6.5930/1
Hardware Architectures for Deep Learning

Sparse Architectures

March 2, 2026

Joel Emer and Vivienne Sze

Massachusetts Institute of Technology
Electrical Engineering & Computer Science

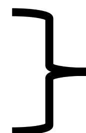
Sources of Sparsity

- **(Input) Activation Sparsity**
  - Sparsity due to ReLU
  - Correlation in input data
  - Structure of input representation (e.g., Graphs)

- **Weight Sparsity**
  - Weight reordering and reuse
  - Network pruning

Exploiting Sparsity

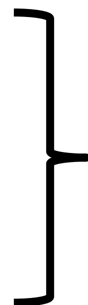
Sparse data can be compressed



Can save space and energy by avoiding **storage and movement** of zero values

$$\boldsymbol{anything} \times \mathbf{0} = \mathbf{0}$$

$$\boldsymbol{anything} + \mathbf{0} = \boldsymbol{anything}$$



Can save time and energy by avoiding fetching unnecessary operands and avoiding **ineffectual** computations

Impact of Ineffectual Operations

- Total operations = effectual operations + ineffectual operations

  (total operations referred to as ***"algorithmic computes"*** *in Lab 4)*

- Total operations ***performed*** = effectual operations + unexploited ineffectual operations

  (total operations referred to as ***"actual computes"*** *in Lab 4)*

Unexploited Ineffectual Operations

- Ideally, hardware would exploit and thus avoid **all** ineffectual operations
  - Total operations performed = effectual operations ~~+ unexploited ineffectual operations~~
- In practice, trying to avoid **all** ineffectual operations is challenging and can increase hardware overhead
  - Tradeoff between reducing number of operations and cost per operation
- In this lecture, we will discuss the various hardware approaches to avoid ineffectual operations

Irregular Processing

- Exploiting sparsity makes processing irregular!
- The number of non-zeros within and across a tensor can vary, causing variation in the number of cycles and the amount of required storage
- The location of non-zeros may also be unknown
- Introduces challenges such as
  - underutilized buffers and PEs, workload imbalance, random data access

Sparse Acceleration Features (SAF)

Gating:
Explicitly eliminate ineffectual storage accesses and computes by letting the hardware unit staying idle for the cycle to save energy

Skipping:
Explicitly eliminate ineffectual storage accesses and computes by skipping the cycle to save energy and time

Format:
Choose tensor representations to save storage space and energy associated with zero accesses

Intersection for Gating + Skipping

Sparsity Example with 1D Dot Product

total operations = 6

effectual operations = 2

ineffectual operations = 4

To determine whether operation is effectual or ineffectual, need to look at the **intersection** of the two vectors.

Key Attributes and Design Choices

- Finding Intersection
  - Whether to check one or both inputs (impacts reads & total operations performed)
    - **Single sided versus Dual sided**
  - If check only one input (single sided), which one:
    - Leader → Follower
      - Check if “Leader” is non-zero, and decide whether to access “Follower” based on “Leader”

- Exploiting Ineffectual Computes
  - Whether to save cycles
    - **Gating versus Skipping**

Example Sparse Accelerators

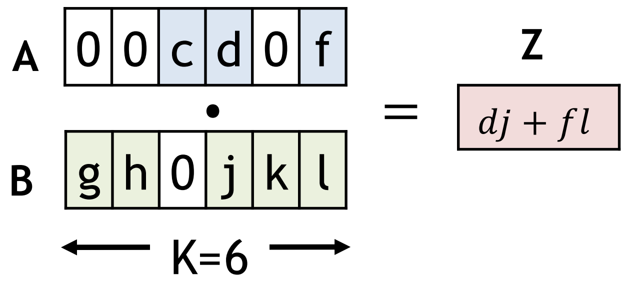
Sparse Accelerator	Gating/Skipping
Eyeriss [JSSC 2017]	Single-sided Gating: Gate W $\leftarrow$ I, Gate O $\leftarrow$ I
Eyeriss v2 [JETCAS 2019]	Single-sided Skipping: Skip W $\leftarrow$ I, Skip O $\leftarrow$ I &W
SCNN [ISCA 2017]	Single-sided Skipping: Skip W $\leftarrow$ I, Skip O $\leftarrow$ I &W
ExTensor [MICRO 2019]	Dual-sided Skipping: Skip A $\leftrightarrow$ B, Skip Z $\leftarrow$ A &B
DSTC [ISCA 2021]	Dual-sided Skipping: Skip A $\leftrightarrow$ B, Skip Z $\leftarrow$ A &B

Classification in https://sparseloop.mit.edu/documents/2022-micro-sparseloop.pdf

Impact of Different Approaches

cycles →

SAFs	*1*	*2*	*3*	*4*	*5*	*6*	*Total*
None	read(0) read(g) compute(0,g)	read(0) read(h) compute(0,h)	read(c) read(0) compute(c,0)	read(d) read(j) compute(d,j)	read(0) read(k) compute(0,k)	read(f) read(l) compute(f,l)	6 reads 6 reads 6 computes
Gating on B storage accesses based on A *(Gate B ← A)*	read(0) ~~read(g)~~ ~~compute(0,g)~~	read(0) ~~read(h)~~ ~~compute(0,h)~~	read(c) read(0) compute(c,0)	read(d) read(j) compute(d,j)	read(0) ~~read(k)~~ ~~compute(0,k)~~	read(f) read(l) compute(f,l)	6 reads 3 reads 3 computes
Skipping on B storage accesses based on A *(Skip B ← A)*	read(c) read(0) compute(c,0)	read(d) read(j) compute(d,j)	read(f) read(l) compute(f,l)				3 reads 3 reads 3 computes
Skipping on A and B storage accesses based on A & B *(B ↔ A)*	read(d) read(j) compute(d,j)	read(f) read(l) compute(f,l)					2 reads 2 reads 2 computes



Approaches exploiting ineffectual computation vary in terms of number of cycles, reads, and computes

- Gating saves reads and compute, but not cycles
- Skipping saves reads, compute and cycles

Terminology

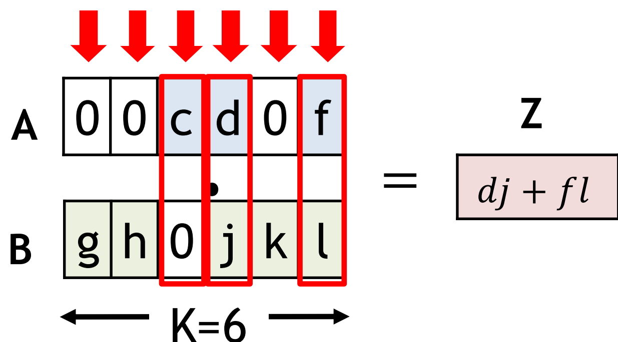
- **Coordinate versus Position**
  - e.g., value at coordinate 2 of vector A is c, while value at position 2 of vector A is f

coordinate	0	1	2	3	4	5
position			0	1		2
A	0	0	c	d	0	f

coordinate	0	1	2	3	4	5
position	0	1		2	3	4
B	g	h	0	j	k	l

Finding Intersection

Gating, Single-sided (Gate B based on A)



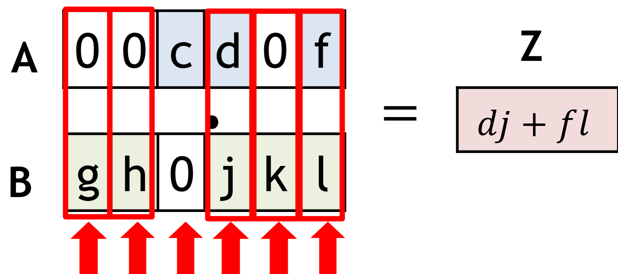
total operations = 6
effectual operations = 2
ineffectual operations = 4

total cycles = 6
total A reads = 6
total B reads = 3
total operations performed = 3

Memory access pattern:
A sequential (increment by one)
B depends on A

Finding Intersection

Gating, Single-sided (Gate A based on B)



total operations = 6
effectual operations = 2
ineffectual operations = 4

total cycles = 6
total A reads = 5
total B reads = 6
total operations performed = 5

Memory access pattern:
A depends on B
B sequential (increment by one)

Eyeriss – Gating

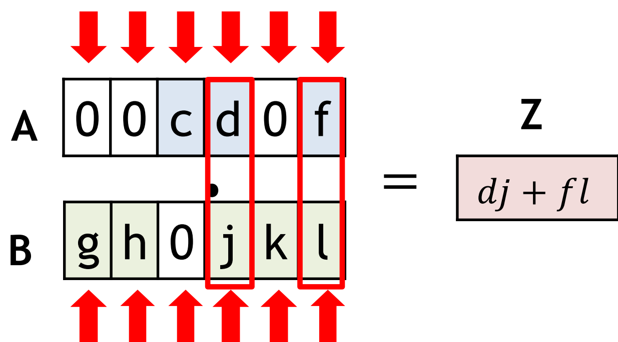
Gate **multiply** and **memory reads** when input data is zero.
Reduce **PE power** by **45%**

Input Activations
Image Scratch Pad (12x16b REG)
== 0
Zero Buffer
Enable
Filter Weights
Filter Scratch Pad (225x16b SRAM)
Input Psum
2-stage pipelined multiplier
Accumulate Input Psum
0
1
Output Psum
Partial Sum Scratch Pad (24x16b REG)
0
1
0

[**Chen**, *ISSCC* 2016]

Finding Intersection

Gating, Dual Sided



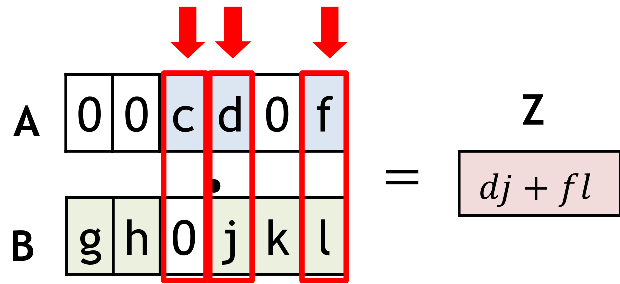
total operations = 6
effectual operations = 2
ineffectual operations = 4

total cycles = 6
total A reads = 6
total B reads = 6
total operations performed = 2

Memory access pattern:
A sequential (increment by one)
B sequential (increment by one)

Finding Intersection

Skipping, Single-sided (Skip B based on A)



Go to **next** non-zero

Memory access pattern:
A sequential (jump to next non-zero)
B depends on A

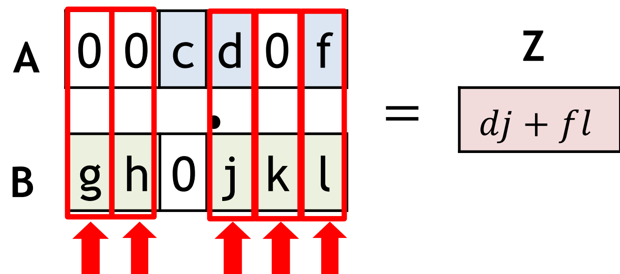
total operations = 6
effectual operations = 2
ineffectual operations = 4

total cycles = 3
total A reads* = 3
total B reads* = 3
total operations performed = 3

* Reads don't include metadata

Finding Intersection

Skipping, Single-sided (Skip A based on B)



total operations = 6
effectual operations = 2
ineffectual operations = 4

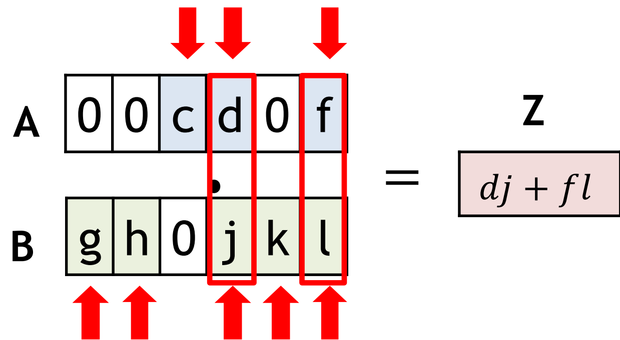
total cycles = 5
total A reads* = 5
total B reads* = 5
total operations performed = 5

Memory access pattern:
A depends on B
B sequential (jump to next non-zero)

* Reads don't include metadata

Finding Intersection

Skipping, Dual Sided



total operations = 6
effectual operations = 2
ineffectual operations = 4

total cycles = ? (min 2)
total A reads* = 2
total B reads* = 2
total operations performed = 2

Memory access pattern:
A sequential (jump to next non-zero) & depends on B
B sequential (jump to next non-zero) & depends on A

* Reads don't include metadata

Summary: Memory Access Patterns

<table>
<tr><th></th><th>Gating</th><th>Skipping</th></tr>
<tr><td>Single-sided</td><td>1. Determine access coordinate: Increment by one (access every coordinate in leader)
2. Access data at coordinate: If non-zero, read value(s) at coordinate and perform compute
3. Go to 1 (until end of either inputs)</td><td>1. Determine access coordinate: Find the coordinate of next non-zero in leader
2. Access data at coordinate: Read value(s) at the coordinate and perform compute
3. Go to 1 (unless reach end of either inputs)</td></tr>
<tr><td>Dual-sided</td><td>1. Determine access coordinate: Increment by one (access every coordinate in both inputs)
2. Access data at coordinate: If both non-zero, read value(s) at coordinate and perform compute
3. Go to 1 (until end of either inputs)</td><td>1. Determine access coordinate: Find the coordinate of next pair of non-zero inputs
• iterate between inputs starting with the one at the earlier coordinate, until find non-zero coordinate that are the same (number of iterations varies – design choice: set max iterations, if exceed then idle cycle)
2. Access data at coordinate: Read value(s) at the coordinate and perform compute
3. Go to 1 (until end of either inputs)</td></tr>
</table>

* **next** is defined by traversal order – we assume for now same as storage format [layout] order (concordant traversal)

Gating versus Skipping

- **Gating**
  - As cycles are not reduced, can spend cycles to check whether data is non-zero (i.e., can find non-zeros in real time)

- **Skipping**
  - To reduce cycles, want to **only** spend cycles on non-zero value
  - Would like to jump to **next** non-zero value without wasting cycles
    - Needs "preprocessing” to indicate coordinate of non-zeros
    - Preprocessing can be done as part of **representation format!**

Representation Formats

Sparse Acceleration Features (SAF)

Gating:
Explicitly eliminate ineffectual storage accesses and computes by letting the hardware unit staying idle for the cycle to save energy

Skipping:
Explicitly eliminate ineffectual storage accesses and computes by skipping the cycle to save energy and time

Format:
Choose tensor representations to save storage space and energy associated with zero accesses

Exploiting Sparsity to Reduce Data Movement

- Key idea is to **compress** data
  - Reduce storage required for a given tensor or tile
  - Reduce data movement required to access a given tensor or tile

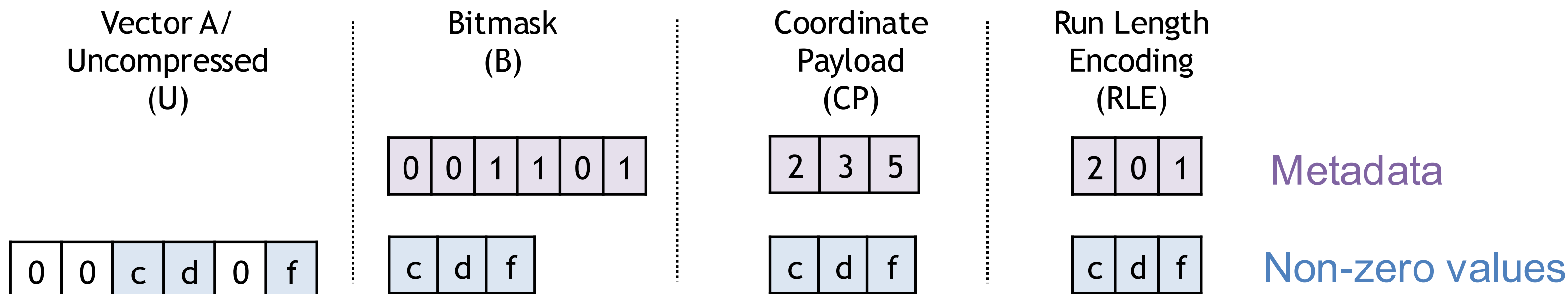
- Impact on memory hierarchy and data movement
  - Reduce the size of memory (reduce energy cost per memory access), **or**
  - For same memory size, larger tile → increase data reuse → reduce data movement from next level of hierarchy
    - e.g., store larger tile in global buffer and reduce data movement from DRAM

