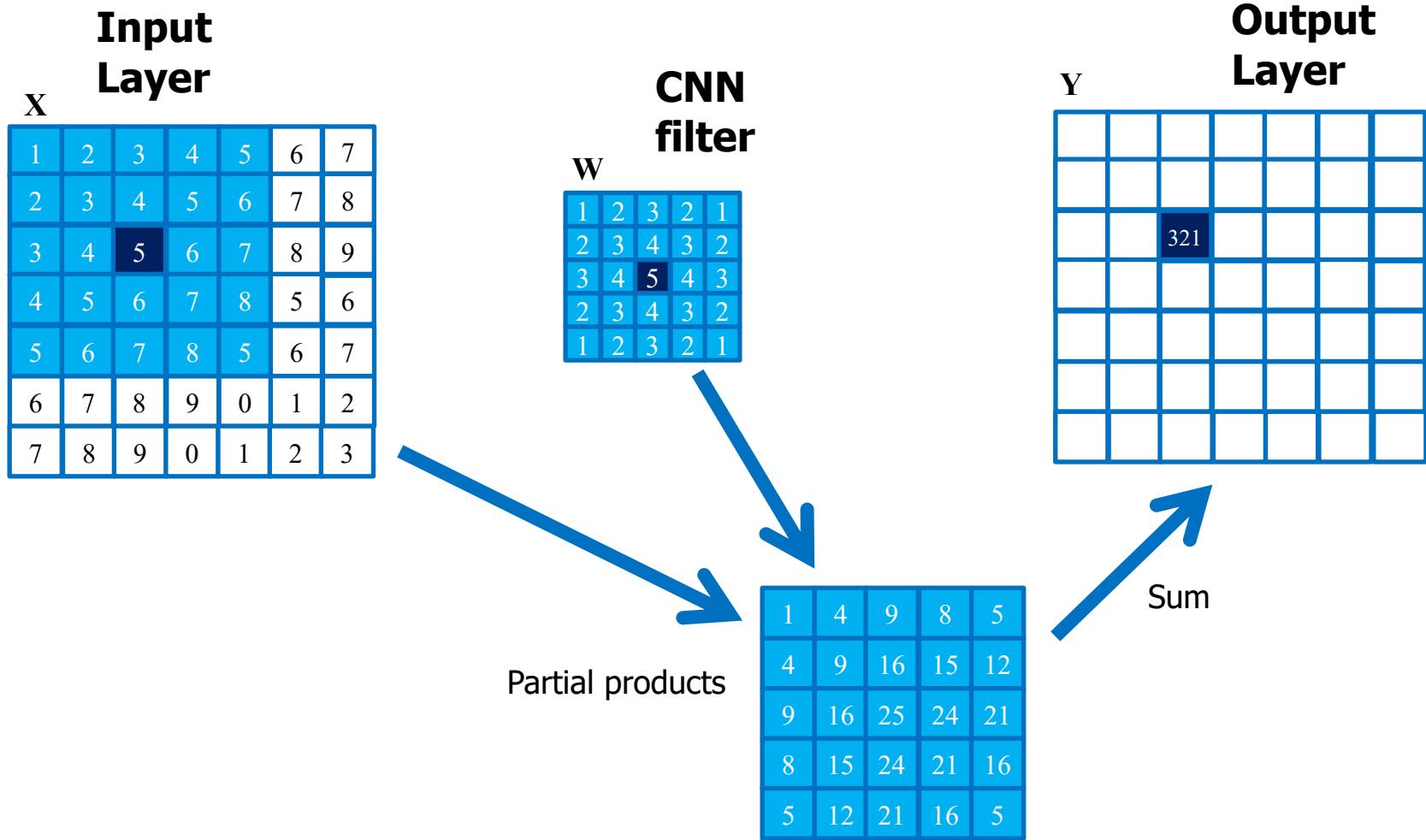
* The term “kernel” may create confusion in the context of GPUs (recall a CUDA/GPU kernel is a function executed by a GPU)

1D Convolution Example

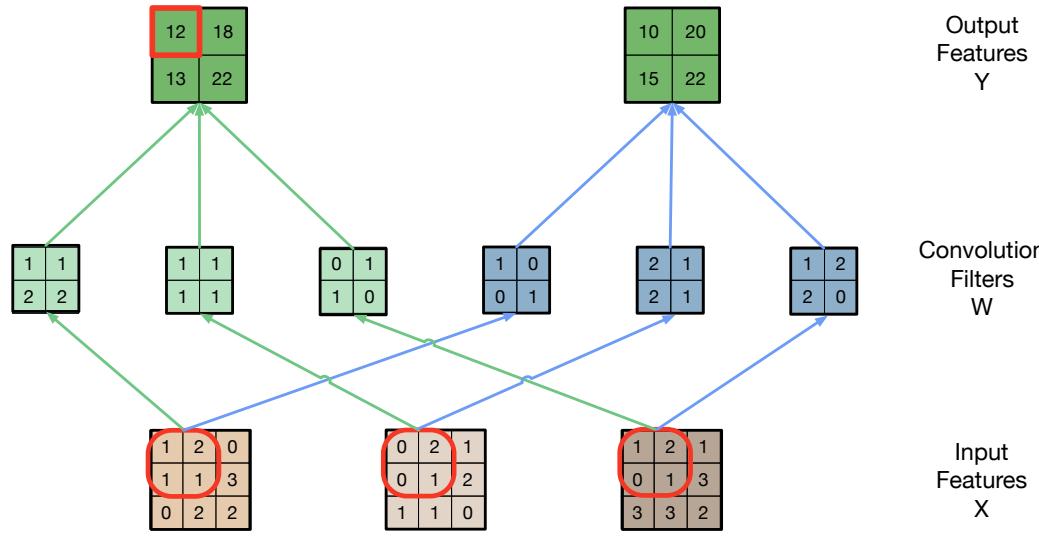
- Commonly used for audio processing
- Mask size is usually **an odd number of elements** for symmetry (5 in this example)
- Calculation of $P[2]$:



Another Example of 2D Convolution



Implementing a Convolutional Layer with Matrix Multiplication



$$\begin{array}{c} \boxed{1 \ 1 \ 2 \ 2 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 1 \ 0} \\ \boxed{1 \ 0 \ 0 \ 1 \ 2 \ 1 \ 2 \ 1 \ 1 \ 2 \ 2 \ 0} \end{array} * \begin{array}{c} \boxed{1 \ 2 \ 1 \ 1} \\ \boxed{2 \ 0 \ 1 \ 3} \\ \boxed{1 \ 1 \ 0 \ 2} \\ \boxed{1 \ 3 \ 2 \ 2} \\ \boxed{0 \ 2 \ 0 \ 1} \\ \boxed{2 \ 1 \ 1 \ 2} \\ \boxed{0 \ 1 \ 1 \ 1} \\ \boxed{1 \ 2 \ 1 \ 0} \\ \boxed{1 \ 2 \ 0 \ 1} \\ \boxed{2 \ 1 \ 1 \ 3} \\ \boxed{0 \ 1 \ 3 \ 2} \\ \boxed{3 \ 3 \ 2 \ 0} \end{array} = \begin{array}{c} \boxed{12 \ 18} \\ \boxed{13 \ 22} \end{array}$$

Convolution Filters W'

Input Features X (unrolled)

Output Features Y

Prefix Sum (Scan)

- Prefix sum or scan is an operation that takes an input array and an associative operator,
 - E.g., addition, multiplication, maximum, minimum
- And returns an output array that is the result of recursively applying the associative operator on the elements of the input array

- Input array $[x_0, x_1, \dots, x_{n-1}]$
- Associative operator \oplus

- An output array $[y_0, y_1, \dots, y_{n-1}]$ where
 - Exclusive scan: $y_i = x_0 \oplus x_1 \oplus \dots \oplus x_{i-1}$
 - Inclusive scan: $y_i = x_0 \oplus x_1 \oplus \dots \oplus x_i$

Hierarchical (Inclusive) Scan

Input **Block 0** **Block 1** **Block 2** **Block 3**

1	2	3	4	1	1	1	1	0	1	2	3	2	2	2	2
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Per-block (Inclusive) Scan

1	3	6	10	1	2	3	4	0	1	3	6	2	4	6	8
---	---	---	----	---	---	---	---	---	---	---	---	---	---	---	---

10	4	6	8
----	---	---	---

Scan Partial Sums

10	14	20	28
----	----	----	----

Inter-block synchronization

- Kernel termination and
 - Scan on CPU, or
 - Launch new scan kernel on GPU
- Atomic operations in global memory

Add

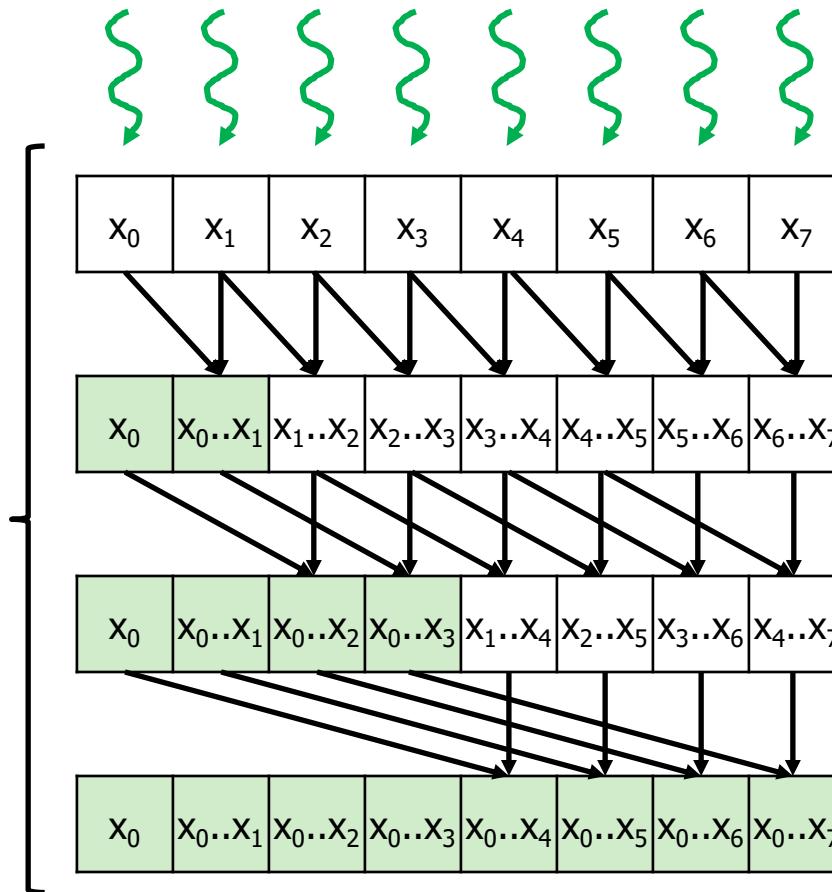
1	3	6	10	1	2	3	4	0	1	3	6	2	4	6	8
---	---	---	----	---	---	---	---	---	---	---	---	---	---	---	---

Output (Inclusive Scan)

1	3	6	10	11	12	13	14	14	15	17	20	22	24	26	28
---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----

Kogge-Stone Parallel (Inclusive) Scan

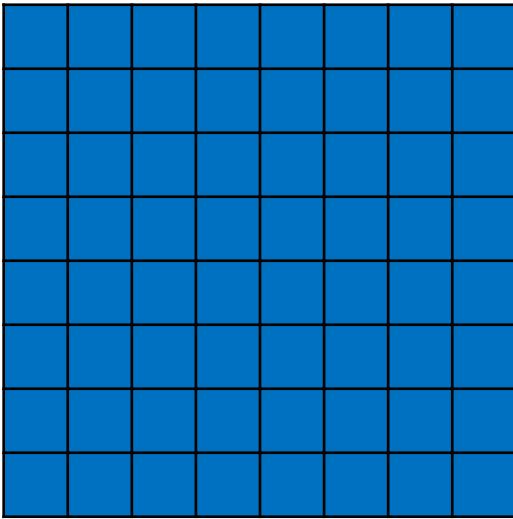
Observation:
memory locations
are reused