Evaluating Representation Formats

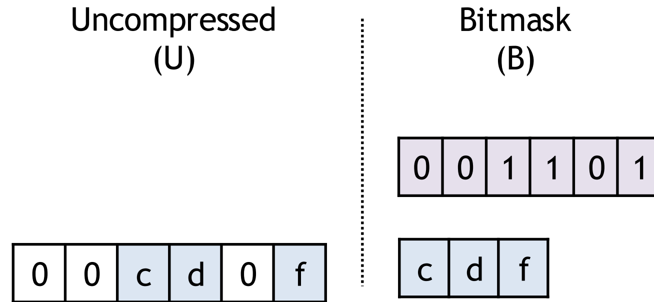
- **Compression Efficiency:** How well it compresses data
  - Measure ratio of representation format size vs uncompressed representation size
- **Access Efficiency:** How easy it is to access non-zero values (or the coordinate of the non-zeros) → required for skipping operation
  - Measure complexity for finding **next** non-zero value and checking whether there is a non-zero value at a **given** corresponding coordinate

Representation Formats

- Many different representation formats (four examples given below)
- For compression formats, would like to only send the non-zero values, but also need to indicate coordinate of non-zero value
- Formats differ in representation of non-zero coordinates (in the **metadata**)
  - How many bits spent on metadata impacts **compression efficiency**

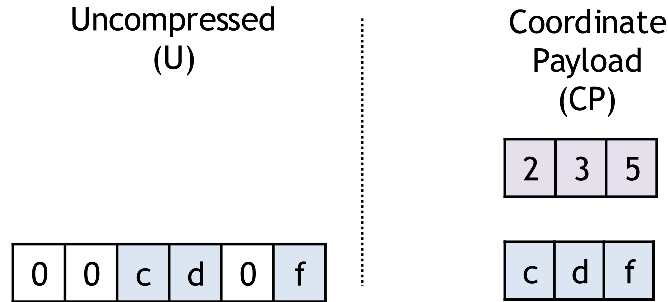


Bitmask



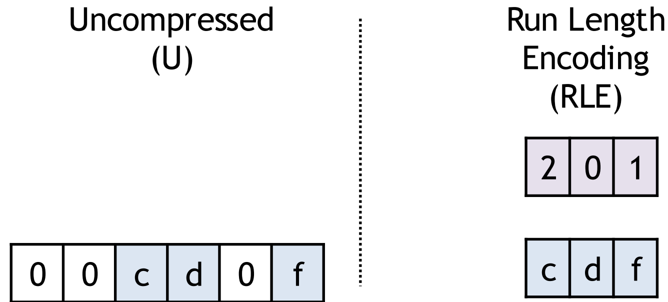
- Metadata
  - Use 1 bit per coordinate to indicate if zero (0) or non-zero (1)
  - Overhead: 1 bit per coordinate in uncompressed representation
- In above example, if each value was 8-bits
  - Uncompressed is 6 x 8-bits = 48-bits
  - Bitmask is **6 x 1-bit** + 3 x 8-bits = 30-bits

Coordinate Payload



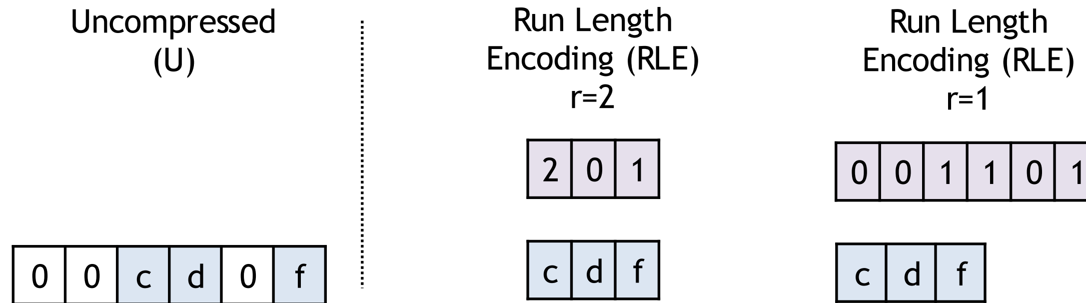
- Metadata
  - The coordinate for each coordinate of non-zero value
  - Overhead: n-bits per non-zero value, where n is the number of bits per coordinate depends and on $\log_2$ of max coordinate value (i.e., size of vector)
- In above example, if each value was 8-bits and 3-bits for coordinate
  - Uncompressed is 6 x 8-bits = 48-bits
  - Coordinate Payload is **3 x 3-bit** + 3 x 8-bits = 33-bits

Run Length Encoding



- Metadata
  - The number of zeros ('run length') between each non-zero
  - Overhead: r-bits used to encode the run length (max run length is 2^r-1)
- In above example, if each value was 8-bits and 3-bits for run length
  - Uncompressed is 6 x 8-bits = 48-bits
  - Run Length Encoding is **3 x 3-bit** + 3 x 8-bits = 33-bits

Run Length Encoding (Choice of r)



- The number of r-bits (max run length 2^r -1) is a design choice (default is vector size)
- In above example, if each value was 8-bits and 2-bits for run length (max run 3)
  - Run Length Encoding is **3 x 2-bit** + 3 x 8-bits = 30-bits
- In above example, if each value was 8-bits and 1-bits for run length (max run 1)
  - Run Length Encoding is **6 x 1-bit** + 3 x 8-bits = 30-bits
  - Need to send multiple run length values when run larger than max run
  - Note: for r=1, run length encoding becomes same as bitmask
- Choose r based on distributions of run length if known a priori

I/O Compression in Eyeriss

Link Clock | Core Clock

DNN Accelerator

14×12 PE Array

Filter

Input Image

Output Image

Decomp

Comp

Run-Length Encoding (RLE)

Example:

Input: 0, 0, 12, 0, 0, 0, 0, 53, 0, 0, 22, …

Output (64b):

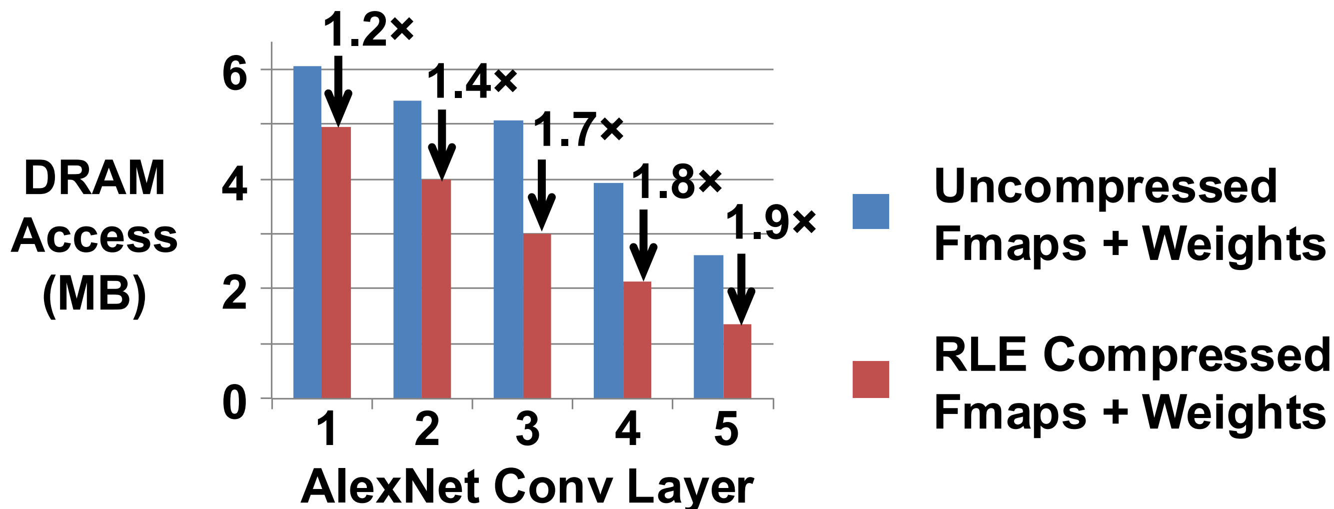
Run	Level	Run	Level	Run	Level	Term
2	12	4	53	2	22	0
5b	16b	5b	16b	5b	16b	1b

64 bits

Off-Chip DRAM

[Chen, *ISSCC* 2016]

Compression Reduces DRAM BW



Simple RLE within 5% - 10% of theoretical entropy limit

[**Chen**, *ISSCC* 2016]

From information theory,

Entropy $H = -\sum_{i=0}^{L-1} p_i \cdot \log_2 p_i$

↑

minimum average number of bits required to code f

Comparison of Compression Efficiency of Formats

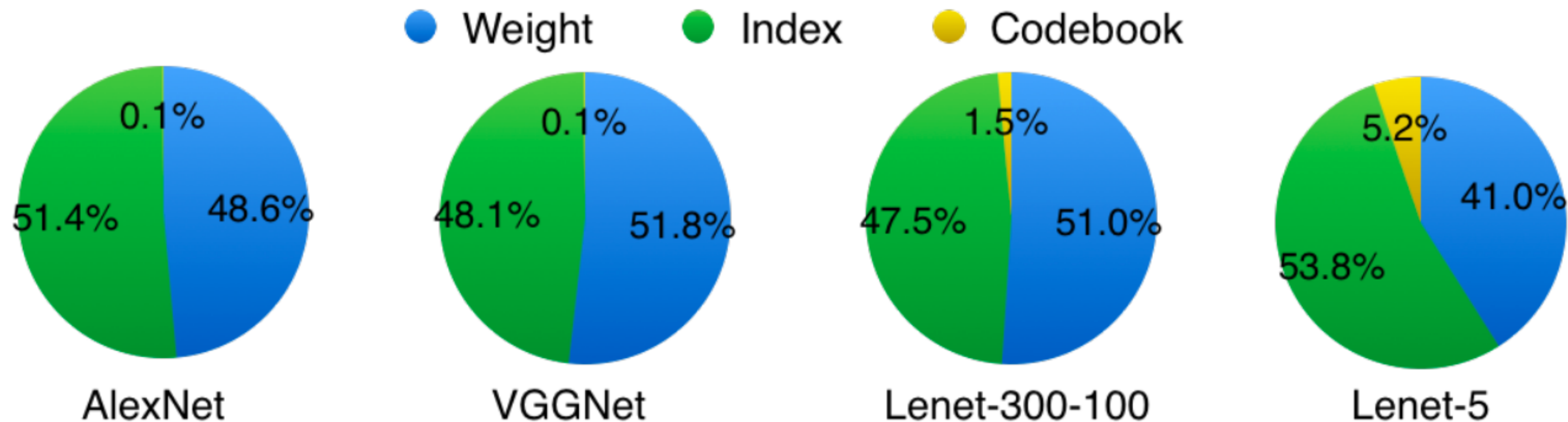
In this example, 8-bits per value and vector size K=16

# of non-zeros (density)	**1 (6.25%)**	**8 (50%)**	**16 (100%)**
Uncompressed	16 x 8b = **128b**	16 x 8b = **128b**	16 x 8b = **128b**
Bitmask	16 x 1b + 1 x 8b = **24b**	16 x 1b + 8 x 8b = **80b**	16 x 1b + 16 x 8b = **144b**
Coordinate Payload (n=4b per coordinate)	1 x 4b + 1 x 8b = **12b**	8 x 4b + 8 x 8b = **96b**	16 x 4b + 16x 8b = **192b**
Run Length Encoding (r=4b for run)	1 x 4b + 1 x 8b = **12b**	8 x 4b + 8 x 8b = **96b**	16 x 4b + 16 x 8b = **192b**

Compression efficiency varies for different formats for different densities

Metadata Overhead

Metadata to indicate non-zero coordinate (i.e., Index) accounts for approximately **half** of storage for unstructured sparsity



[**Han**, *ICLR* 2016]

Compression Efficiency of Formats

- Uncompressed
  - works best for dense data since no metadata overhead
- Bitmask
  - 1 bit overhead per coordinate
  - works best for moderate sparsity
  - max compression is 1/# of bits per non-zero value
- Coordinate Payload
  - n-bit overhead per non-zero
  - works best for high sparsity
- Run Length Encoding
  - r-bit overhead per non-zero
  - works best for high sparsity and long runs of zeros (design choice of max run length)

Skipping (Single Sided)

Cycle	**Read A (coordinate)**	**Read A (value)**	**Non-zero (?)**	**Read B (coordinate)**	**Read B (value)**	**Compute**
1	**2**	c	Yes	2 (same as A)	0	c*0
2	**3**	d	Yes	3 (same as A)	j	d*j
3	**5**	f	Yes	5 (same as A)	l	f*l

How to determine coordinate of non-zeros in A (2, 3, 5) from metadata?

A: | 0 | 0 | c | d | 0 | f |

•

B: | g | h | 0 | j | k | l |

K=6

= Z: $dj + fl$

Uncompressed: Coordinate of Next Non-zero

Uncompressed
(U)

0	0	c	d	0	f

Need to check each coordinate in vector and perform comparison with 0

A (coordinate)	**Read A (value 8-bits)**	**Non-zero (?)**
0	0	No
1	0	No
2	c	Yes
3	d	Yes
4	0	No
5	f	Yes

Bitmask: Coordinate of Next Non-Zero

Need to check each coordinate in metadata (equal to size of vector)

A (coordinate)	**Read A (metadata 1-bit)**	**Read A (value)**
0	0	-
1	0	-
2	1	c
3	1	d
4	0	-
5	1	f

Run Length Encoding: Coordinate of Next Non-Zero

Run Length Encoding (RLE)

2	0	1

c	d	f

Read A (metadata r-bit)	Compute coordinate	Read A (value)
2	2	c
0	2+1+0=3	d
1	3+1+1=5	f

- Determine coordinate by accumulating values in metadata
  - 2
  - 2+1+0 = 3
  - 3+1+1 = 5
- Number of memory access = Number of run length values
  - Number of run length values can be **more** than number of non zeros if there are runs of zeros that exceed max run length set by r
- Note: Accumulation and comparison have similar cost, but accumulation needs to store prior value

Coordinate Payload: Coordinate of Next Non-Zero

Coordinate
Payload
(CP)

2	3	5

c	d	f

- No compute needed, just access next coordinate directly in metadata
- Number of memory access = Number of non-zeros values

Read A (metadata n-bit)	**Coordinate**	**Read A (value)**
2	2	c
3	3	d
5	5	f

Summary: Compute for Finding Coordinate of Next Non-Zero

Uncompressed	Read value and compare at each coordinate until find next non-zero (multiple value reads and comparisons)
Bitmask	Read metadata at each coordinate until find next non-zero (multiple 1 bit reads)
Run Length Encoding	Read metadata (run length) and accumulate to get next non-zero coordinate (metadata read and accumulation)
Coordinate Payload	Read metadata to get next non-zero coordinate (metadata read)

Evaluating Representation Formats

- **Compression Efficiency:** How well it compresses data
  - Measure ratio of representation format size vs uncompressed representation size
  - Depends on how well the compression format ***matches*** distribution of data and the metadata overhead

- **Access Efficiency:** How easy it is to access non-zero values (or the coordinate of the non-zeros) → required for skipping operation
  - Measure complexity for finding **next** non-zero value or checking whether there is a non-zero value at a **given** corresponding coordinate
  - Depends on whether need to do any processing on the metadata to determine coordinate and whether need to uncompress data (discordant traversal when "next" does not match compression order)

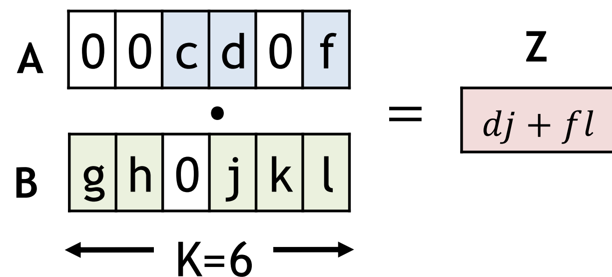
Access Efficiency of Formats (Find Coordinate of Next Non-zero)