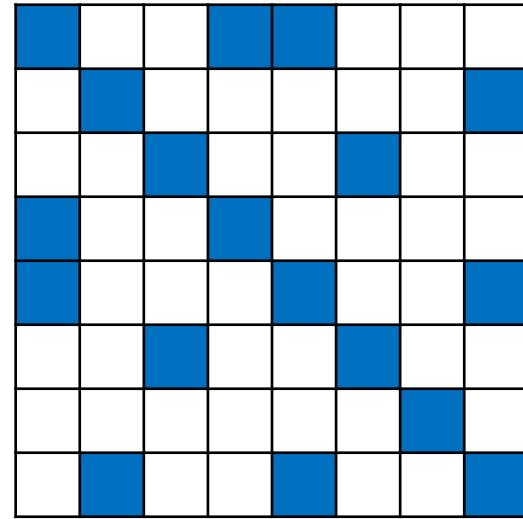
Sparse Matrices and Sparse Matrix Computation

Sparse Matrices

A **dense matrix** is one where the majority of elements are not zero



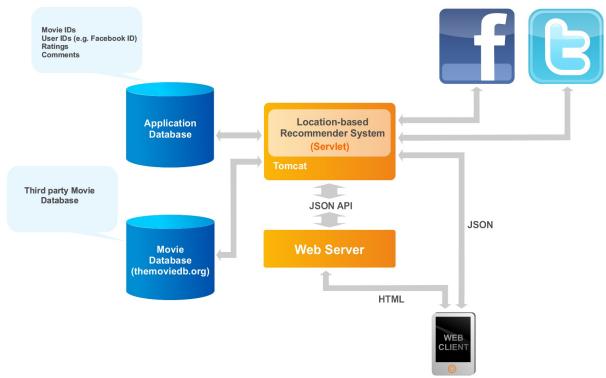
A **sparse matrix** is one where many elements are zero
(many real world systems are sparse)



- Opportunities:
 - Do not need to allocate **space for zeros** (save memory capacity)
 - Do not need to **load zeros** (save memory bandwidth)
 - Do not need to **compute with zeros** (save computation time)

Sparse Matrices are Widespread Today

Recommender Systems



- Collaborative Filtering

Graph Analytics



Neural Networks



- PageRank
- Breadth First Search
- Betweenness Centrality
- Sparse DNNs
- Graph Neural Networks

Real-World Matrices Have High Sparsity



0.0003%
non-zero elements



2.31%
non-zero elements

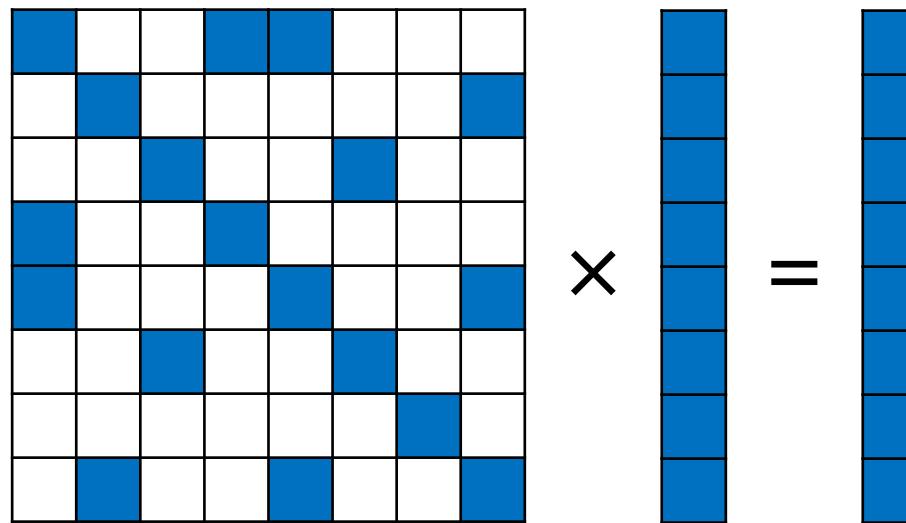
Sparse matrix compression
is essential to enable
efficient storage and computation

Sparse Matrix Storage Formats

- Many storage formats for sparse matrices:
 - Coordinate Format (COO)
 - Compressed Sparse Row (CSR)
 - ELLPACK Format (ELL)
 - Jagged Diagonal Storage (JDS)
 - ...
- Format design considerations:
 - Space efficiency (memory consumed)
 - Flexibility (ease of adding/reordering elements)
 - Accessibility (ease of finding desired data)
 - Memory access pattern (enabling coalescing)
 - Load balance (minimizing control divergence)

SpMV

- Choice of best format depends on computation
 - We will use **Sparse Matrix-Vector multiplication** (SpMV) as an example to study different formats



Coordinate Format (COO)

Matrix:

1	7		
5		3	9
	2	8	
			6

Store every nonzero
along with its row index
and column index

Row:

0	0	1	1	1	2	2	3
---	---	---	---	---	---	---	---

Column:

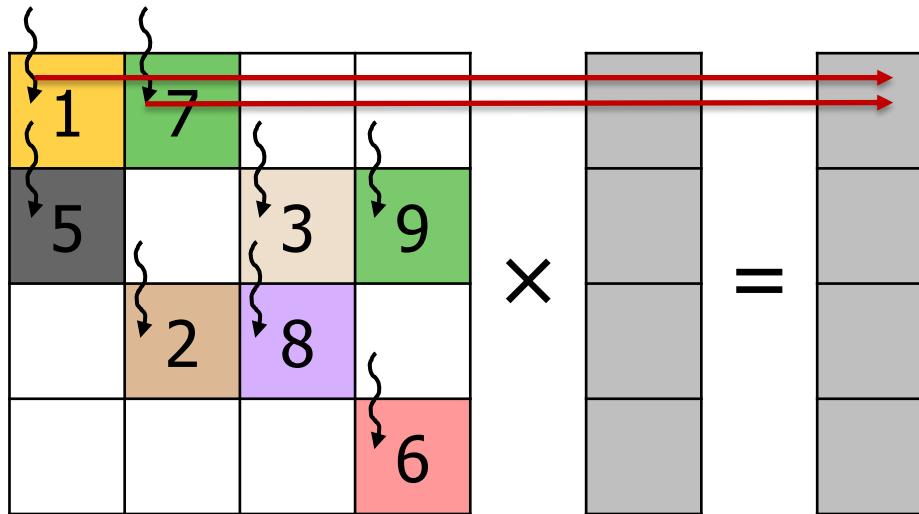
0	1	0	2	3	1	2	3
---	---	---	---	---	---	---	---

Value:

1	7	5	3	9	2	8	6
---	---	---	---	---	---	---	---

SpMV/COO

Matrix:



Multiple threads writing
to the same output
(need atomic
operations)

Row:

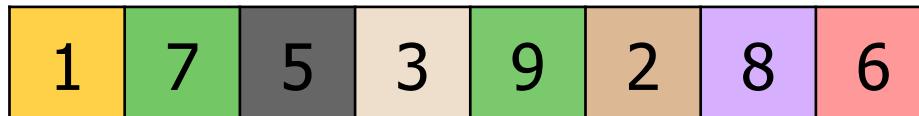


Column:



Parallelization approach:
Assign one thread
per nonzero

Value:



SpMV/COO Code

```
// All elements of outVector are zero-initialized

__global__ void spmv_coo_kernel(cooMatrix cooMatrix, float* invector, float* outvector) {

    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;

    if(i < cooMatrix.numNonzeros) {

        unsigned int row = cooMatrix.rowIdxs[i]; // Read row and column of element i
        unsigned int col = cooMatrix.colIdxs[i];

        float value = cooMatrix.values[i]; // Read value of element i

        atomicAdd(&outVector[row], inVector[col]*value); // Atomic addition
    }
}
```

Recall: Coalesced Atomic Operations

- Identify threads operating on the same atomic and use a reduction

```
int atomic_add(int * ptr, int value){  
  
    unsigned active_mask = __activemask();  
    unsigned active_mask = __match_any_sync(active_mask, ptr);  
  
    int value = reduce_warp(active_mask, value);  
  
    if(__ffs(active_mask) - 1) == lane) {  
  
        value = atomicAdd(ptr, value);  
    }  
  
    value = __shfl_sync(active_mask, value, __ffs(active_mask) - 1);  
    return value;  
}
```

COO Tradeoffs

■ Advantages:

- Flexibility: easy to add new elements to the matrix, nonzeros can be stored in any order
- Accessibility: given nonzero, easy to find row and column
- SpMV/COO has coalesced memory accesses
- SpMV/COO has no control divergence

■ Disadvantages:

- Accessibility: given a row or column, hard to find all nonzeros (need to search)
- SpMV/COO uses atomic operations

Widely Used Format: Compressed Sparse Row

- Compressed Sparse Row (CSR) provides **high compression ratio**
- Used in **multiple libraries & frameworks**:
 - Intel MKL¹
 - TACO²
 - Ligra³
 - Polymer⁴
 - Gunrock⁵
 - Etc.

¹ Intel Math Kernel Library, <http://software.intel.com/en-us/articles/intel-mkl/>

² F. Kjolstad, S. Chou, D. Lugato, S. Kamil, and S. Amarasinghe, "taco: A Tool to Generate Tensor Algebra Kernels," ASE 2017

³ J. Shun and G. E. Blelloch, "Ligra: A Lightweight Graph Processing Framework for Shared Memory," PPoPP 2013

⁴ K. Zhang, R. Chen, and H. Chen, "Numa-aware Graph-structured Analytics," PPoPP 2015

⁵ Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A High-performance Graph Processing Library on the GPU," PPoPP 2016

Compressed Sparse Row (CSR)

Matrix:

1	7		
5		3	9
	2	8	
			6

Store nonzeros of the same row adjacently and an index to the first element of each row

RowPtrs:

0	2	5	7	8
---	---	---	---	---

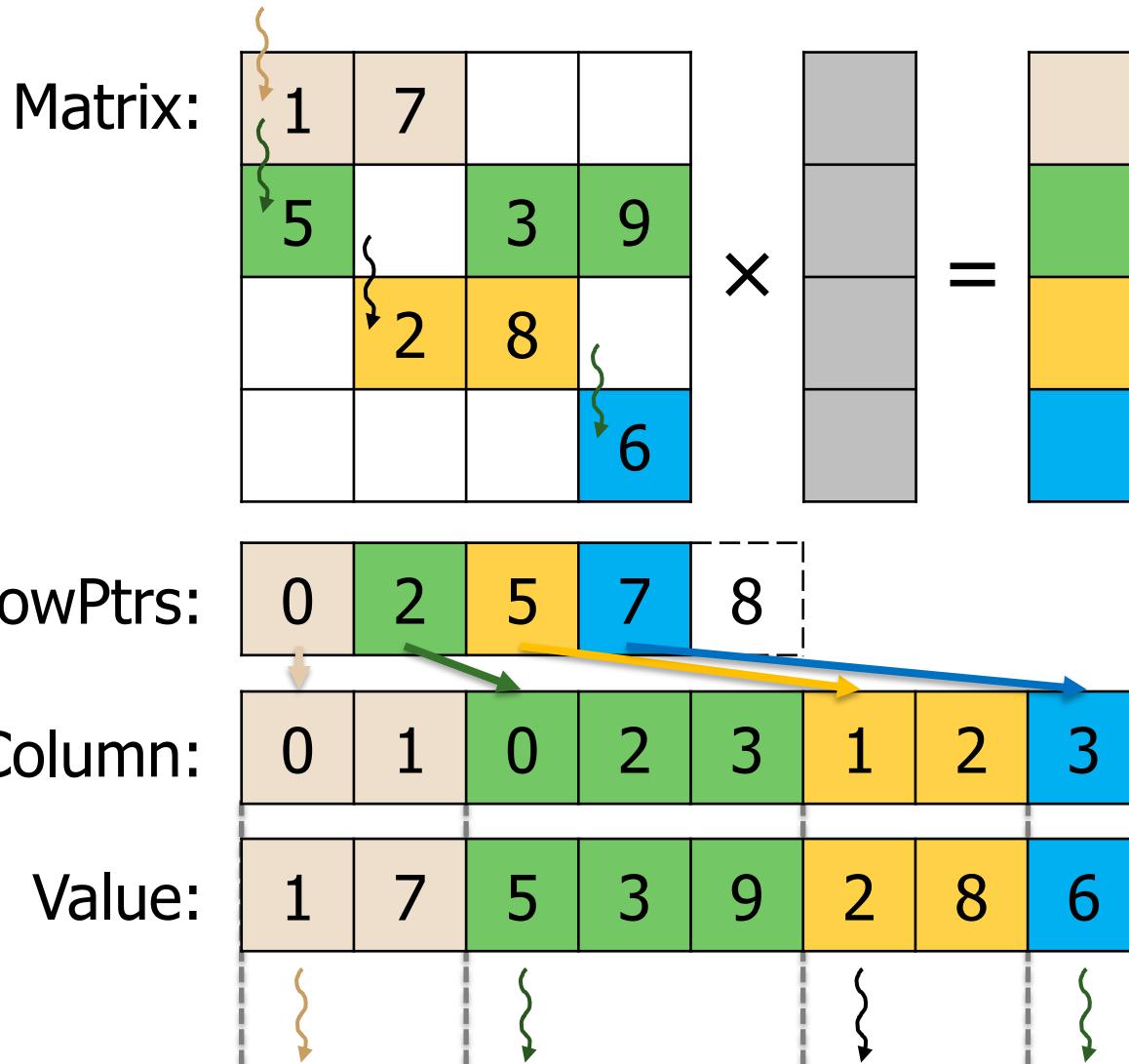
Column:

0	1	0	2	3	1	2	3
---	---	---	---	---	---	---	---

Value:

1	7	5	3	9	2	8	6
---	---	---	---	---	---	---	---

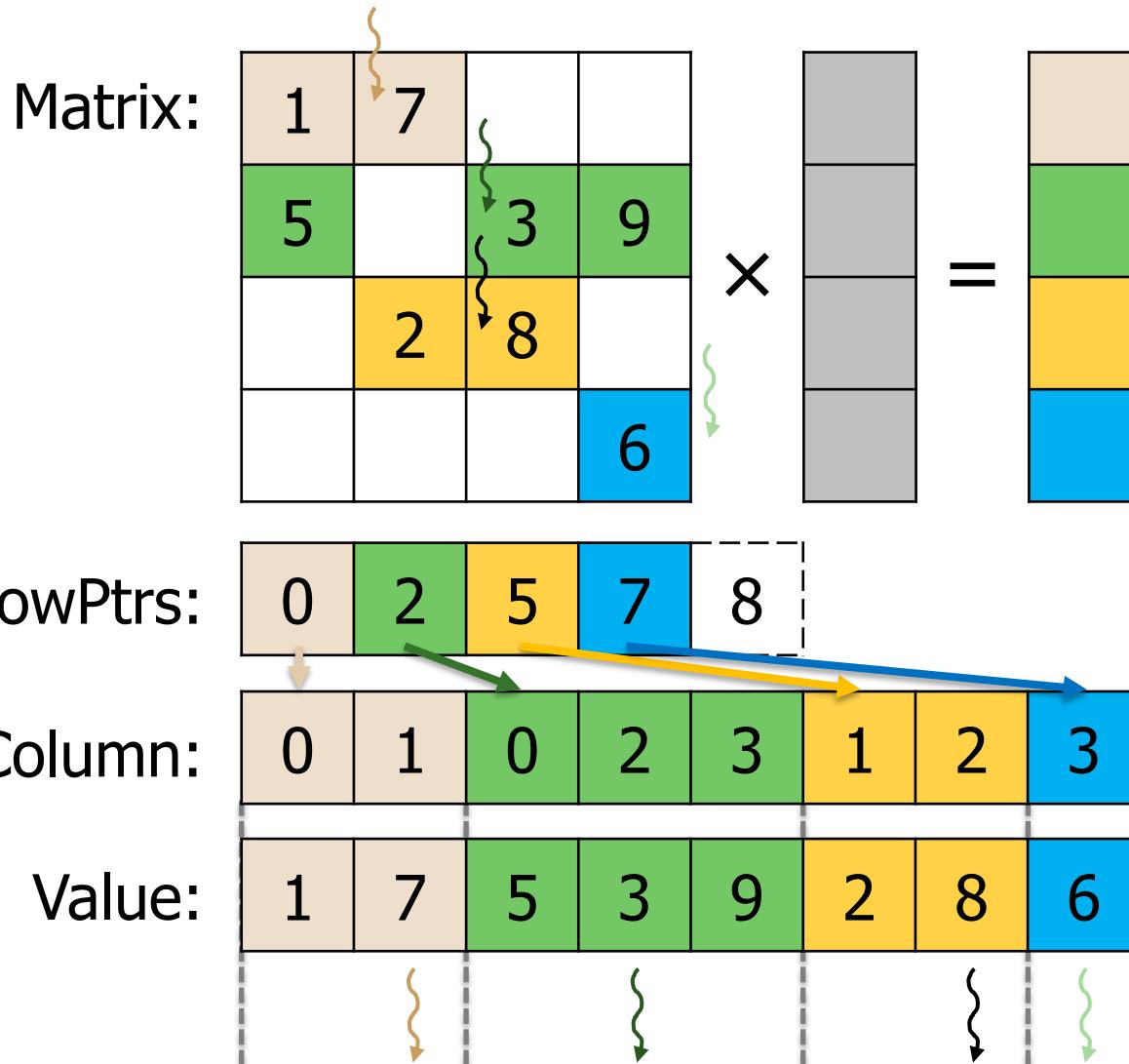
SpMV/CSR



Parallelization approach:

Assign one thread to loop over each input row sequentially and update corresponding output element

SpMV/CSR

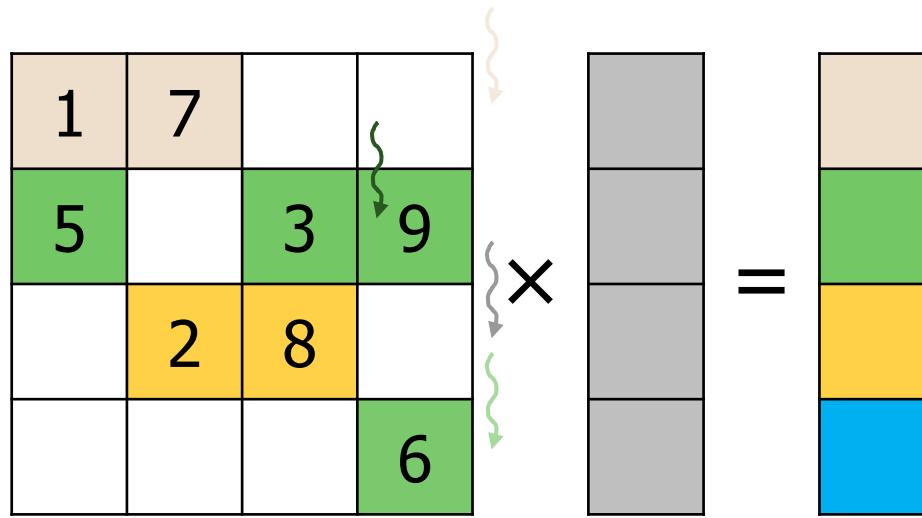


Parallelization approach:

Assign one thread to loop over each input row sequentially and update corresponding output element

SpMV/CSR

Matrix:



Parallelization approach:

Assign one thread to loop over each input row sequentially and update corresponding output element

RowPtrs:



Column:



Value:



SpMV/CSR Code

```
__global__ void spmv_csr_kernel(CSRMatrix csrMatrix, float* inVector, float* outVector) {  
    unsigned int row = blockIdx.x*blockDim.x + threadIdx.x; // Each thread works on one row  
  
    if(row < csrMatrix.numRows) {  
        float sum = 0.0f; // zero initialize temporal accumulator  
  
        // Loop over all elements of a row  
        for(unsigned int i = csrMatrix.rowPtrs[row]; i < csrMatrix.rowPtrs[row + 1]; ++i) {  
            unsigned int col = csrMatrix.colIds[i]; // Read column index  
  
            float value = csrMatrix.values[i]; // Read value  
  
            sum += inVector[col]*value; // Multiply and accumulate  
        }  
  
        outVector[row] = sum;  
    }  
}
```

CSR Tradeoffs (versus COO)

- **Advantages:**
 - Space efficiency: row pointers smaller than row indexes
 - Accessibility: given a row, easy to find all nonzeros
 - SpMV/CSR avoids atomics, every thread owns its output

- **Disadvantages:**
 - Flexibility: hard to add new elements to the matrix
 - Accessibility: given nonzero, hard to find row; given a column, hard to find all nonzeros
 - SpMV/CSR memory accesses are not coalesced
 - SpMV/CSR has control divergence

Compressed Sparse Column (CSC)

Matrix:

1	7		
5		3	9
	2	8	
			6

Like CSR, but groups nonzeros by column

(useful for computations other than SpMV that require column traversal)

ColPtrs:

0	2	4	6	8
---	---	---	---	---

Row:

0	1	0	2	1	2	1	3
---	---	---	---	---	---	---	---

Value:

1	5	7	2	3	8	9	6
---	---	---	---	---	---	---	---

ELLPACK Format (ELL)

Matrix:

1	7		
5		3	9
	2	8	
			6

Group nonzeros by row (like CSR)...

Column:

0	1		
0	2	3	
1	2		
3			

Value:

1	7		
5		3	9
2	8		
6			

ELLPACK Format (ELL)

Matrix:

1	7		
5		3	9
	2	8	
			6

...pad rows so they all have the same size...

Column:

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

Value:

1	7	*
5	3	9
2	8	*
6	*	*

ELLPACK Format (ELL)

Matrix:

1	7		
5		3	9
	2	8	
			6

...and store padded
array of nonzeros in
column major order

Column:

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

Value:

1	7	*
5	3	9
2	8	*
6	*	*

ELLPACK Format (ELL)

Matrix:

1	7		
5		3	9
	2	8	
			6

...and store padded
array of nonzeros in
column major order

Column:

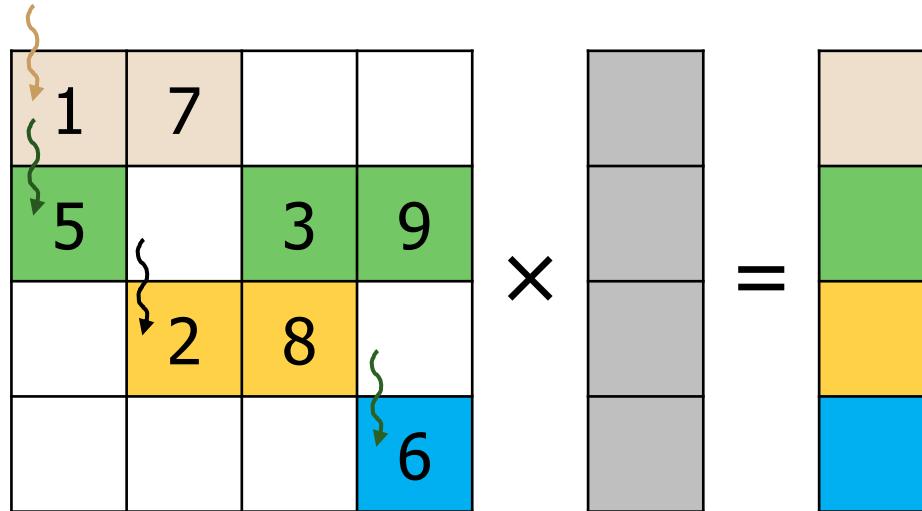
0	0	1	3	1	2	2	*	*	3	*	*
---	---	---	---	---	---	---	---	---	---	---	---