- Uncompressed
  - Perform comparison for every coordinate (expensive)
  - Scales with size of vector
- Bitmask
  - Read 1-bit value of bit mask for every coordinate
  - Scales with size of vector
- Run Length Encoding
  - Accumulate run length to compute next coordinate
  - Scales with number of runs (number of reads) and length of runs (cost per read)
- Coordinate Payload
  - Just go to next coordinate (no compute)
  - Scales with number of non-zeros (number of reads) and size of vector (cost per read)

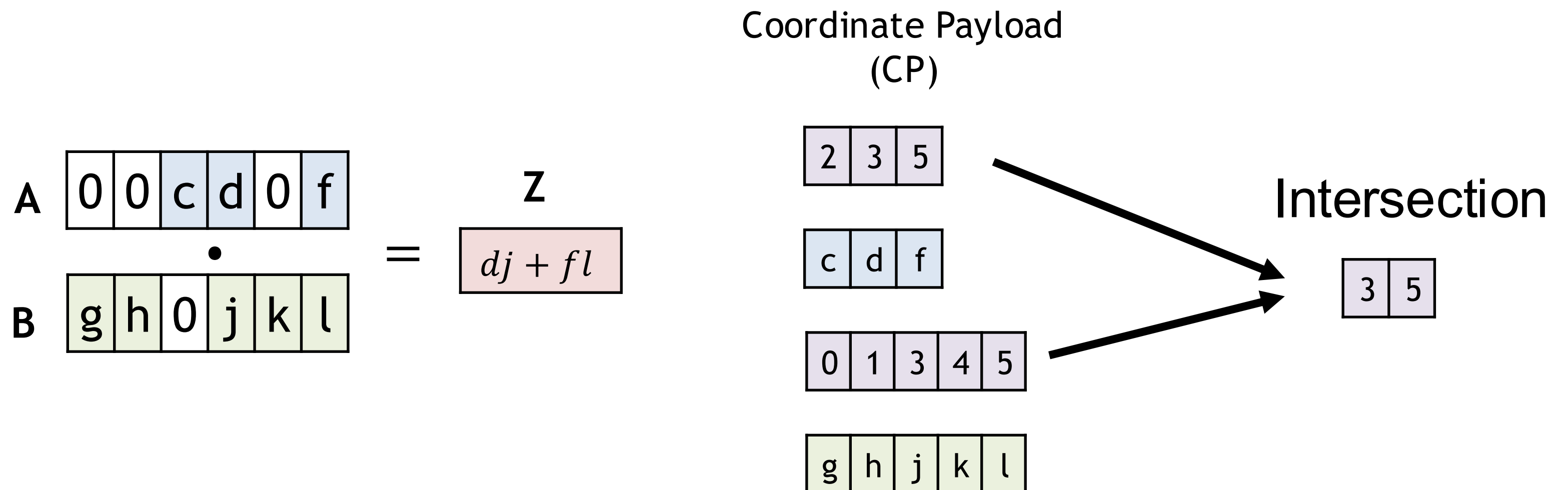
Skipping (Dual Sided)

Cycle	Read A (coordinate)	Read A (value)	Non-zero (?)	Read B (coordinate)	Read B (value)	Compute
1	3	d	Yes	3 (same as A)	j	d*j
2	5	f	Yes	5 (same as A)	l	f*l

How to determine coordinate of non-zeros pairs in A & B (3, 5) from metadata?



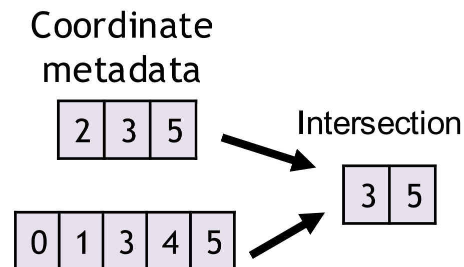
Need to find intersection between metadata



Read A (metadata n-bit)	Read B (metadata n-bit)	Do coordinates match?	Read A (value)	Read B (value)
2	3	No	-	-
3	**3**	Yes	d	j
5	5	Yes	f	l

Finding Intersection for Dual-Sided Skipping

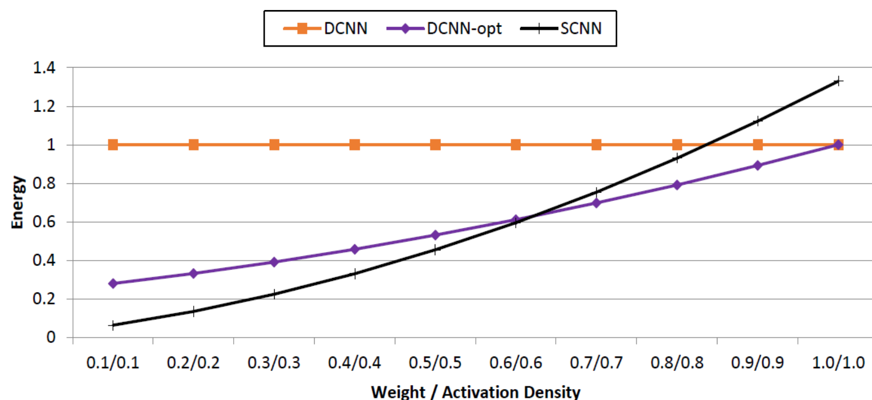
- Find the coordinate of next pair of non-zero input
  - iterate between inputs starting with the one at the **earlier** coordinate, until find non-zero coordinate that are the same; number of iterations varies
  - design choice: set max iterations, if exceed then idle cycle

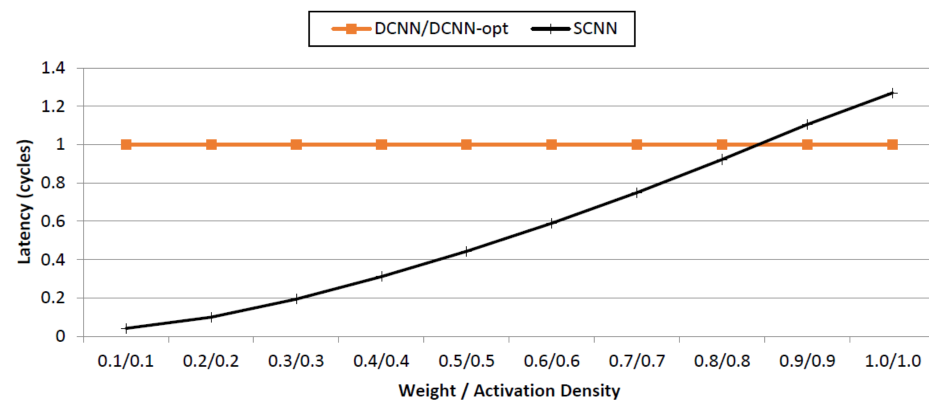


- Challenging to find next pair of non-zero inputs in one cycle

- ExTensor [MICRO 2019] uses binary search of remaining coordinates → effective if jump to next non-zero far away

Exploiting Sparsity Does Not Come for Free





- Additional overhead hardware required exploit sparsity
- Overall benefits also depends on degree sparsity (i.e., 1- density)
- Tradeoff between reducing number of operations and cost per operation

[**Parashar**, *ISCA* 2017]

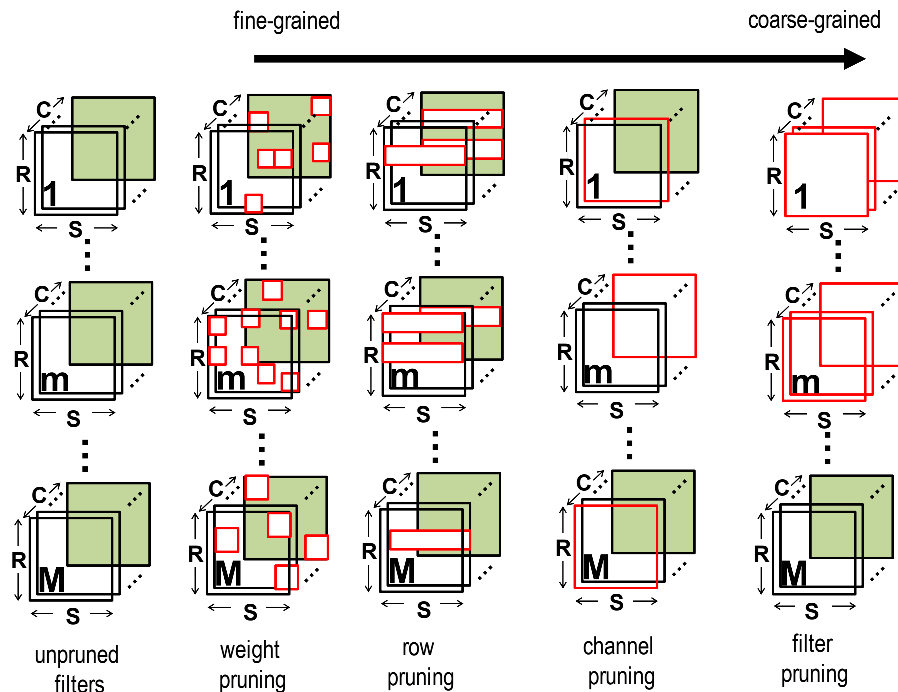
Structured Sparsity

Structure in Sparsity

- **Unstructured sparsity** allows non-zeros to occur in any coordinate
  - As a result, to find non-zeros need to search large range of coordinates (increase hardware complexity or latency/energy) and can increase the metadata overhead (reduce compression efficiency) since need to be able to signal all possible coordinates
  - However, more flexibility in terms of DNN model design (potentially higher accuracy)

- **Structured sparsity** reduces search space to find non-zeros
  - Requires easier to find non-zeros and requires less metadata overhead
  - However, more restrictions on the DNN model design (potential reduce accuracy)

Granularity of Sparsity

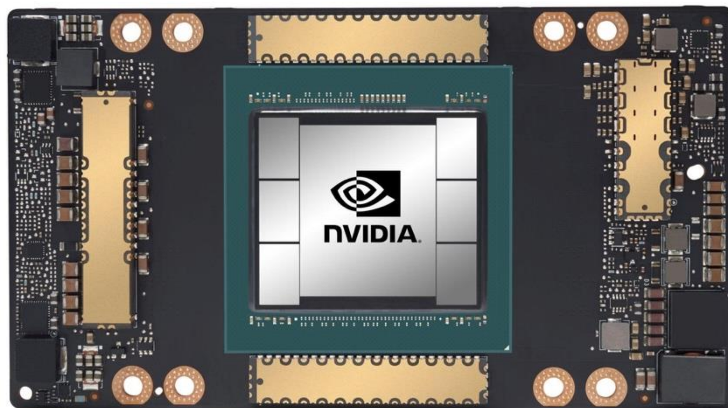


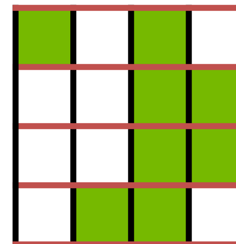
Benefits:

Increase coarseness → more structure in sparsity (easier for hardware)

Less signaling for location of zeros → better compression

Example of Structured Sparsity



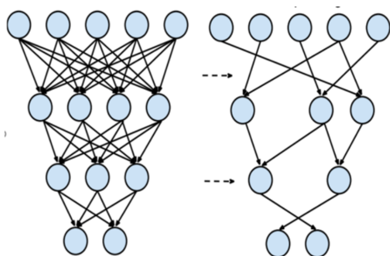


Per-row
2:4 structured sparse
(G:H pattern)

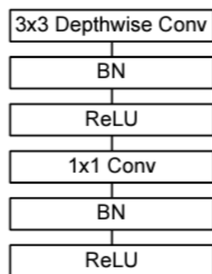
NVIDIA Sparse Tensor Core (STC)
[NVIDIA, TechReport20]

Different DNN Optimizations Introduce Different Sparsity

Optimizations to Reduce Model Size



Pruning Techniques
[Han, NeurIPS15]

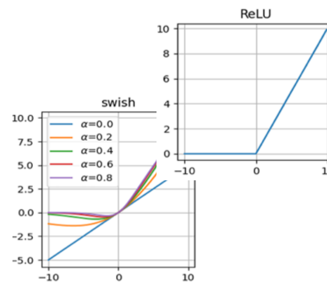


Depth-wise Separable Layers
[Howard, CVPR17]

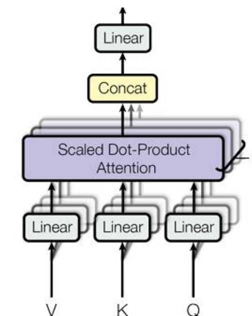
Introduces Sparse Weights

Introduces Dense Weights

Optimizations to Improve Accuracy



Activation Functions
[Apicella, NN21]



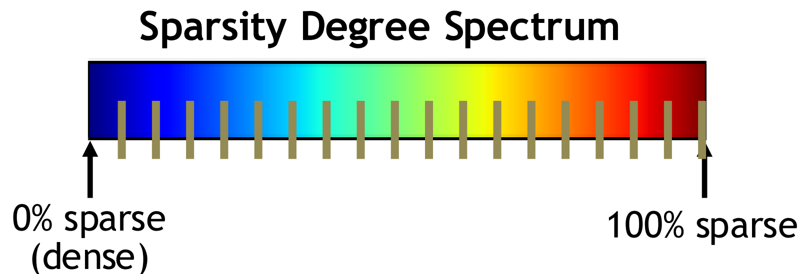
Attention-based Modules
[Vaswani, NeurIPS17]

Introduces Dense/Sparse Activations

Introduces Dense Act./Weights

Modern DNNs can weights and activations that are either **dense or sparse** with various sparsity degrees

Requirements for an Ideal Sparse DNN Accelerator



Flexible

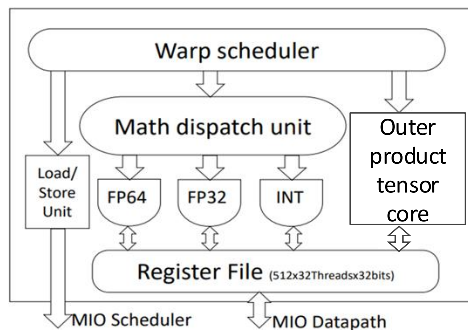
exploit a wide range of many sparsity degrees

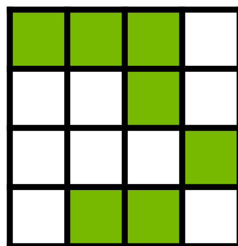
Efficient

low hardware overhead SAF implementations

Existing Works Do Not Meet Such Requirements

Unstructured Sparse Accelerators





Unstructured sparse

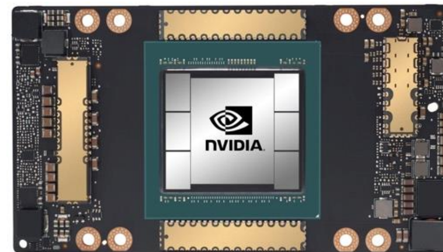
Dual-Side Sparse Tensor Core (DSTC)
[Wang, ISCA21]

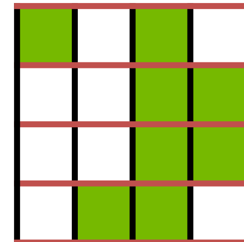
Sparsity Degree Spectrum

Continuously Translated into Savings

– Ineffcient
+ Flexible

Structured Sparse Accelerators





Per-row
2:4 structured sparse
(G:H pattern)

NVIDIA Sparse Tensor Core (STC)
[NVIDIA, TechReport20]