Value:

1	5	2	6	7	3	8	*	*	9	*	*
---	---	---	---	---	---	---	---	---	---	---	---

SpMV/ELL

Matrix:

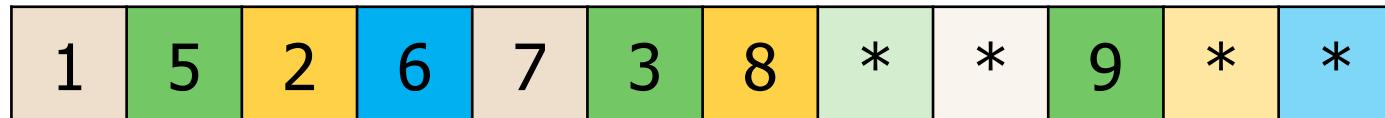


Parallelization approach:
Assign one thread to loop over each input row sequentially and update corresponding output element

Column:



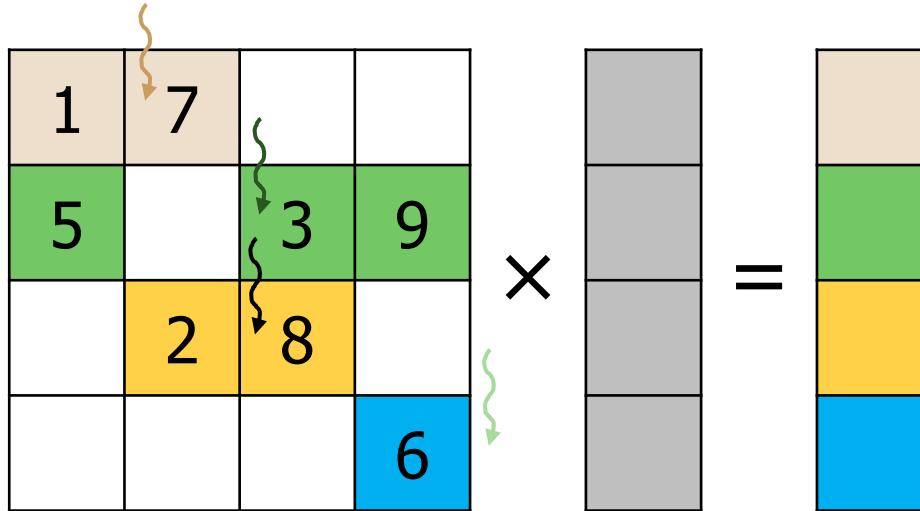
Value:



Memory accesses are coalesced

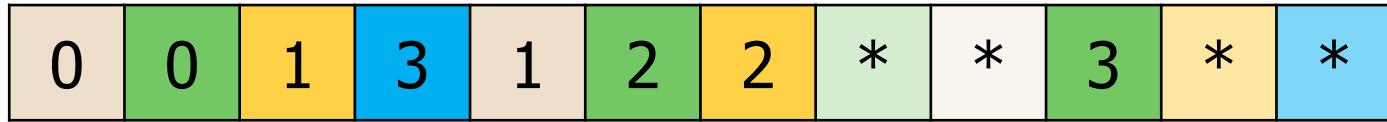
SpMV/ELL

Matrix:

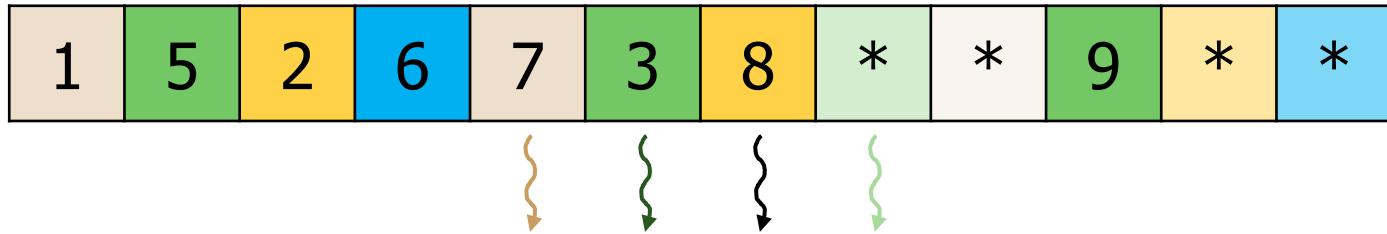


Parallelization approach:
Assign one thread to loop over each input row sequentially and update corresponding output element

Column:



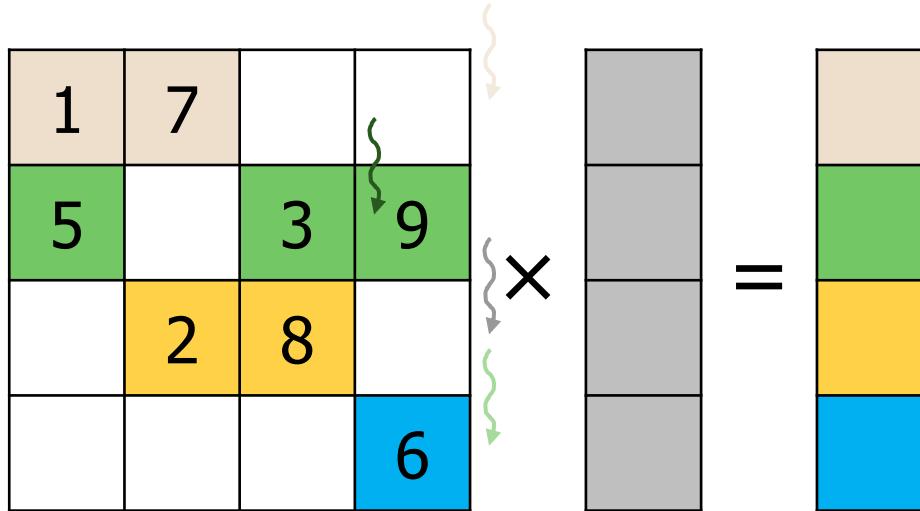
Value:



Memory accesses are coalesced

SpMV/ELL

Matrix:



Parallelization approach:

Assign one thread to loop over each input row sequentially and update corresponding output element

Column:

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

Value:

1	5	2	6	7	3	8	*	*	9	*	*
---	---	---	---	---	---	---	---	---	---	---	---



Memory accesses are coalesced

SpMV/ELL Code

```
__global__ void spmv_ell_kernel(ELLMatrix ellMatrix, float* inVector, float* outVector) {  
    unsigned int row = blockIdx.x*blockDim.x + threadIdx.x; // Each thread works on one row  
  
    if(row < ellMatrix.numRows) {  
        float sum = 0.0f; // zero initialize temporal accumulator  
  
        // Loop over all non-zero elements in the own row  
        for(unsigned int nnzIdx = 0; nnzIdx < ellMatrix.nzPerRow[row]; ++nnzIdx) {  
            unsigned int i = nnzIdx*ellMatrix.numRows + row; // Index of non-zero element  
            unsigned int col = ellMatrix.colIds[i]; // Read column index  
            float value = ellMatrix.values[i]; // Read value  
            sum += inVector[col]*value; // Multiply and accumulate  
        }  
        outVector[row] = sum;  
    }  
}
```

ELL Tradeoffs

■ Advantages:

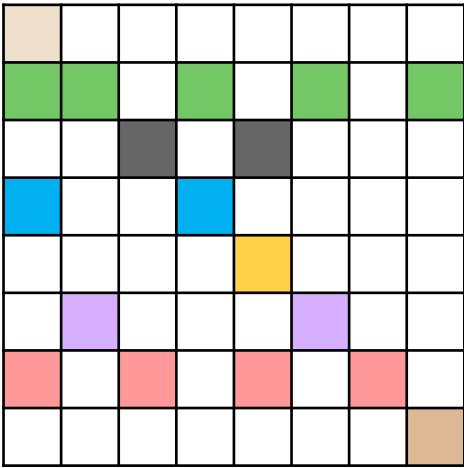
- Flexibility: can add new elements as long as row not full
- Accessibility: given a row, easy to find all nonzeros; given nonzero, easy to find row and column
- SpMV/ELL memory accesses are coalesced

■ Disadvantages:

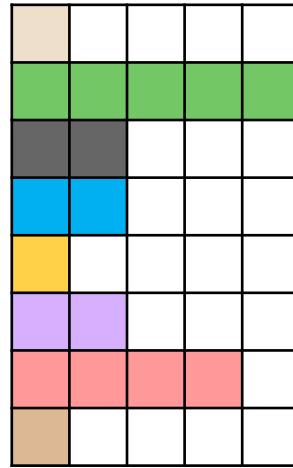
- Space efficiency: overhead due to padding
- Accessibility: given a column, hard to find all nonzeros
- SpMV/ELL has control divergence

Hybrid ELL + COO

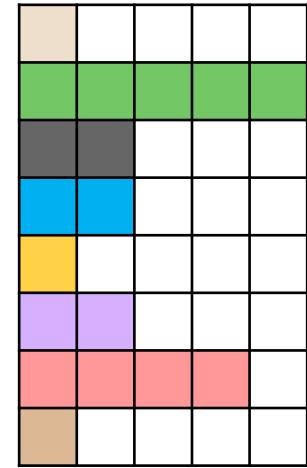
Matrix:



Column:



Value:

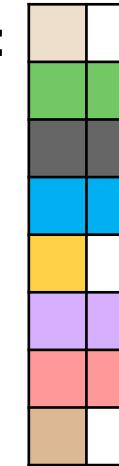


ELL Format

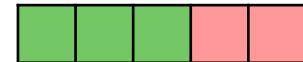
Column_{ELL}:



Value_{ELL}:



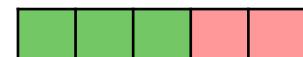
Row_{COO}:



Column_{COO}:



Value_{COO}:



Hybrid ELL + COO
(use COO for very long rows)

ELL + COO Tradeoffs

- Similar to ELL, with the following added benefits from using COO:
 - Space efficiency: less padding
 - Flexibility: can add new elements to any row
 - Less control divergence

Jagged Diagonal Storage (JDS)

Matrix:

a					
b	c		d	e	
		f		g	
h			i		j
	k			l	
m		n			o

Group nonzeros by row (like CSR)...

Column:

0				
0	1	3	4	
2	4			
0	3	5		
1	4			
0	2	5		

Value:

a				
b	c	d	e	
f	g			
h	i	j		
k	l			
m	n	o		

Jagged Diagonal Storage (JDS)

Matrix:

a					
b	c		d	e	
		f		g	
h			i		j
	k			l	
m		n			o

...sort rows by size
and remember the
original row
index...

Column:

0	1	3	4
0	3	5	
0	2	5	
2	4		
1	4		
0			

Value:

b	c	d	e
h	i	j	
m	n	o	
f	g		
k	l		
a			

Row:

1
3
5
2
4
0

Jagged Diagonal Storage (JDS)

Matrix:

a					
b	c		d	e	
		f		g	
h			i		j
	k			l	
m		n			o

...store nonzeros in
column major...

Column:

0	1	3	4
0	3	5	
0	2	5	
2	4		
1	4		
0			

Value:

b	c	d	e
h		i	
m	n	o	
f	g		
k			
a			

Row:

1
3
5
2
4
0

Jagged Diagonal Storage (JDS)

Matrix:

a						
b	c		d	e		
		f		g		
h			i		j	
	k			l		
m		n			o	

...store nonzeros in
column major...