Sparsity Degree Spectrum

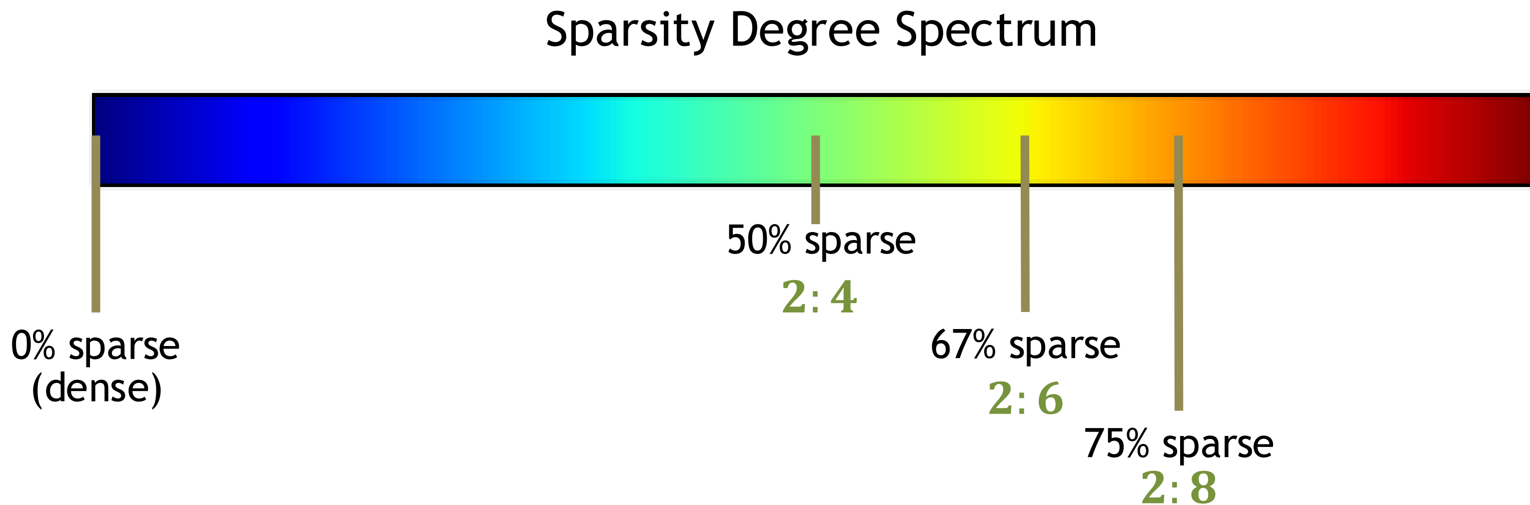
0% sparse (dense)

50% sparse (2:4)

+ Efficient
– Inflexible

Naïve Way to Increase Flexibility Structured Sparse Designs

Extend the Number of G:H Ratios Supported



Not Scalable: Hardware complexity increases approximately in proportion to the number of sparsity degrees

Hierarchical Structured Sparsity (HSS)

Compose G:H sparsity patterns in a hierarchical fashion

N-Rank HSS: G:H → G:H … → G:H

Rank N-1 *Rank N-2* *Rank 0*

What does a 3:4→2:4 pattern look like?

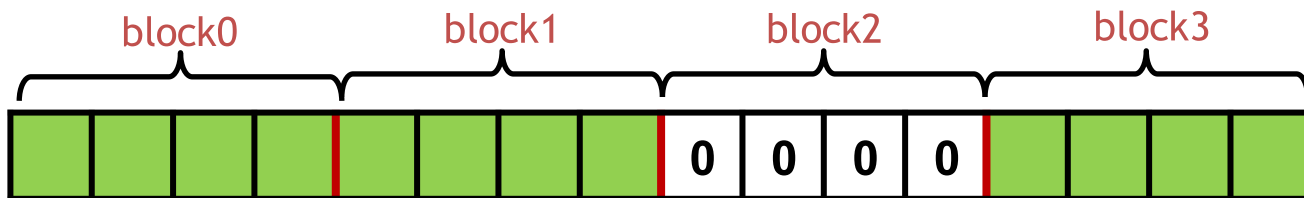
Dense Vector

Hierarchical Structured Sparsity (HSS)

Compose G:H sparsity patterns in a hierarchical fashion

What does a 3:4→2:4 pattern look like?

Rank1: 3 nonempty blocks out of the 4 blocks



Vector with Rank1 Sparsity Applied

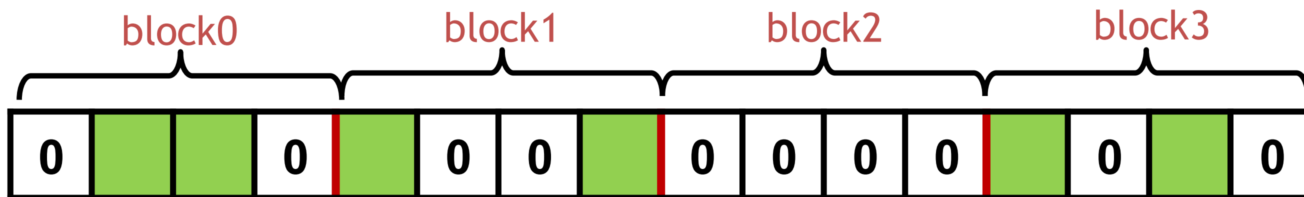
[**Wu**, *MICRO* 2023]

Hierarchical Structured Sparsity (HSS)

Compose G:H sparsity patterns in a hierarchical fashion

What does a 3:4→2:4 pattern look like?

Rank1: 3 nonempty blocks out of the 4 blocks
Rank0: 2 nonzero values out of 4 values within the block



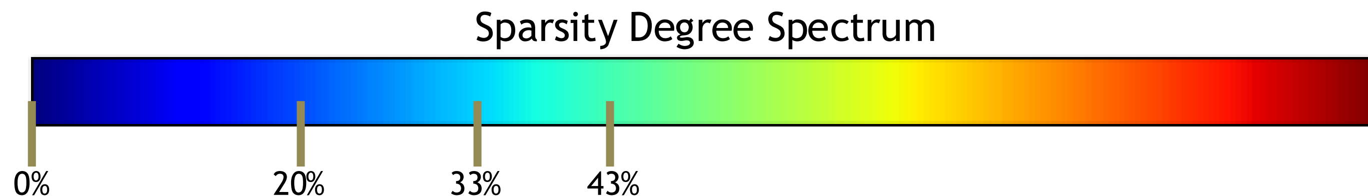
Vector with Both Ranks' Sparsity Applied

HSS Introduces a Flexible Way to Express Sparsity Degrees

4 sparsity degrees

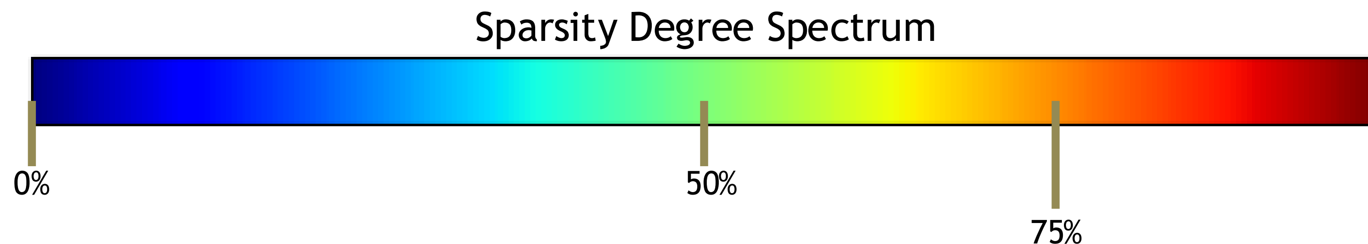
Rank 1			
4:4 (0%)	4:5 (20%)	4:6 (33%)	4:7 (43%)

Sparsity Degree Spectrum



[**Wu**, *MICRO* 2023]

HSS Introduces a Flexible Way to Express Sparsity Degrees



[**Wu**, *MICRO* 2023]

HSS Introduces a Flexible Way to Express Sparsity Degrees

[**Wu**, *MICRO* 2023]

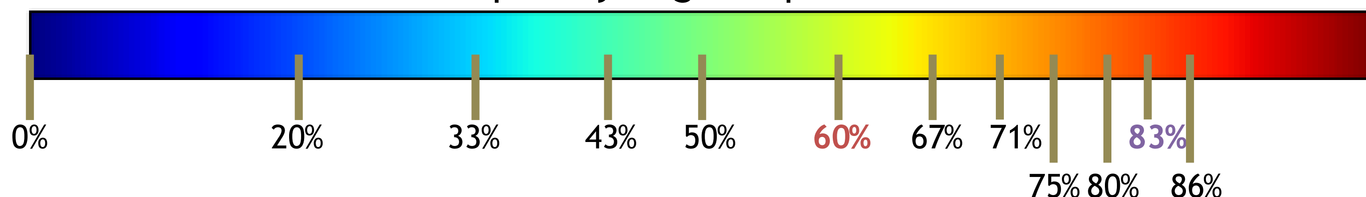
HSS Introduces a Flexible Way to Express Sparsity Degrees

Sparsity Degree Spectrum

[**Wu**, *MICRO* 2023]

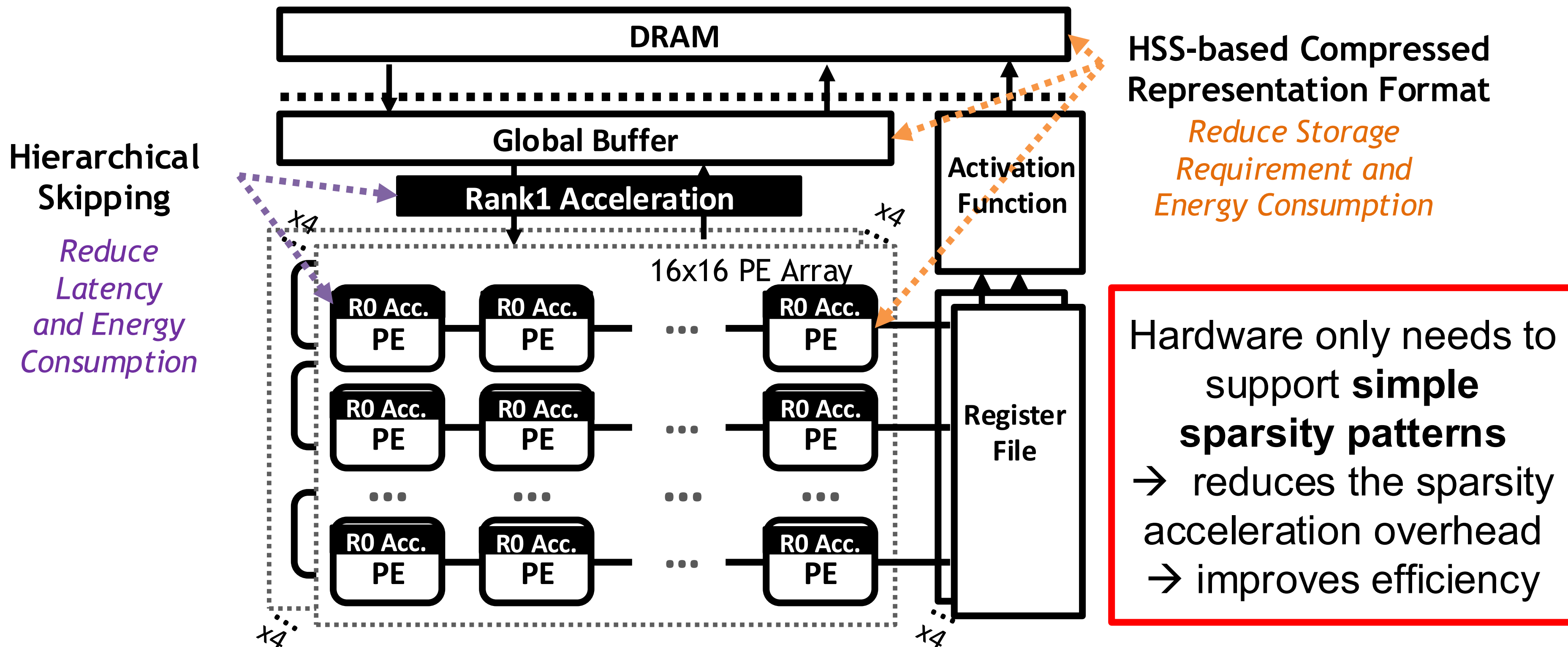
HSS Introduces a Flexible Way to Express Sparsity Degrees

4:4-4:4 (0%)	4:5-4:4 (20%)	4:6-4:4 (33%)	4:7-4:4 (43%)	4:4-2:4 (50%)	4:5-2:4 (60%)	4:6-2:4 (67%)	4:7-2:4 (71%)	4:4-1:4 (75%)	4:5-1:4 (80%)	4:6-1:4 (83%)	4:7-1:4 (86%)



Systematically represent diverse sparsity degrees by having them **hierarchically composed from multiple simple sparsity pattern**

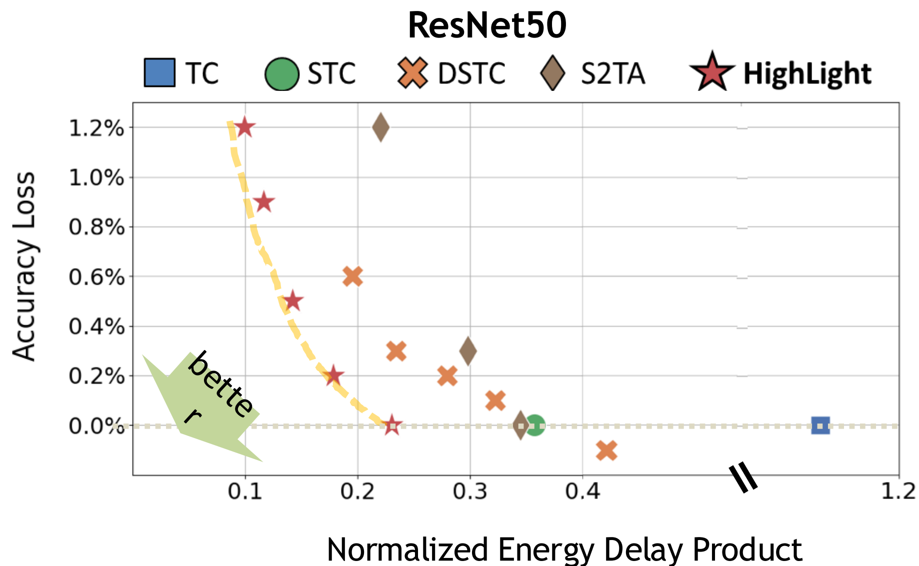
HighLight: Flexible and Efficient Sparse DNN Accelerator

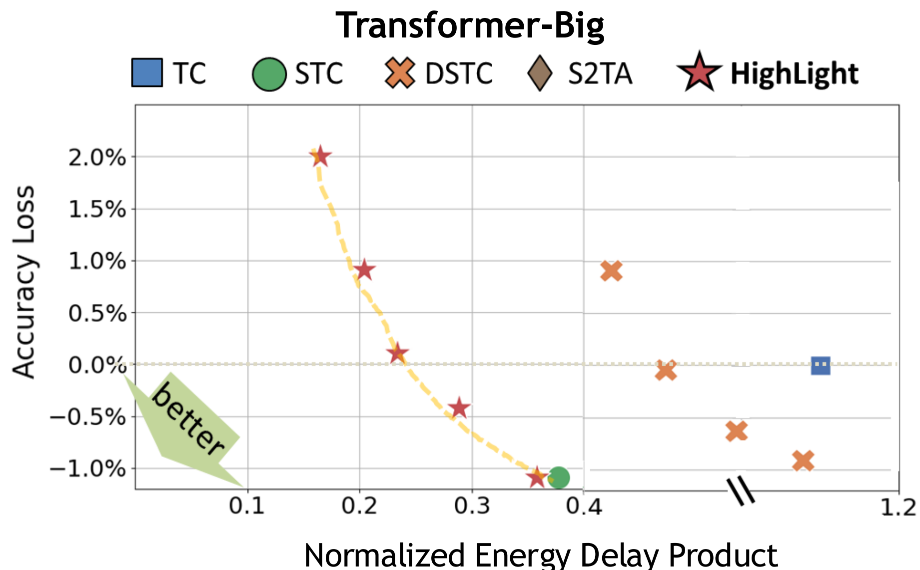


[Wu, *MICRO* 2023]

Accuracy-Energy Delay Product Pareto Frontier

We evaluate the designs with representative DNNs pruned to different sparsity degrees, each with its respective sparsity structure (if any)



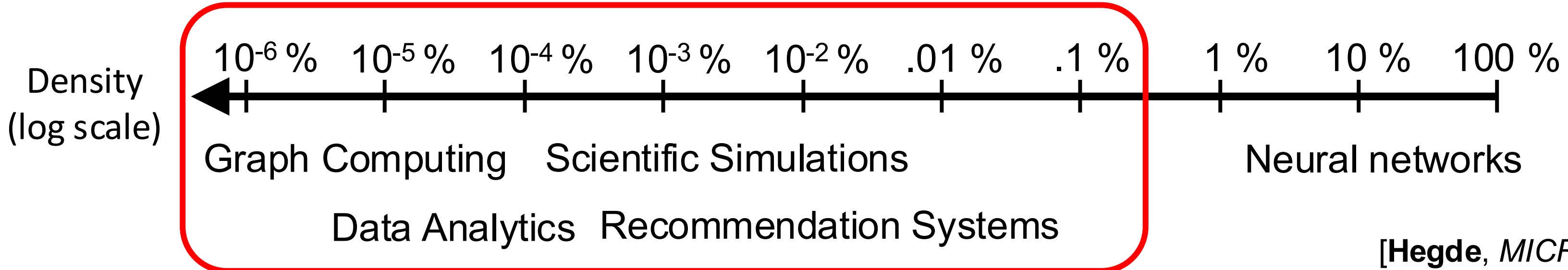


HighLight at the accuracy-energy delay product pareto frontier

[**Wu**, *MICRO* 2023]

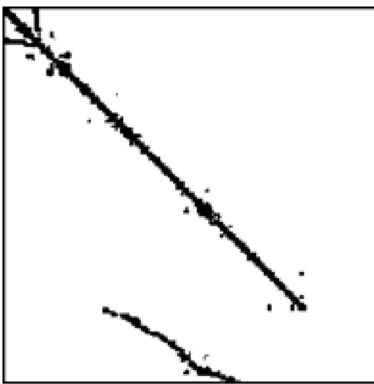
Handling Tiling with Sparsity

Tensor Sparsity Varies Widely Across Different Applications

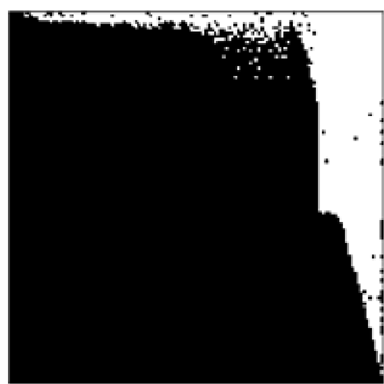


[**Hegde**, *MICRO 2019*]

367K nonzeros



2.33M nonzeros

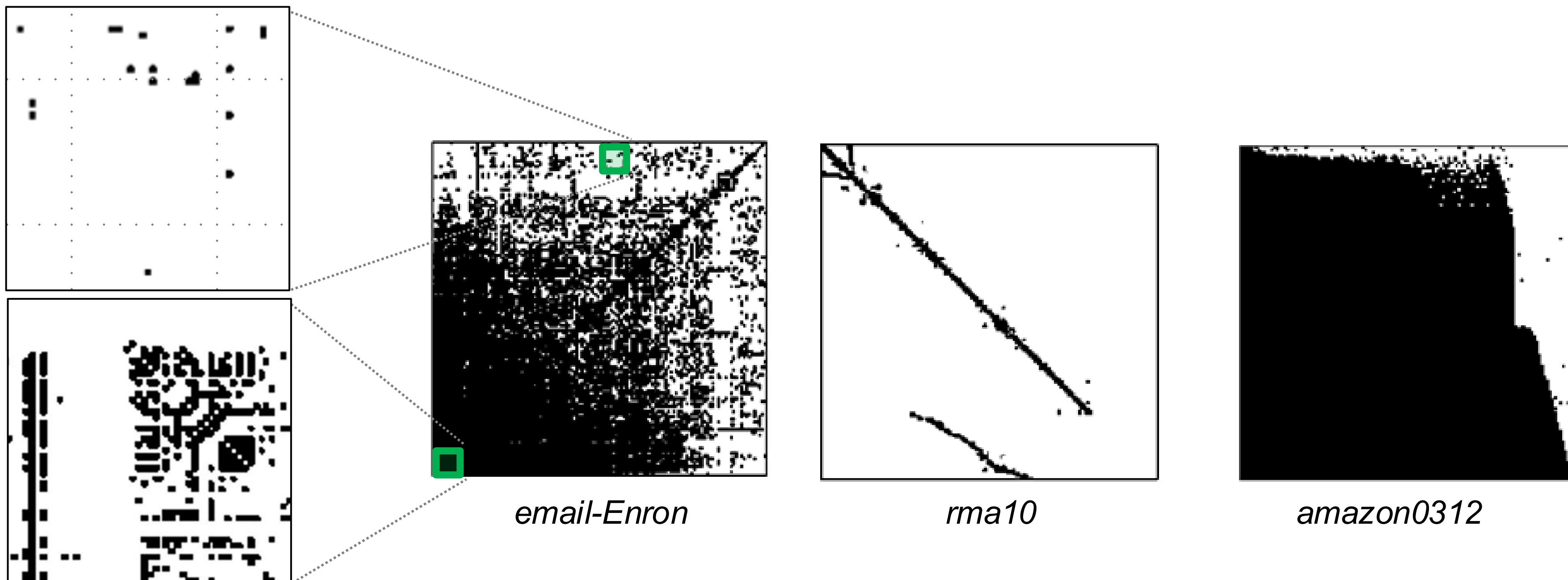


3.20M nonzeros

Zero Values

Nonzero Values

Tensors vary in the *distribution* of sparsity



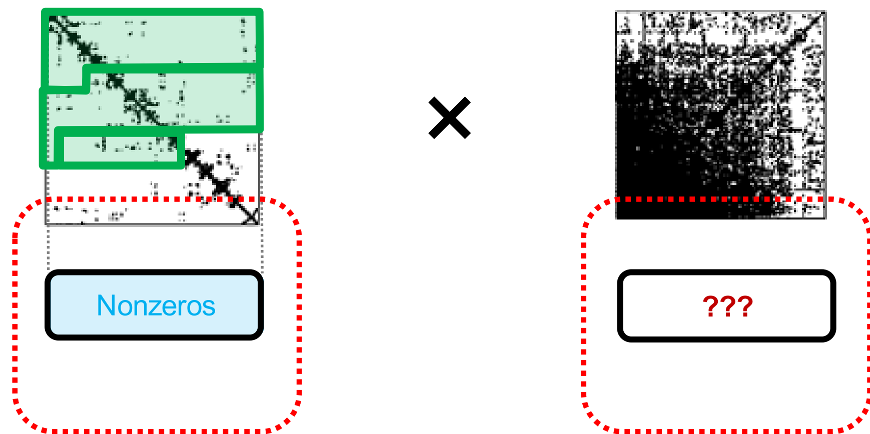
Key Challenge: Determining Tile Size

- Need to break up tensor into tiles to fit in buffer
- Larger tiles enables more data reuse for higher energy efficiency and throughput → pick largest tile that can fit in buffer
- Variation in sparsity distribution across tensor makes it difficult to determine largest tile size

Tiling for different forms of occupancy

Tiling with *uniform occupancy*

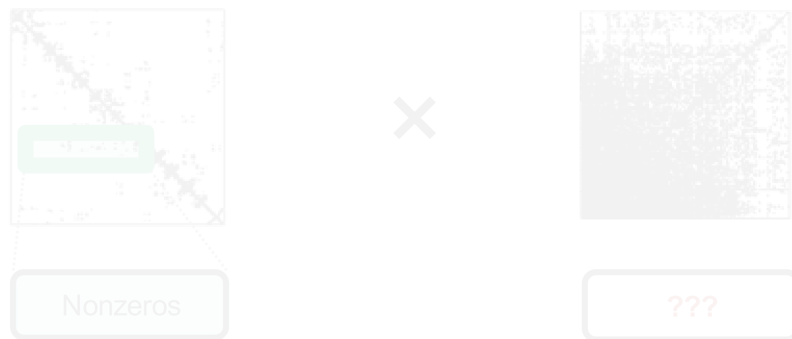
- low occupancy variation
- difficult to tile other operand



Tiling for different forms of occupancy

Tiling with *uniform occupancy*

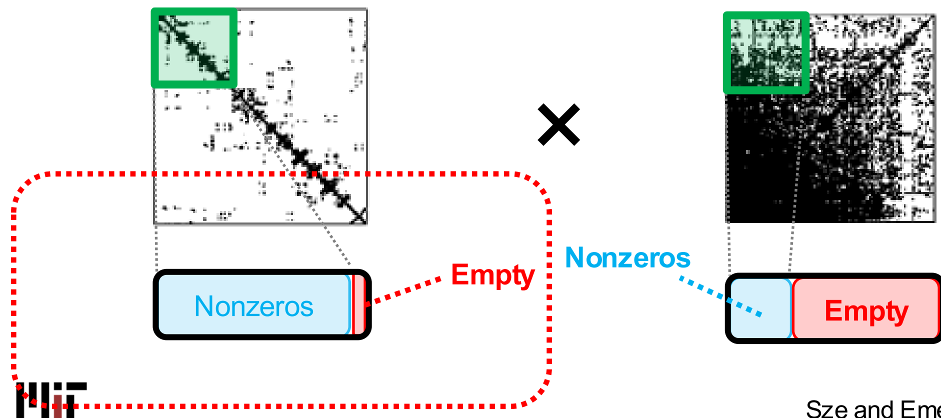
- low occupancy variation
- difficult to tile other operand



Tiling with *uniform shape*

\+ easy to tile both operands

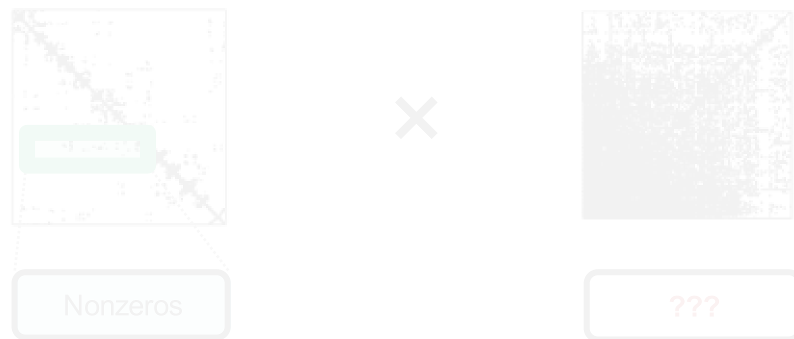
\- high occupancy variation



Tiling for different forms of occupancy

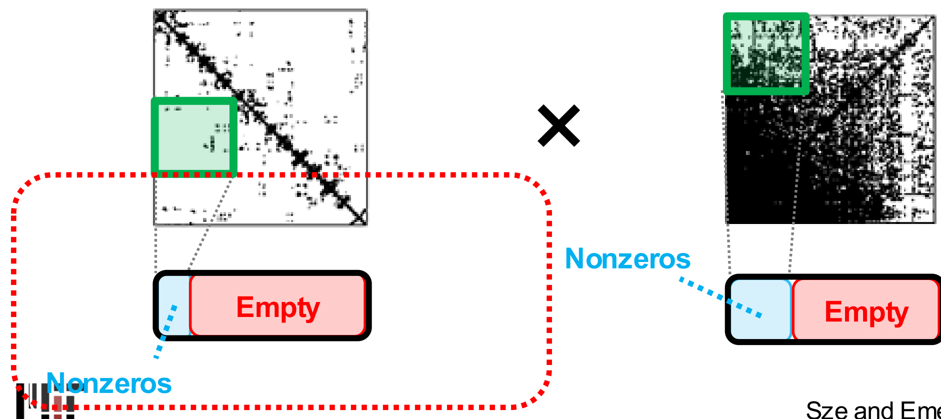
Tiling with *uniform occupancy*

- low occupancy variation
- difficult to tile other operand



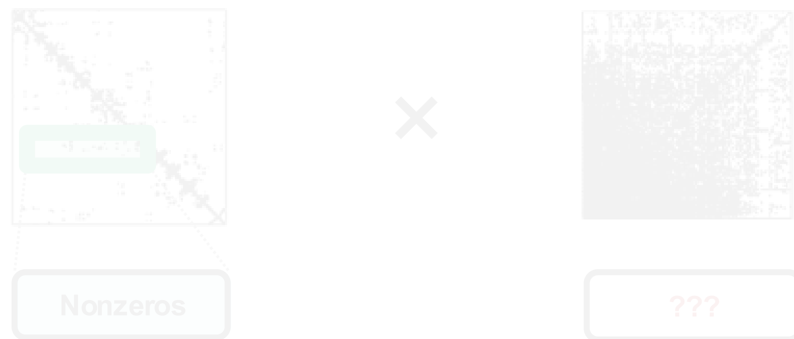
Tiling with *uniform shape*

+ easy to tile both operands
- high occupancy variation



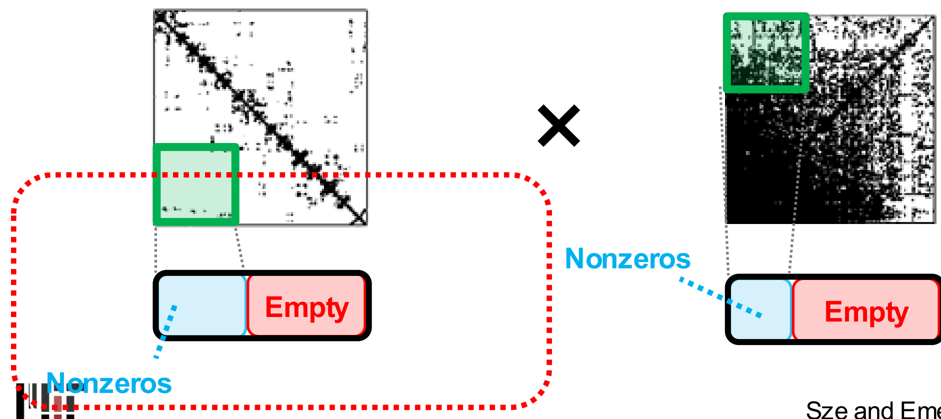
Tiling for different forms of occupancy

Tiling with *uniform occupancy*
- low occupancy variation
- difficult to tile other operand