Column:

0	0	0	2	1	0	1	3	2	4	4	3	5	5	4
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Value:

b	h	m	f	k	a	c	i	n	g	l	d	j	o	e
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Row:

1
3
5
2
4
0

Jagged Diagonal Storage (JDS)

Matrix:

a						
b	c		d	e		
		f		g		
h			i		j	
	k			l		
m		n			o	

...and remember
where the nonzeros
of each iteration
start

IterPtr:

0	6	11	14	15	
---	---	----	----	----	--

Row:

1
3
5
2
4
0

Column:

0	0	0	2	1	0	1	3	2	4	4	3	5	5	4
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Value:

b	h	m	f	k	a	c	i	n	g	l	d	j	o	e
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

SpMV/JDS Kernel

Matrix:

a						
b	c		d	e		
		f		g		
h			i		j	
	k			l		
m		n			o	

×

=

Parallelization approach:
Assign one thread to loop
over each input row
sequentially and update
corresponding output
element

IterPtr:

0	6	11	14	15
---	---	----	----	----

Row:

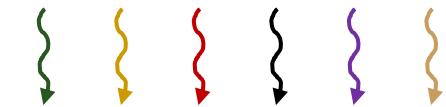
1
3
5
2
4
0

Column:

0	0	0	2	1	0	1	3	2	4	4	3	5	5	4
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Value:

b	h	m	f	k	a	c	i	n	g	l	d	j	o	e
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

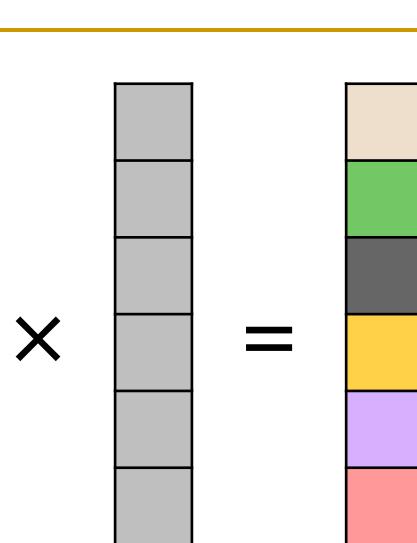


Memory accesses are coalesced...

SpMV/JDS Kernel

Matrix:

a							
b	c		d	e			
		f		g			
h			i		j		
	k			l			
m		n			o		



Parallelization approach:

Assign one thread to loop over each input row sequentially and update corresponding output element

IterPtr:

0	6	11	14	15
---	---	----	----	----

Row:

1
3
5
2
4
0

Column:

0	0	0	2	1	0	1	3	2	4	4	3	5	5	4
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Value:

b	h	m	f	k	a	c	i	n	g	l	d	j	o	e
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

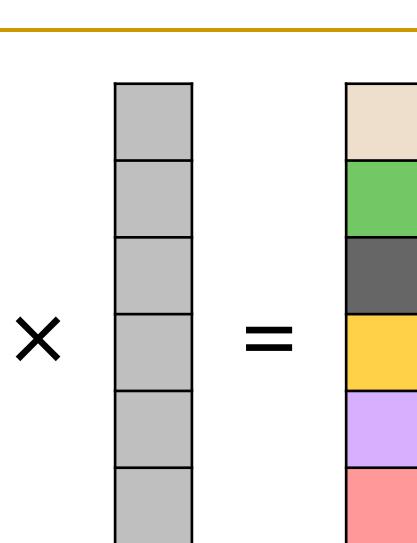
Memory accesses are coalesced...

...and threads drop out from the end, minimizing control divergence

SpMV/JDS Kernel

Matrix:

a						
b	c		d	e		
		f		g	j	
h			i		j	
	k			l		o
m		n				o



Parallelization approach:

Assign one thread to loop over each input row sequentially and update corresponding output element

IterPtr:

0	6	11	14	15
---	---	----	----	----

Row:

1
3
5
2
4
0

Column:

0	0	0	2	1	0	1	3	2	4	4	3	5	5	4
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

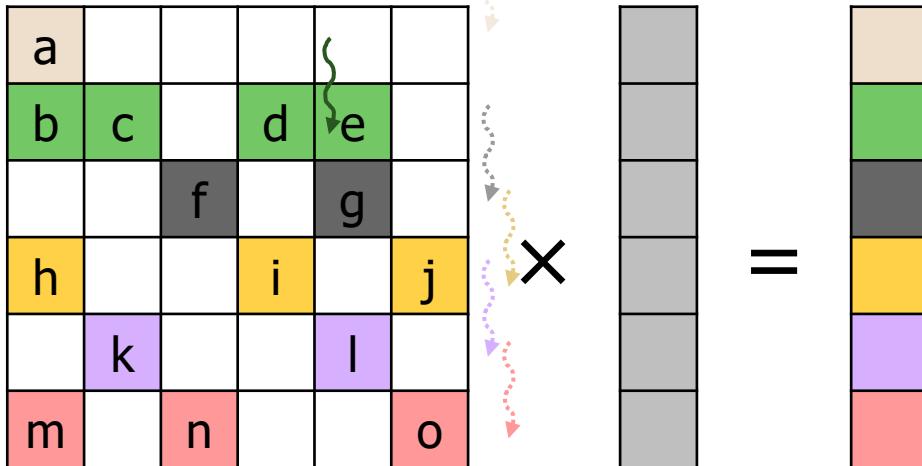
Value:

b	h	m	f	k	a	c	i	n	g	l	d	j	o	e
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Memory accesses are coalesced...
...and threads drop out from the end, minimizing control divergence

SpMV/JDS Kernel

Matrix:



Parallelization approach:

Assign one thread to loop over each input row sequentially and update corresponding output element

IterPtr:

0	6	11	14	15
---	---	----	----	----

Row:

1
3
5
2
4
0

Column:

0	0	0	2	1	0	1	3	2	4	4	3	5	5	4
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Value:

b	h	m	f	k	a	c	i	n	g	l	d	j	o	e
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Memory accesses are coalesced...

...and threads drop out from the end, minimizing control divergence

JDS Tradeoffs

- **Advantages:**
 - Space efficiency: no padding
 - Accessibility: given a row, easy to find all nonzeros
 - SpMV/JDS memory accesses are coalesced
 - SpMV/JDS minimizes control divergence

- **Disadvantages:**
 - Flexibility: hard to add new elements to the matrix
 - Accessibility: given nonzero, hard to find row; given a column, hard to find all nonzeros

Indexing Overhead in Sparse Kernels

Sparse Matrix Vector Multiplication (SpMV)

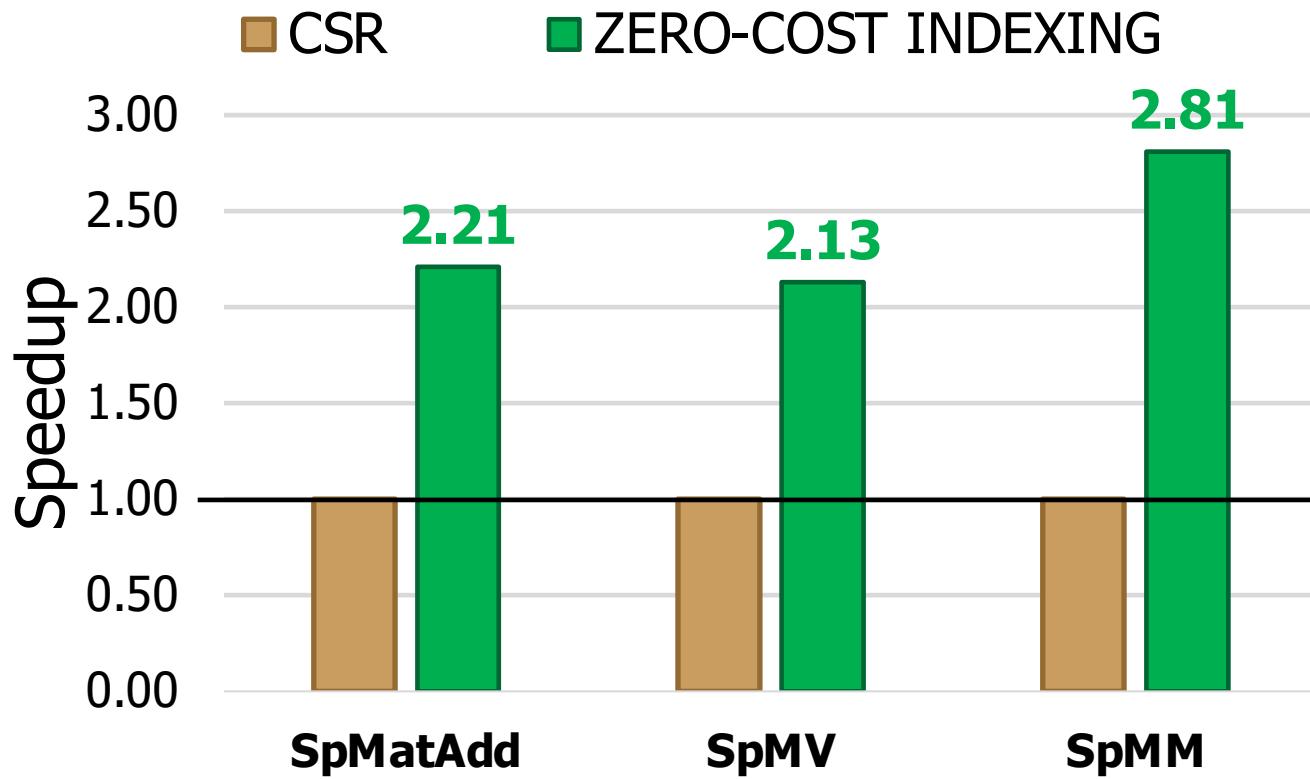
Indexing for every non-zero element of the sparse matrix to multiply with the corresponding element of the vector

Sparse Matrix Matrix Multiplication (SpMM)

Index matching for every inner product between the 2 sparse matrices

Indexing is expensive
for major sparse matrix kernels

Performing Indexing with Zero Cost



Reducing the cost of indexing
can accelerate sparse matrix operations

Limitations of Existing Compression Formats

1

General formats
optimize for storage



Expensive discovery of the
positions
of non-zero elements

2

Specialized formats assume
specific matrix structures
and patterns (e.g., diagonals)



Narrow
applicability

What is an Ideal Compression Format?

- A sparse matrix compression mechanism that:
 - Minimizes the indexing overheads
 - Can be used across a wide range of sparse matrices and sparse matrix operations
 - Enables high compression ratio
- SMASH* proposes a bitmap-based compression format

SMASH: Key Idea

Hardware/Software cooperative mechanism:

- Enables **highly-efficient** sparse matrix compression and computation
- **General** across a diverse set of sparse matrices and sparse matrix operations

Software

Efficient
compression using
a Hierarchy of
Bitmaps

Hardware

Unit that scans
bitmaps to
accelerate indexing

SMASH ISA

SMASH: Software Compression Scheme (I)

Software

Efficient
compression using
a Hierarchy of
Bitmaps

- Encodes the positions of non-zero elements using bits to maintain **low storage overhead**

SMASH: Software Compression Scheme (II)

BITMAP:

Encodes if a block of the matrix contains any non-zero element

MATRIX

NZ			
	NZ		
		NZ	NZ

BITMAP

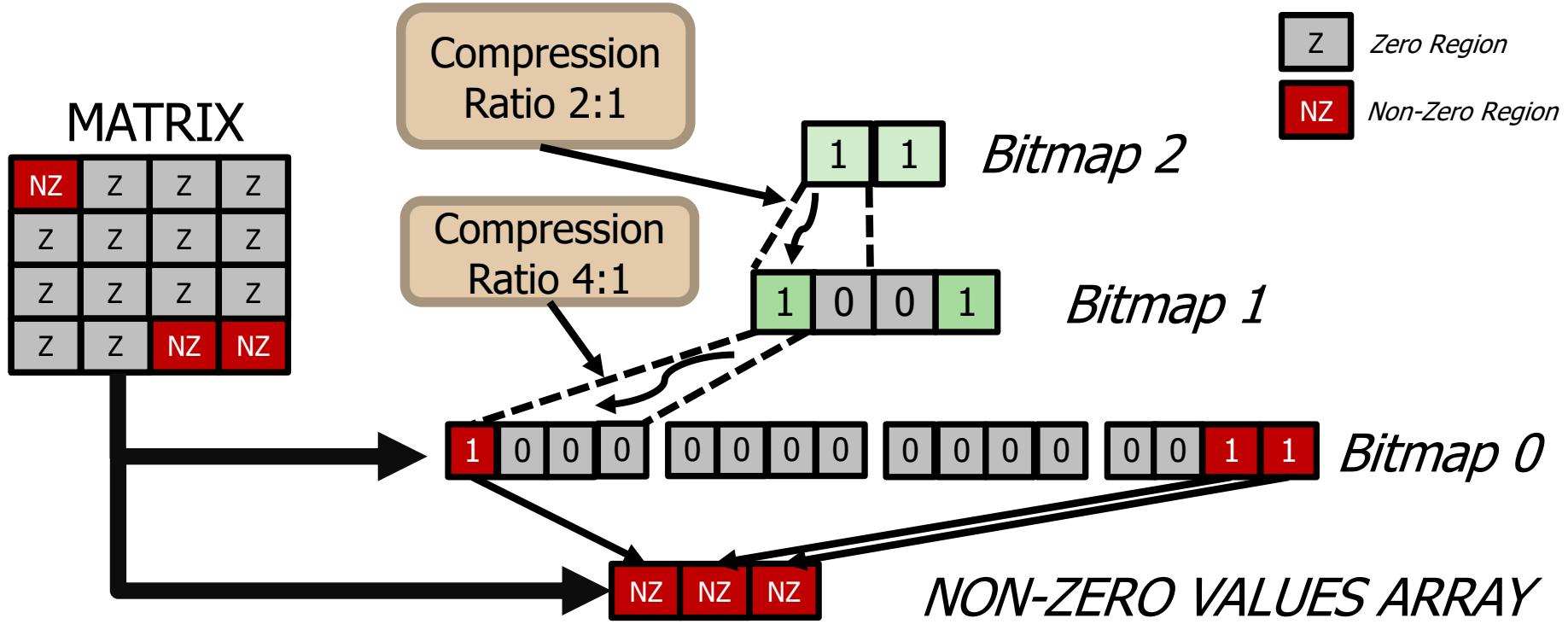
1			
	1		
		1	1



Might contain high number of zero bits

Idea: Apply the same encoding recursively to compress more effectively

Hierarchy of Bitmaps

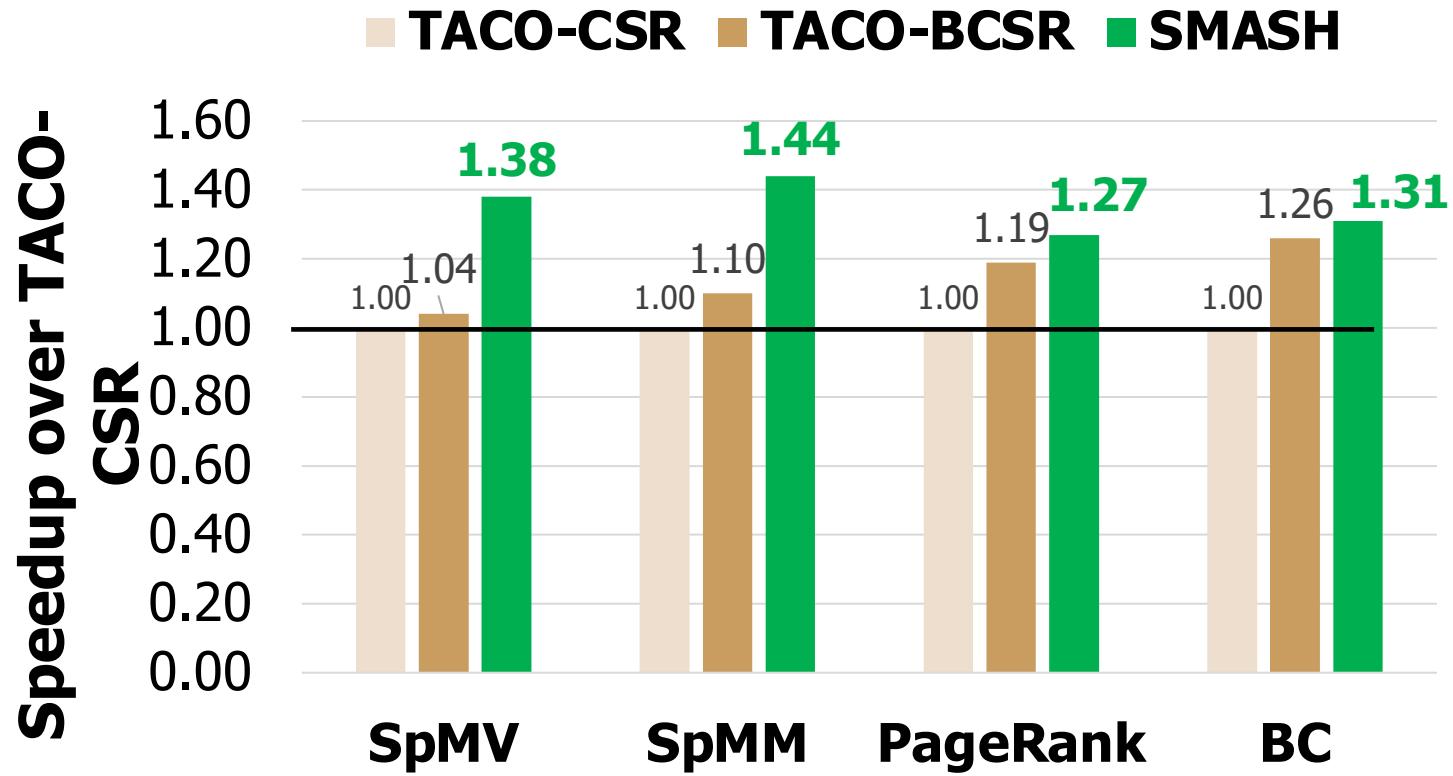


Storage
Efficient

Fast
Indexing

Hardware
Friendly

Performance Improvement Using SMASH

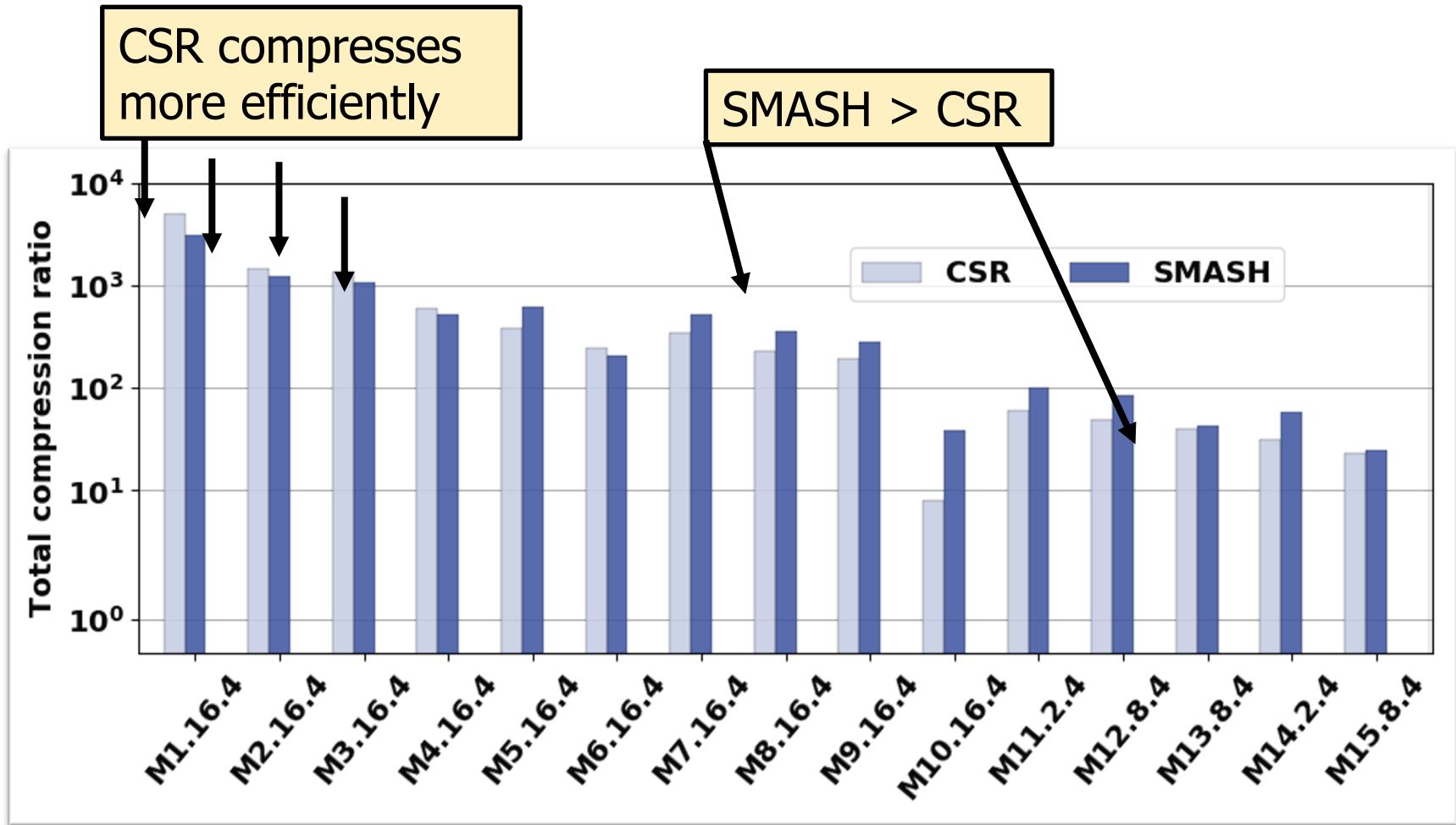


SMASH provides significant performance improvement over state-of-the-art formats
(even software-only SMASH outperforms TACO-CSR)

Evaluated Sparse Matrices

	Name	# Rows	Non-Zero Elements	Sparsity (%)
M1:	descriptor_xingo6u	20,738	73,916	0.01
M2:	g7jac060sc	17,730	183,325	0.06
M3:	Trefethen_20000	20,000	554,466	0.14
M4:	IG5-16	18,846	588,326	0.17
M5:	TSOPF_RS_b162_c3	15,374	610,299	0.26
M6:	ns3Da	20,414	1,679,599	0.40
M7:	tsyl201	20,685	2,454,957	0.57
M8:	pkustk07	16,860	2,418,804	0.85
M9:	ramage02	16,830	2,866,352	1.01
M10:	pattern1	19,242	9,323,432	2.52
M11:	gupta3	16,783	9,323,427	3.31
M12:	nd3k	9,000	3,279,690	4.05
M13:	human_gene1	22,283	24,669,643	4.97
M14:	exdata_1	6,001	2,269,500	6.30
M15:	human_gene2	14,340	18,068,388	8.79

Storage Efficiency



SMASH for Efficient Sparse Matrix Operations

- Konstantinos Kanellopoulos, Nandita Vijaykumar, Christina Giannoula, Roknoddin Azizi, Skanda Koppula, Nika Mansouri Ghiasi, Taha Shahroodi, Juan Gomez-Luna, and Onur Mutlu, **"SMASH: Co-designing Software Compression and Hardware-Accelerated Indexing for Efficient Sparse Matrix Operations"**

Proceedings of the 52nd International Symposium on Microarchitecture (MICRO), Columbus, OH, USA, October 2019.

[[Slides \(pptx\)](#) ([pdf](#))]

[[Lightning Talk Slides \(pptx\)](#) ([pdf](#))]

[[Poster \(pptx\)](#) ([pdf](#))]

[[Lightning Talk Video](#) (90 seconds)]

[[Full Talk Lecture](#) (30 minutes)]

SMASH: Co-designing Software Compression and Hardware-Accelerated Indexing for Efficient Sparse Matrix Operations

Konstantinos Kanellopoulos¹ Nandita Vijaykumar^{2,1} Christina Giannoula^{1,3} Roknoddin Azizi¹
Skanda Koppula¹ Nika Mansouri Ghiasi¹ Taha Shahroodi¹ Juan Gomez Luna¹ Onur Mutlu^{1,2}

¹ETH Zürich

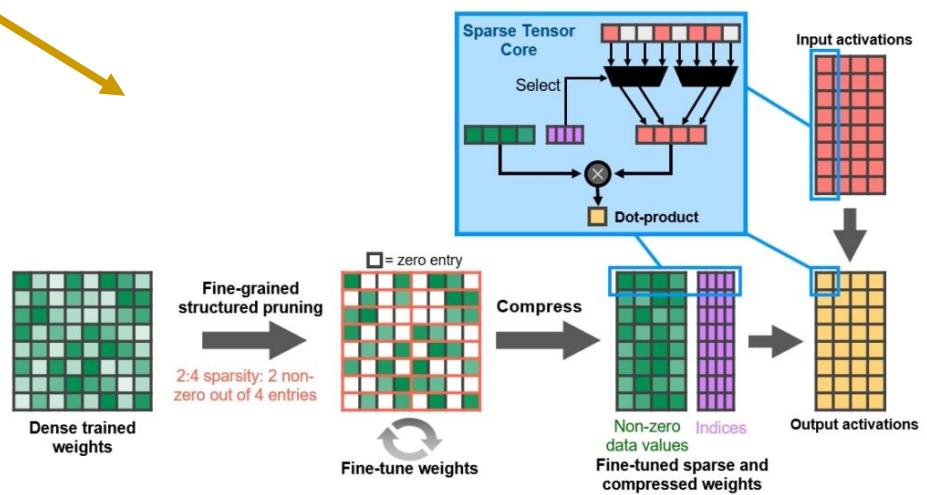
²Carnegie Mellon University

³National Technical University of Athens

Recall: NVIDIA A100 Core



19.5 TFLOPS Single Precision
9.7 TFLOPS Double Precision
312 TFLOPS for Deep Learning (Tensor cores)



Characteristics of Tensor Core Units (TCUs)

$$\begin{matrix} D \\ \text{Fp32} \end{matrix} = \begin{matrix} A \\ \text{Fp16} \end{matrix} *_{(\text{Fp32})} \begin{matrix} B \\ \text{Fp16} \end{matrix} +_{(\text{Fp32})} \begin{matrix} C \\ \text{Fp32} \end{matrix}$$

$$D = A \times B + C$$

or

$$C = A \times B + C$$

Definitions

- TCUs perform matrix multiplication between “small” matrices
- “Small” is e.g. 16x16
- Mixed precision:
Result/Accumulator can be in FP32

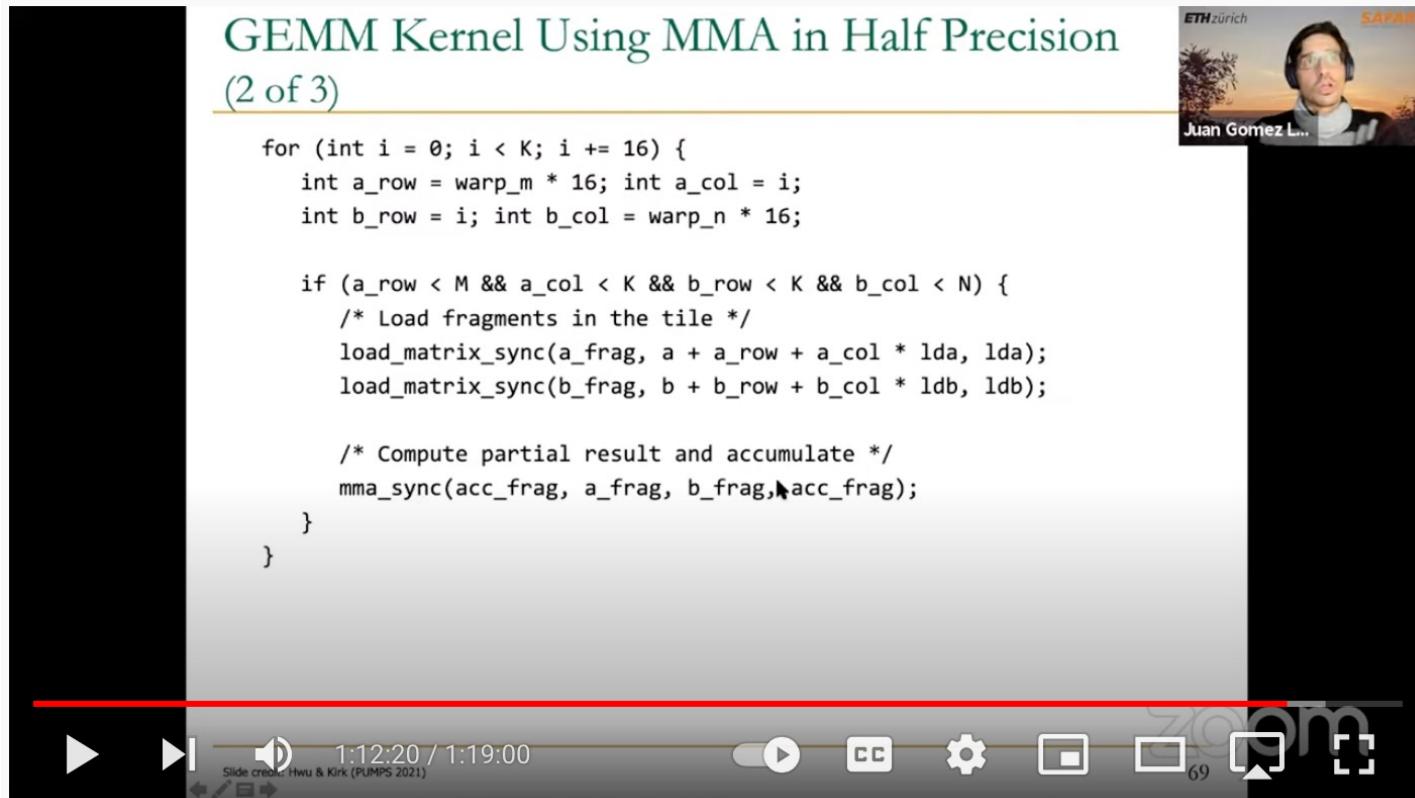
GEMM using Tensor Core Units (TCUs)

GEMM Kernel Using MMA in Half Precision
(2 of 3)

```
for (int i = 0; i < K; i += 16) {
    int a_row = warp_m * 16; int a_col = i;
    int b_row = i; int b_col = warp_n * 16;

    if (a_row < M && a_col < K && b_row < K && b_col < N) {
        /* Load fragments in the tile */
        load_matrix_sync(a_frag, a + a_row + a_col * lda, lda);
        load_matrix_sync(b_frag, b + b_row + b_col * ldb, ldb);

        /* Compute partial result and accumulate */
        mma_sync(acc_frag, a_frag, b_frag, acc_frag);
    }
}
```



Heterogeneous Systems Course: Meeting 8: Parallel Patterns: Convolution (Fall 2021)

262 views • Streamed live on Nov 25, 2021

 13  DISLIKE  SHARE  SAVE ...



Onur Mutlu Lectures

20.6K subscribers

SUBSCRIBED



Sparse Matrix Representation: COO

y\x	0	1	2	3
0	0	10	20	0
1	15	0	0	0
2	0	0	0	0
3	40	50	0	0

Row indices	0	0	1	3	3
Column indices	1	2	0	0	1
Values	10	20	15	40	50

In this representation for every non-zero element we store its **coordinates** and its **value**

SMASH: Software Compression Scheme (II)

BITMAP:

Encodes if a block of the matrix contains any non-zero element

MATRIX

NZ			
	NZ		
		NZ	NZ

BITMAP

1			
	1		
		1	1

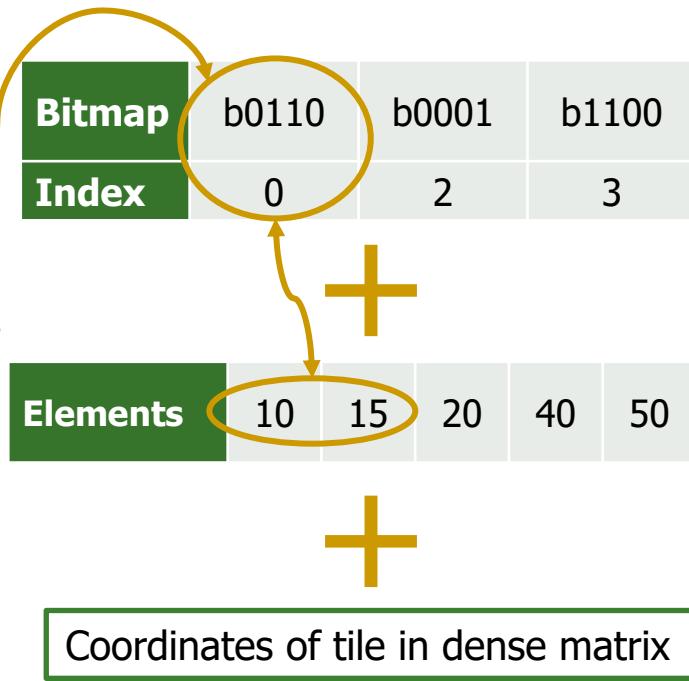


Might contain high number of zero bits

Idea: Apply the same encoding recursively to compress more effectively

Sparse Matrix Representation: Bitmap Format

y\x	0	1	2	3
0	0	10	20	0
1	15	0	0	0
2	0	2	0	0
3	40	50	0	0



Tiles

- A dense matrix is split into tiles

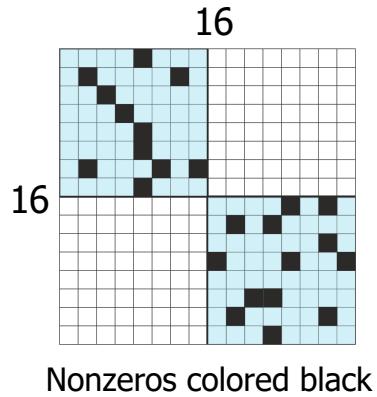
Bitmap

- A binary number for each tile. Each bit of is set to "1" iff there is an element in the corresponding position

Index

- An offset in the element array. It points to the starting position of the elements of each tile

How to Load Sparse Bitmaps into TCUs?



TCU layout

- We use 8x8 tiles
- Each warp multiplies 2 8x8 tiles
- Tensor cores are efficient even if not fully utilized¹

¹ Dakkak et al., "Accelerating Reduction and Scan Using Tensor Core Units," ICS 2019

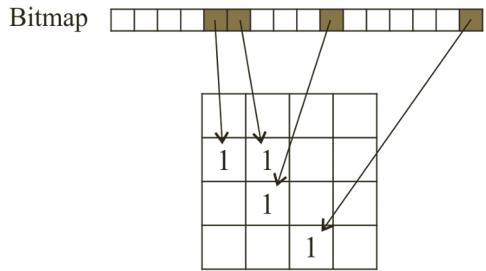
Allocating Memory for the Output

Elements

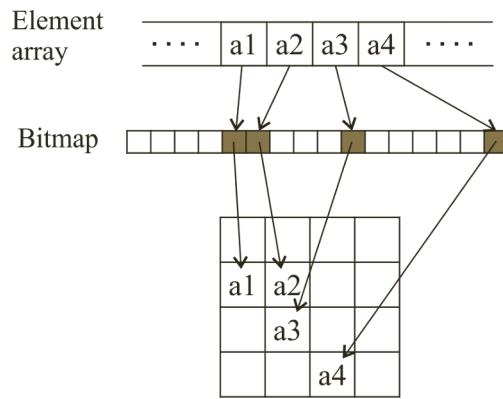


How many elements?