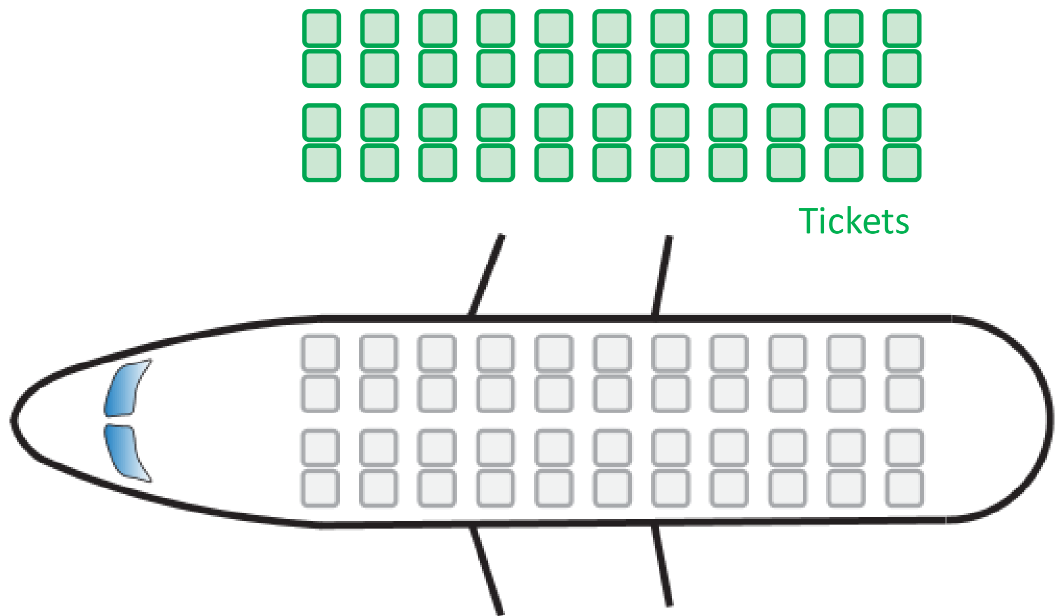


Tiling with *uniform shape*

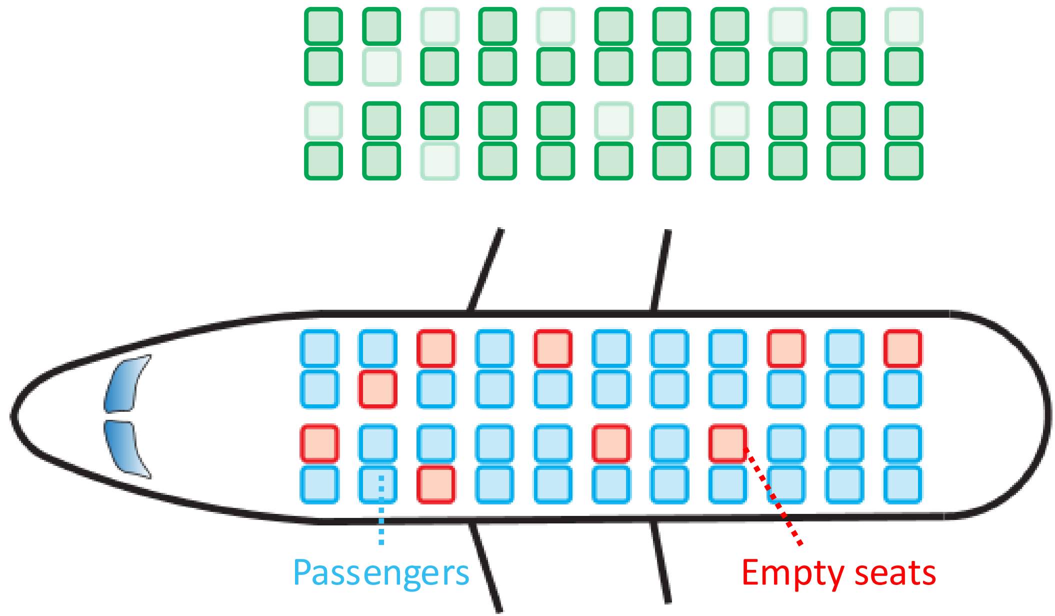
+ easy to tile both operands
- high occupancy variation



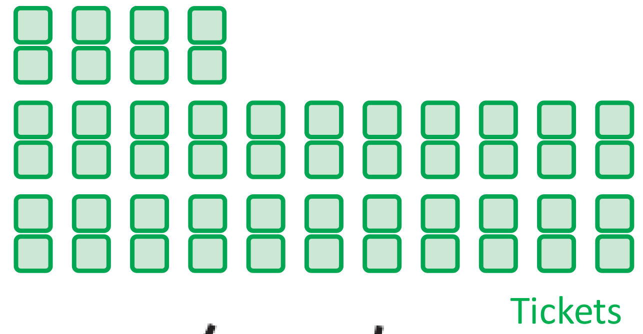
Overbooking improves utilization

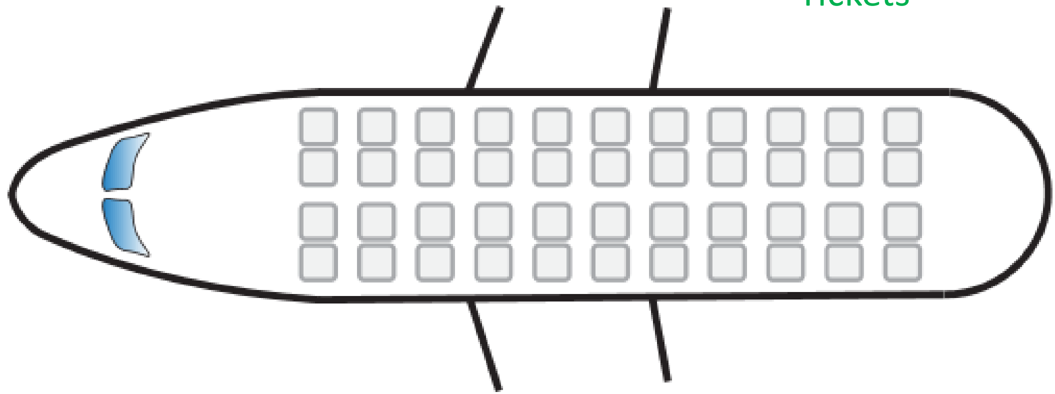


Overbooking improves utilization

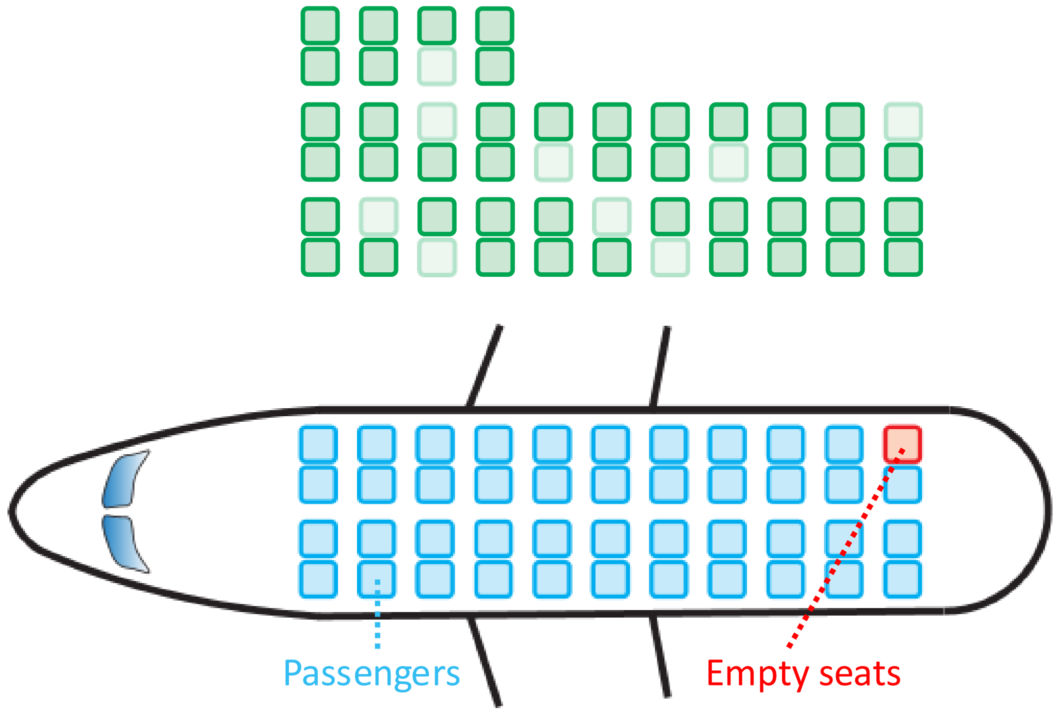


Overbooking improves utilization

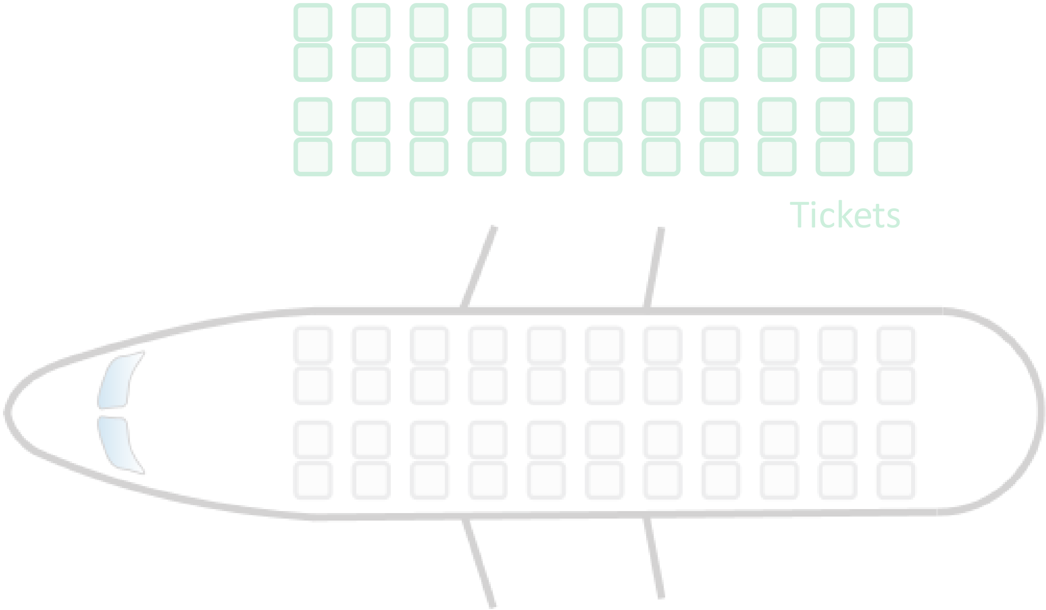


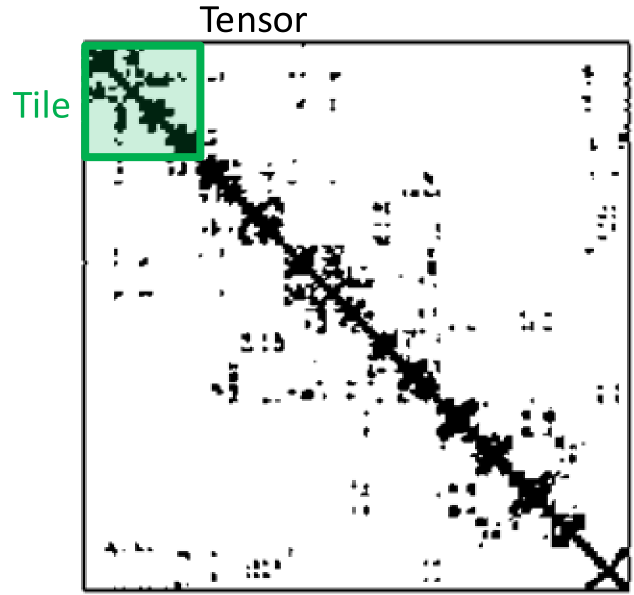


Overbooking improves utilization

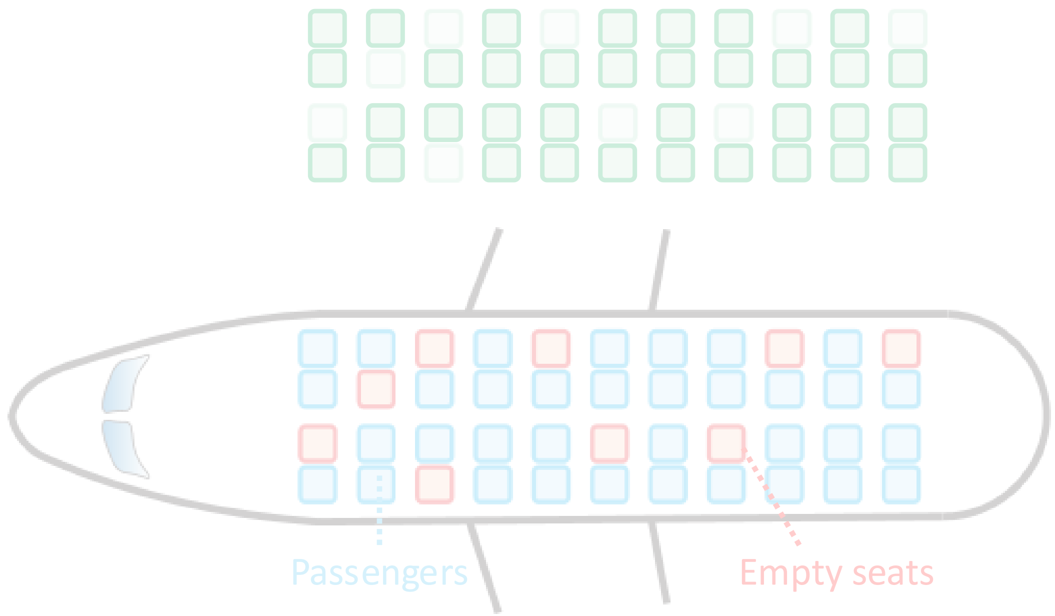


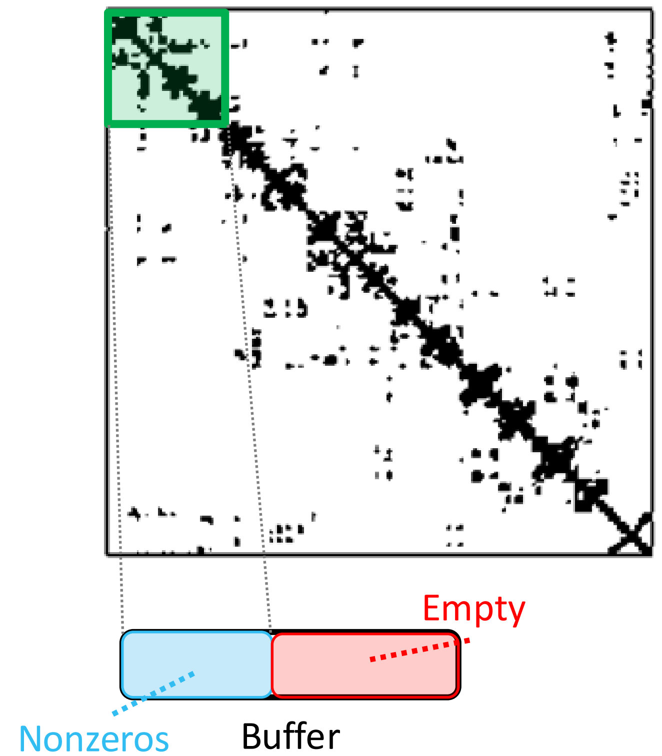
Overbooking improves utilization





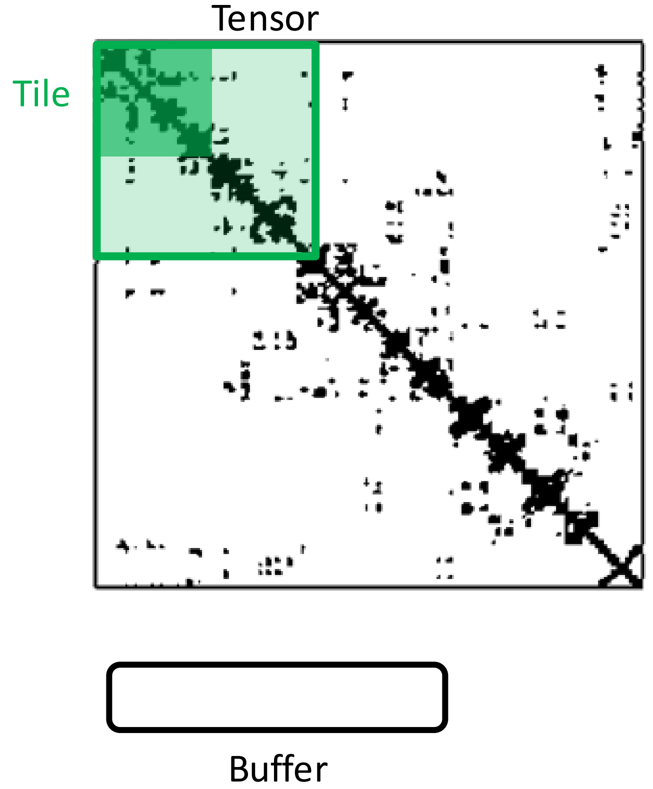
Overbooking improves utilization



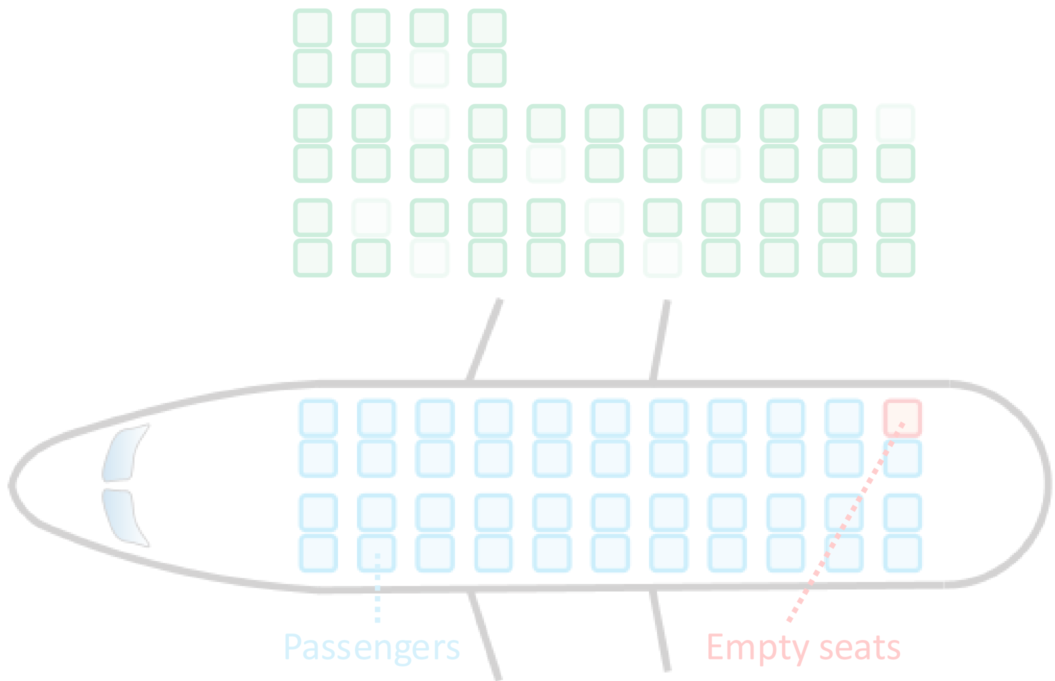


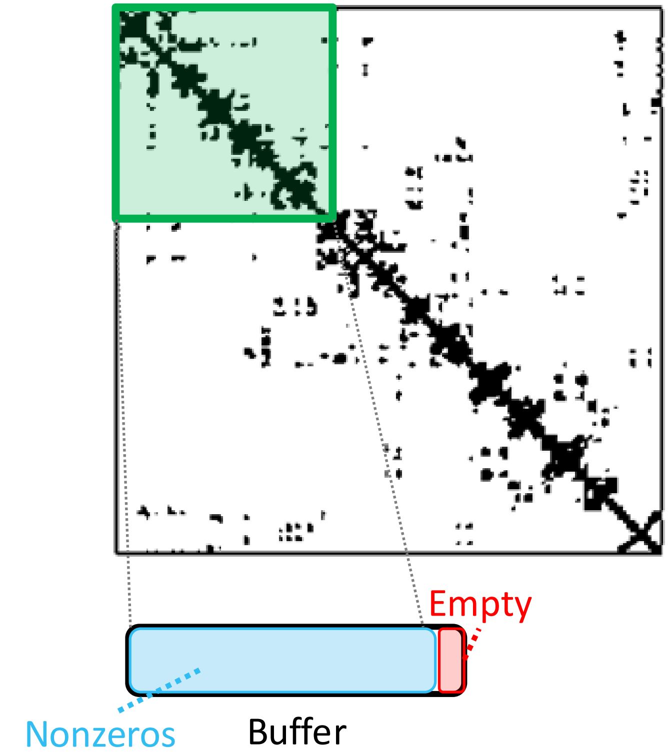
Overbooking improves utilization

Tickets



Overbooking improves utilization

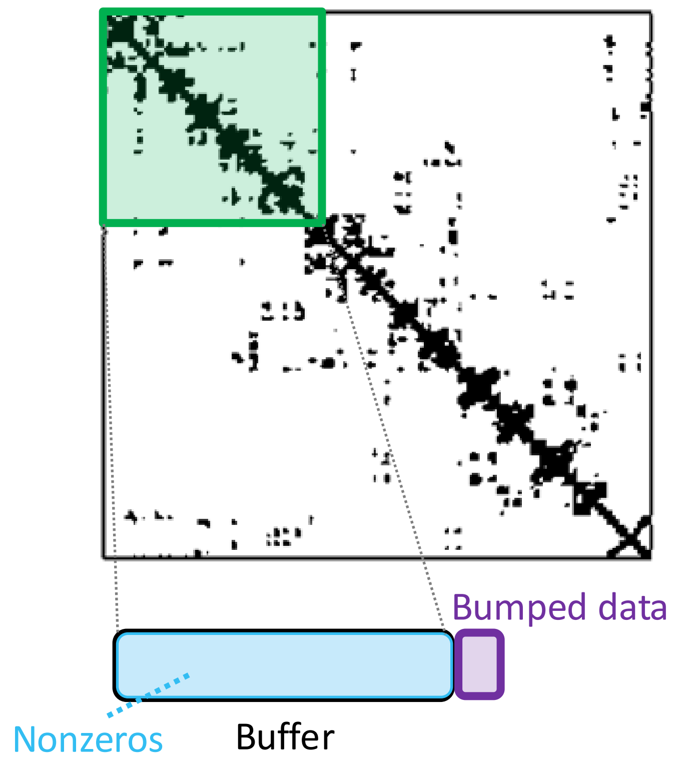




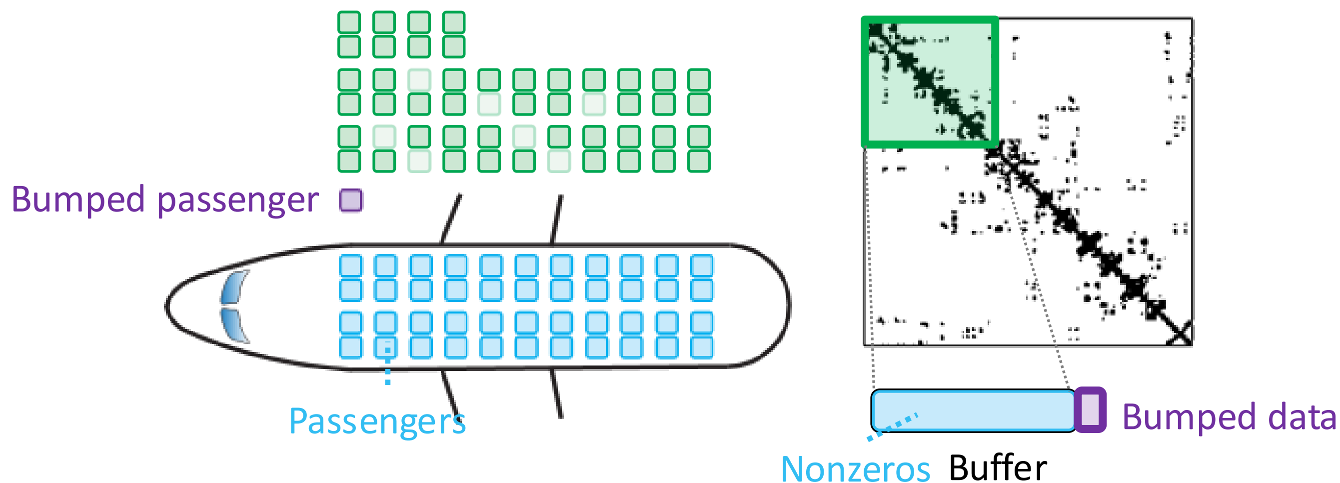
Overbooking improves utilization

Bumped passenger

Passengers



Overbooking improves utilization



How do we deal with the bumped data?

Tailors

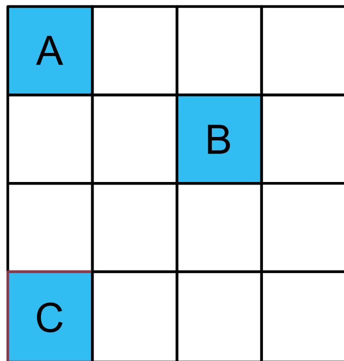
How do we determine how much to overbook?

Swiftiles

[Xue, *MICRO* 2023]

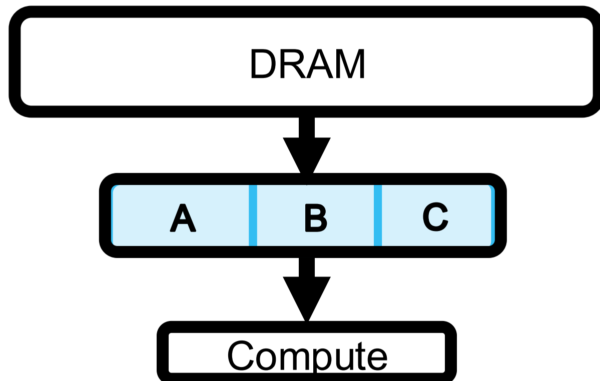
Tailors: Tail Overbooked Buffers for overbooked data

Unbumped (max reuse)



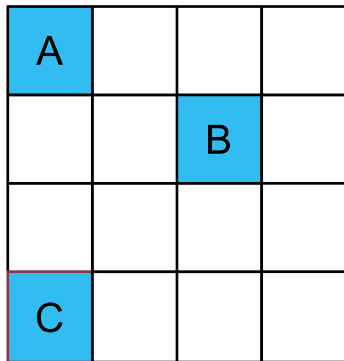
Traversal order

A B C A B C ...

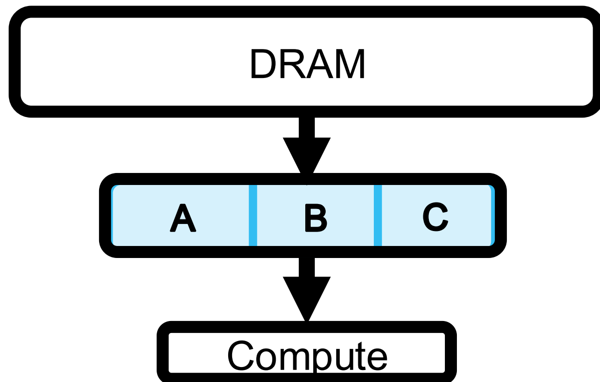


Tailors: Tail Overbooked Buffers for overbooked data

Unbumped (max reuse)

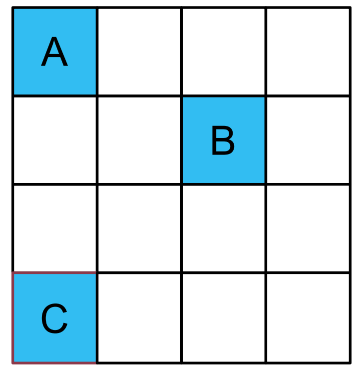


Traversal order
A B C A B C ...

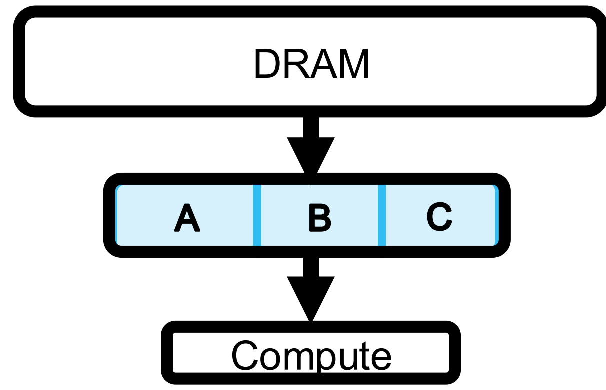


Tailors: Tail Overbooked Buffers for overbooked data

Unbumped (max reuse)

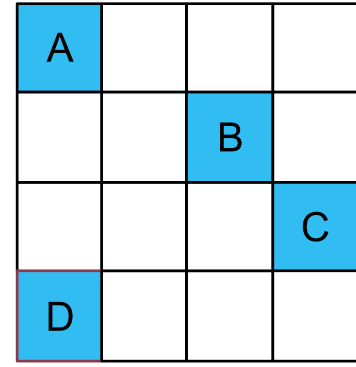


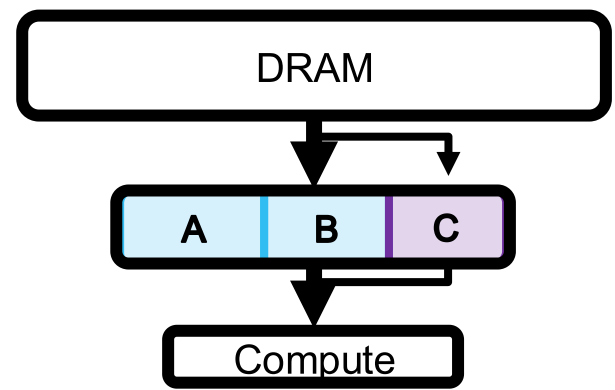
Traversal order
A B C A B C ...



Bumped (stream)

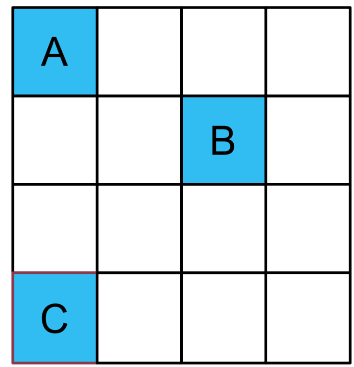
Traversal order
A B C D A B C D ...





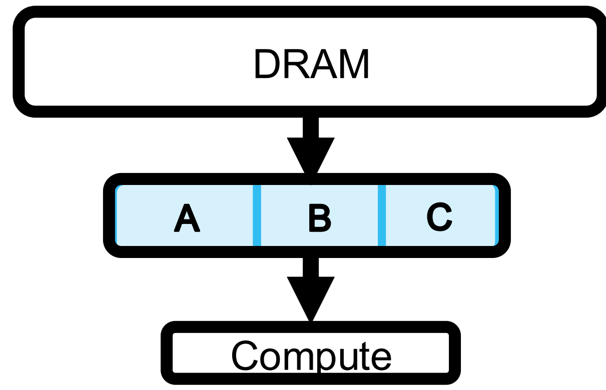
Tailors: Tail Overbooked Buffers for overbooked data

Unbumped (max reuse)



Traversal order

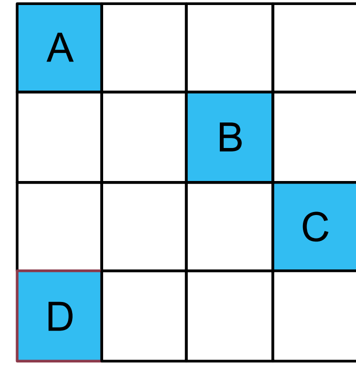
A B C A B C …

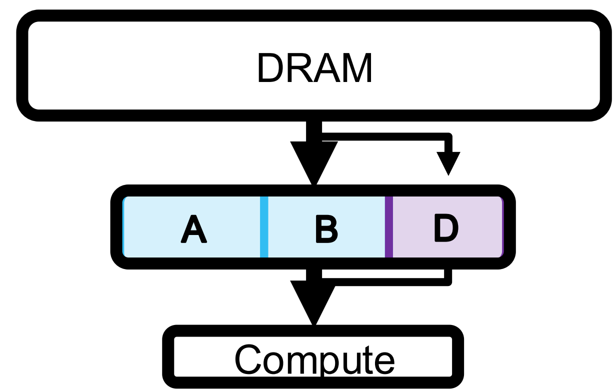


Bumped (stream)

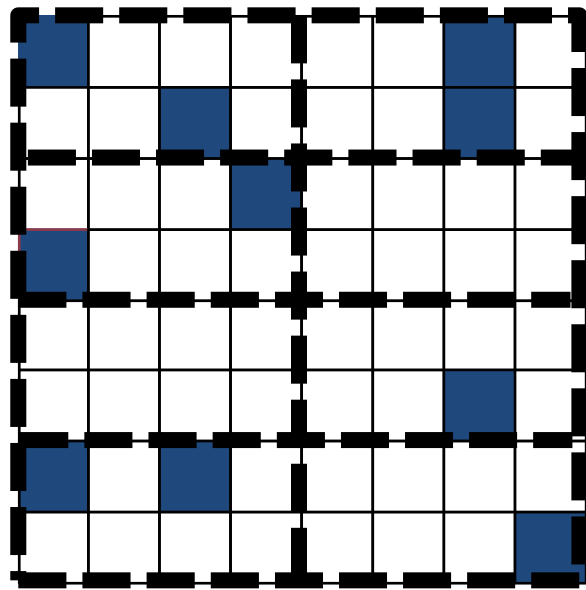
Traversal order

A B C D A B C D …

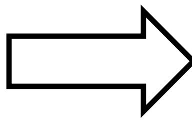




Determining the size of a tile is challenging



Full traversal to find maximum occupancy



Cumulative Tiles (%)