Counting kernel



Multiplication kernel



Memory pre-allocation

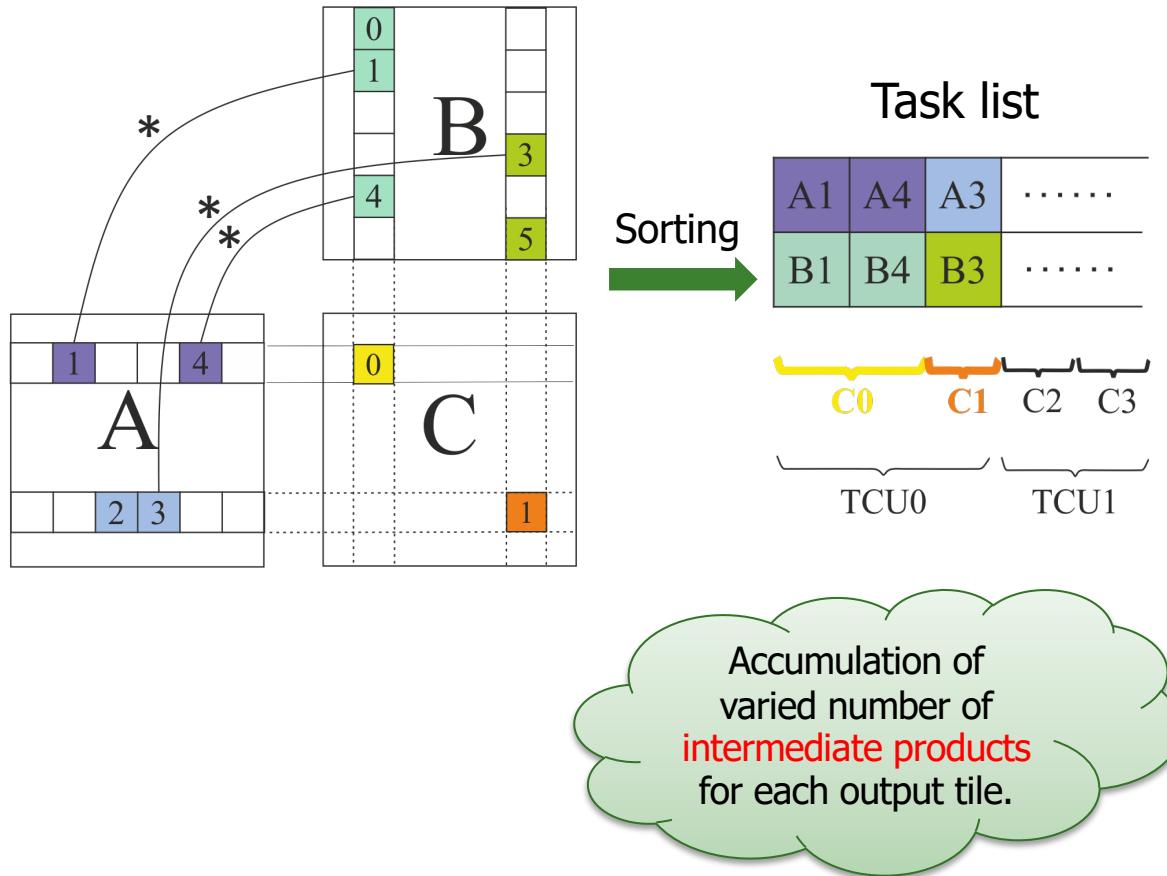
- Before executing the matrix multiplication we need to allocate memory
- We know how much memory to allocate only after the multiplication
- Various solutions: Upper bound, probabilistic, precise, progressive

Precise method

- We estimate the count of elements in the output with a partial multiplication
- We do not load any actual values, instead we use only the bitmaps
- Similar code base, TCUs, etc.

Finding Tiles to Multiply

Finding which tiles to multiply

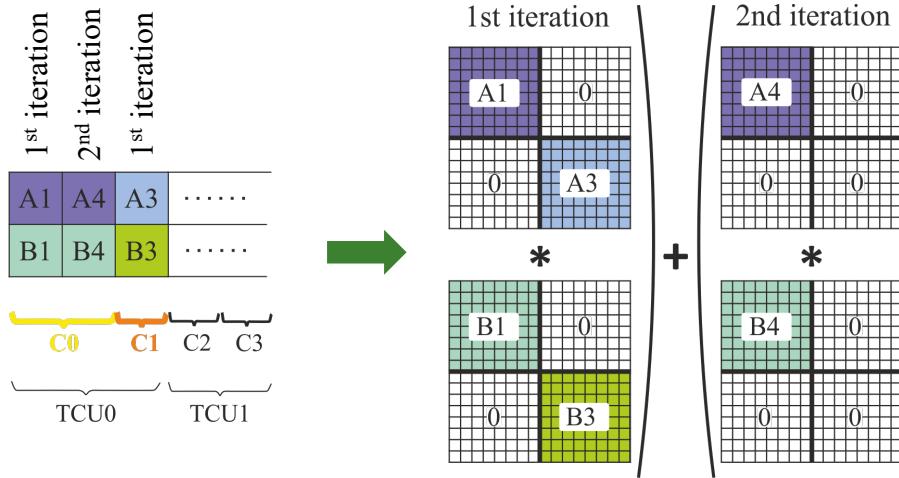


We perform the matrix multiplication as several tile multiplications of the form $C = A \cdot B + C$

Finding matrix multiplications

1. We find which tiles of A need to be multiplied with which tiles of B for each tile of C
2. We group intermediate products using segmented sort algorithms
3. We create a task list of tiles to multiply

Multiplying with TCUs (I)



Multiplying tiles

- The task list feeds the TCUs
- We calculate 2 tiles of the output with each TCU
- We iterate until all intermediate products of both the inner products are accumulated

Multiplying with TCUs (II)

$$\begin{array}{c} \text{1st iteration} \\ \left(\begin{array}{|c|c|} \hline A1 & 0 \\ \hline 0 & A3 \\ \hline * & \\ \hline B1 & 0 \\ \hline 0 & B3 \\ \hline \end{array} \right) + \left(\begin{array}{|c|c|} \hline A4 & 0 \\ \hline 0 & 0 \\ \hline * & \\ \hline B4 & 0 \\ \hline 0 & 0 \\ \hline \end{array} \right) \end{array}$$

$$\rightarrow \begin{array}{|c|c|} \hline C0 & 0 \\ \hline 0 & C1 \\ \hline \end{array}$$

Warp-wide ballot



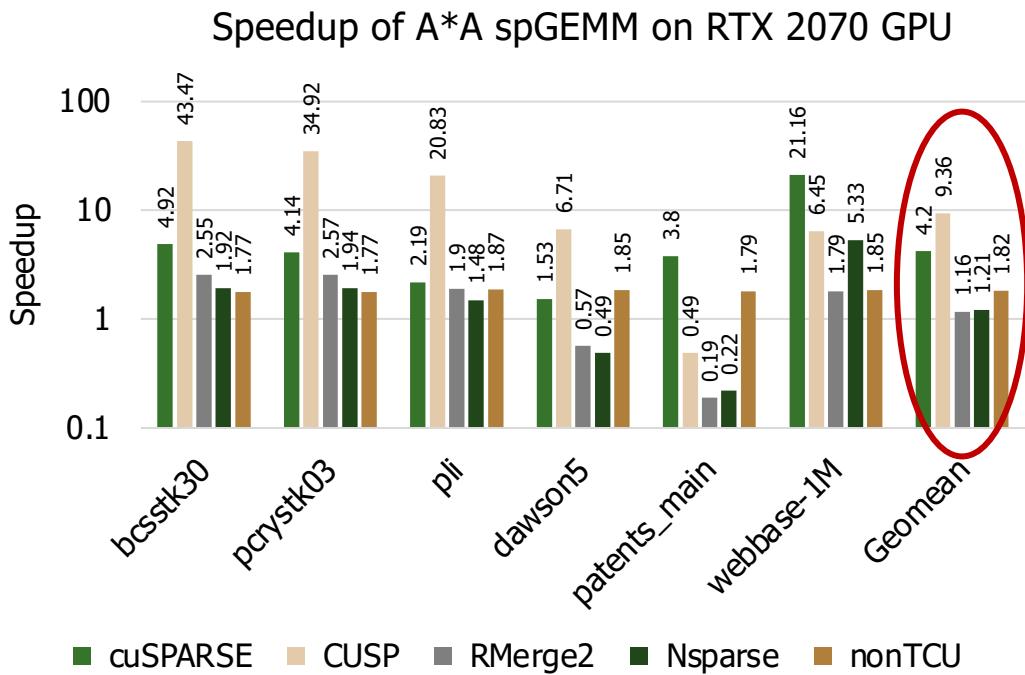
Image: Getty images



Bitmap C0	b0110	b0001	b1100
Bitmap C1	b0010	b1001	

Elements C0	10	15	20	40	50
Elements C1	5	13	7		

Speedup of the Proposed Method (tSparse)



Observations

- On average 4.2x, 9.36x, 1.16x, 1.21x faster than cuSPARSE, CUSP, RMerge2, Nsparse, respectively
- TCUs are effective (1.82x faster than nonTCU)
- tSparse works better with denser tiles (density > 6)

tSparse for SpGEMM

Accelerating sparse matrix–matrix multiplication with GPU Tensor Cores[☆]

Orestis Zachariadis ^{a,*}, Nitin Satpute ^a, Juan Gómez-Luna ^b, Joaquín Olivares ^a

^a Department of Electronic and Computer Engineering, Universidad de Córdoba, Córdoba, Spain

^b Department of Computer Science, ETH Zurich, Zurich, Switzerland

Zachariadis et al., "Accelerating Sparse Matrix–Matrix Multiplication with GPU Tensor Cores," Computers and Electrical Engineering, 2020 [arXiv version: <http://arxiv.org/abs/2009.14600>]

The screenshot shows the GitHub repository page for `oresths/tSparse`. The repository is public and has 1 branch and 0 tags. The commit history shows five commits from `15bf5c4` on Oct 1, 2020, to a better README. The README.md file contains the following text:

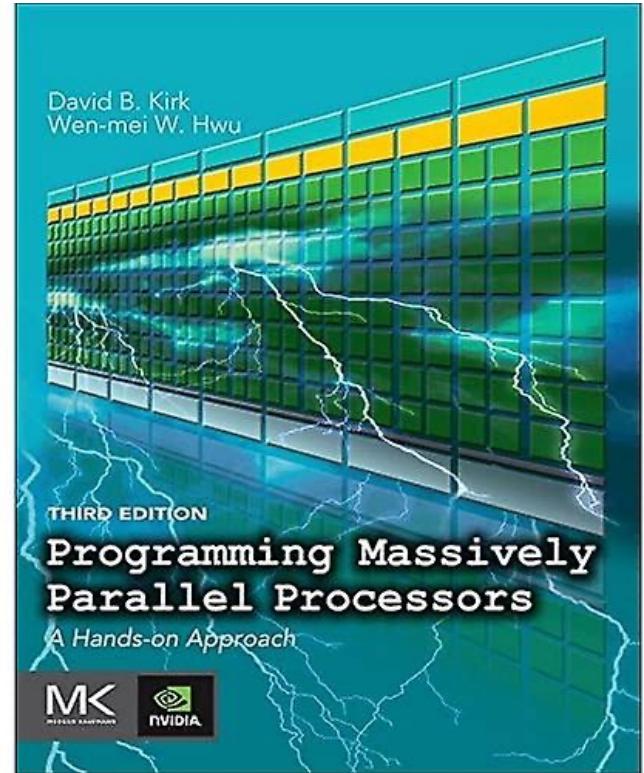
```
Accelerating Sparse Matrix-Matrix Multiplication with GPU Tensor Cores

In this repository we provide the source code of our accelerated Sparse Matrix-Matrix multiplication (SpGEMM) implementation, which we describe in "Accelerating Sparse Matrix-Matrix Multiplication with GPU Tensor Cores" [1].
```

<https://github.com/oresths/tSparse>

Recommended Readings

- Hwu and Kirk, “[Programming Massively Parallel Processors](#),” Third Edition, 2017
 - Chapter 10 - Parallel patterns:
sparse matrix computation:
An introduction to data compression
and regularization



P&S Heterogeneous Systems

Parallel Patterns: Sparse Matrices

Dr. Juan Gómez Luna

Prof. Onur Mutlu

ETH Zürich

Fall 2021

9 December 2021