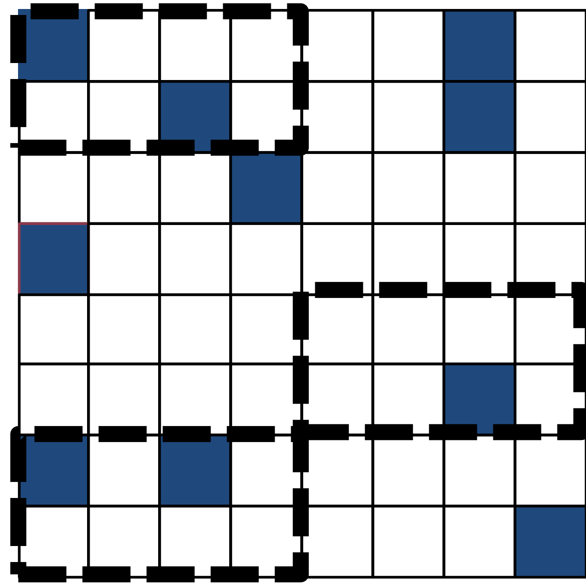
100
80
60
40
20
0

0 50 100 150

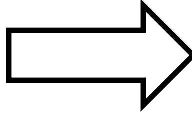
% of buffer occupied

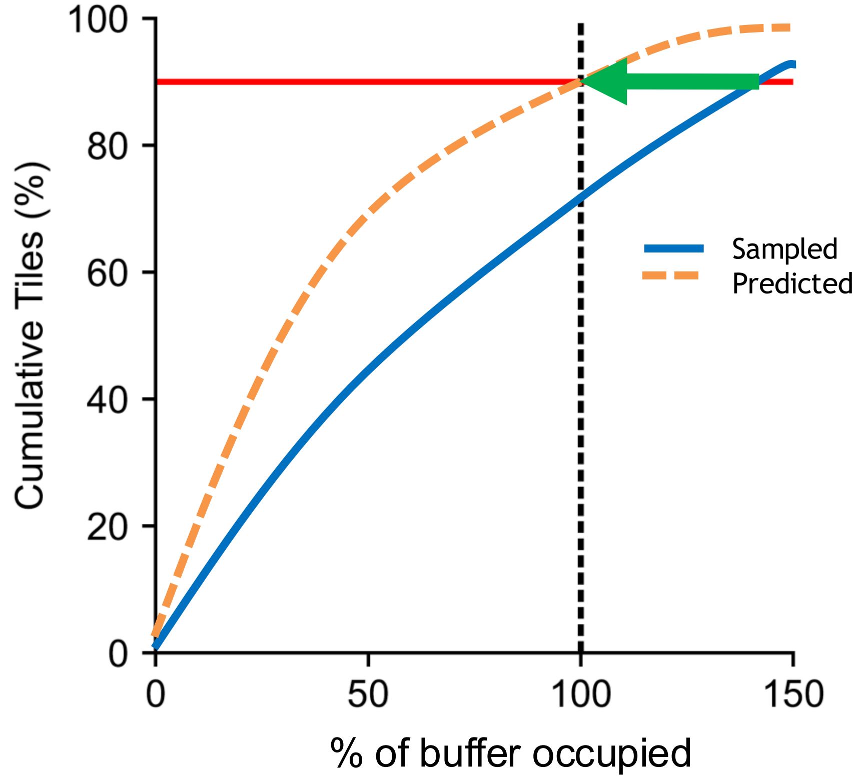
Ideal (every tile fits)

Swiftiles: A swift tiling algorithm for overbooking



Random sampling to generate approximate occupancy distribution

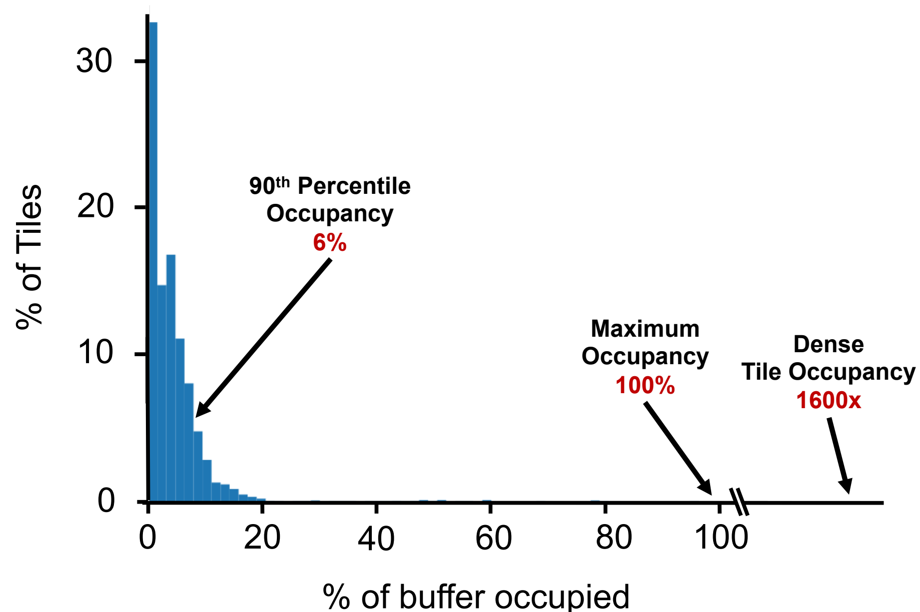




Scale distribution to buffer size

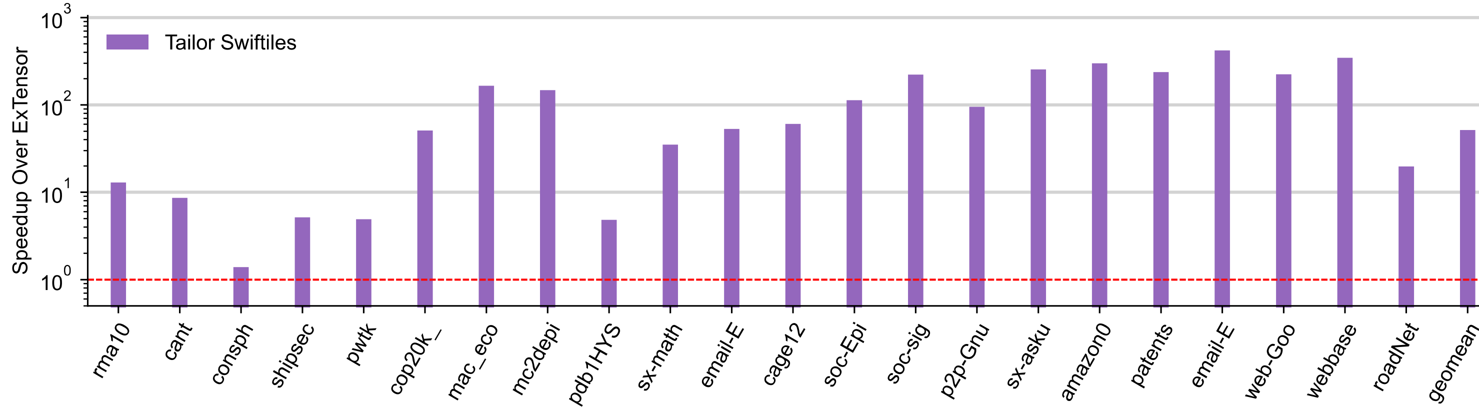
Evaluation against other tiling options

- ExTensor-Naive
  - No knowledge of tile occupancy, so must tile assuming dense tiles
- ExTensor-Prescient
  - Uses the maximum tile size where all tiles still fit in the buffer
- **ExTensor-Overbooking**
  - Tailors+Swiftiles where 90% of tiles fit in buffer

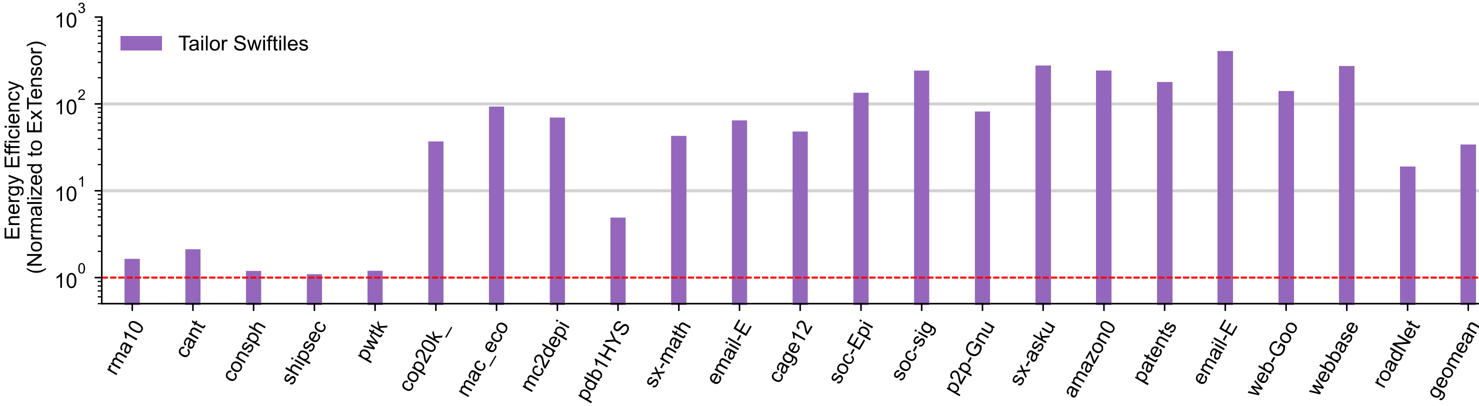


ExTensor [**Hegde**, *MICRO 2019*]

Impact of Overbooking with Tailor Swiftiles



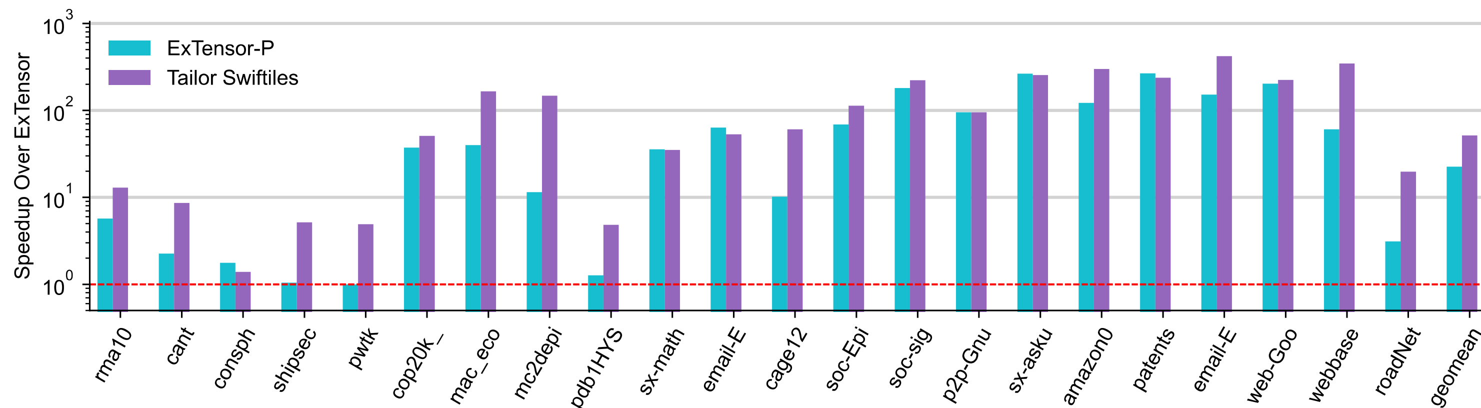
52.7x speed up over ExTensor-N [*MICRO* 2019]



22.5x higher energy efficiency than ExTensor-N

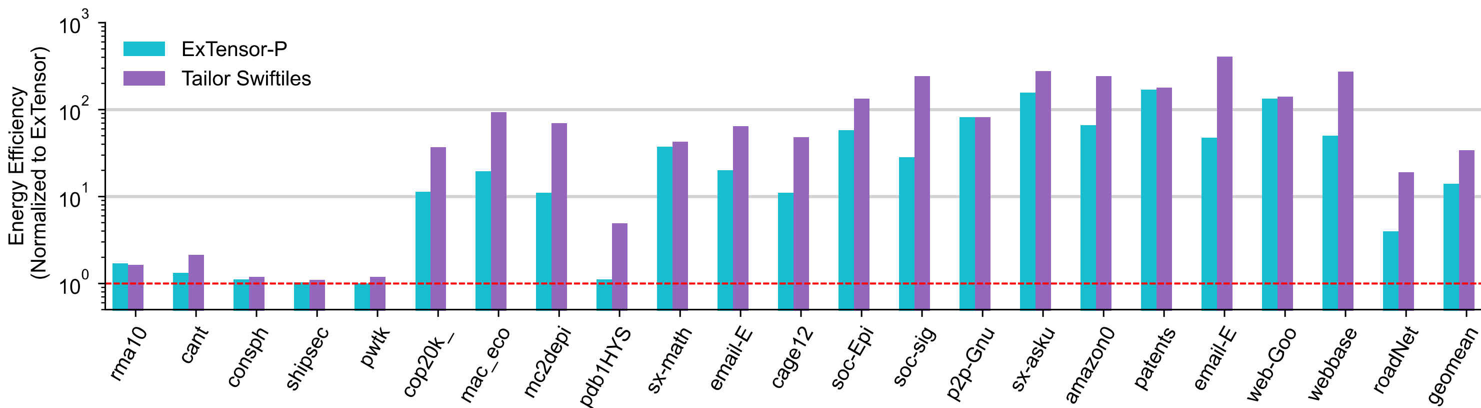
[**Xue**, *MICRO* 2023]

Impact of Overbooking with Tailor Swiftiles



ExTensor-P **(preprocess tile size based on sparsity)**

2.3x speed up over ExTensor-P



2.5x higher energy efficiency than ExTensor-P

[**Xue**, *MICRO* 2023]

Interplay Between Dataflow and Sparsity

- **Dataflow** (loop order)
  - Align loop order with storage order ***and rank order of representation format*** for concordant traversal (improve spatial locality and ***reduce data access overhead***)
  - Increase reuse for given data type (weight, input, output) ***and amortize data access overhead*** by increasing stationarity (move to outer/top loop)
- When choosing which loops to parallelize (spatial_for) need to consider sparsity and how it affects workload imbalance across PEs (PE utilization)
- See Chapter 8.3 on Sparse Dataflows
- **Explore in Final Project!**

Summary

- Exploiting sparsity makes processing irregular!
  - The number of non-zeros vary within and across tensors, causing variation in the number of cycles and the amount of required storage
  - The coordinate of non-zeros may also be unknown
- Variation causes
  - Underutilization of buffers and PEs
  - Workload Imbalance
  - Random data access
- Overhead (must not exceed benefits of sparsity)
  - Storage of coordinate information (metadata) for compressed data
  - Intersection logic for checking if either inputs are zero
- **Lots of challenges in sparse deep neural network acceleration!**

Recommended Reading

- N. Wu, P.-A. Tsai, Po-An, A. Parashar, V. Sze, J. S. Emer
  - Paper: https://sparseloop.mit.edu/documents/2022-micro-sparseloop.pdf
  - Project website: https://sparseloop.mit.edu/ (you will use this tool for Lab 4)

- Textbook: Section 8.2 (Compression) and 8.3 (Sparse Dataflows)
  - https://doi.org/10.1007/978-3-031-01766-7