CUDA-Based Acceleration of PNG Image Encoder and Decoder

Hongbin Liu, Developer Technology | GTC2023

Tong Liu, Developer Technology(Intern) | GTC 2023

CUDA-Based Acceleration of PNG Image Encoder and Decoder

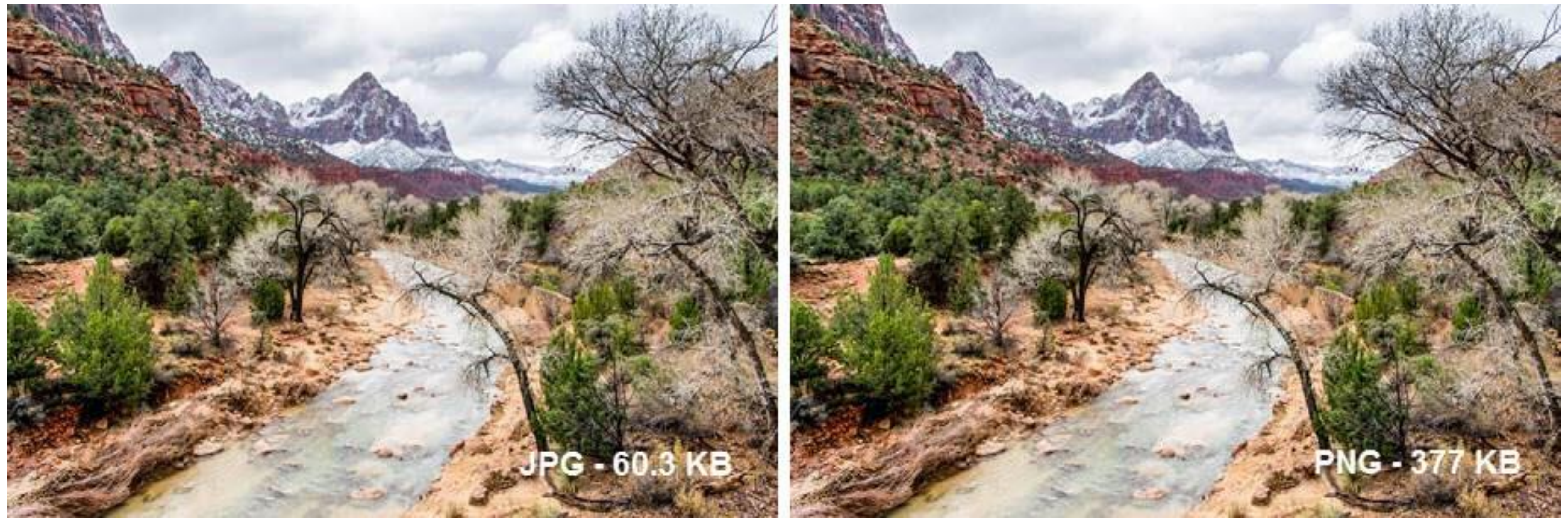
Contents

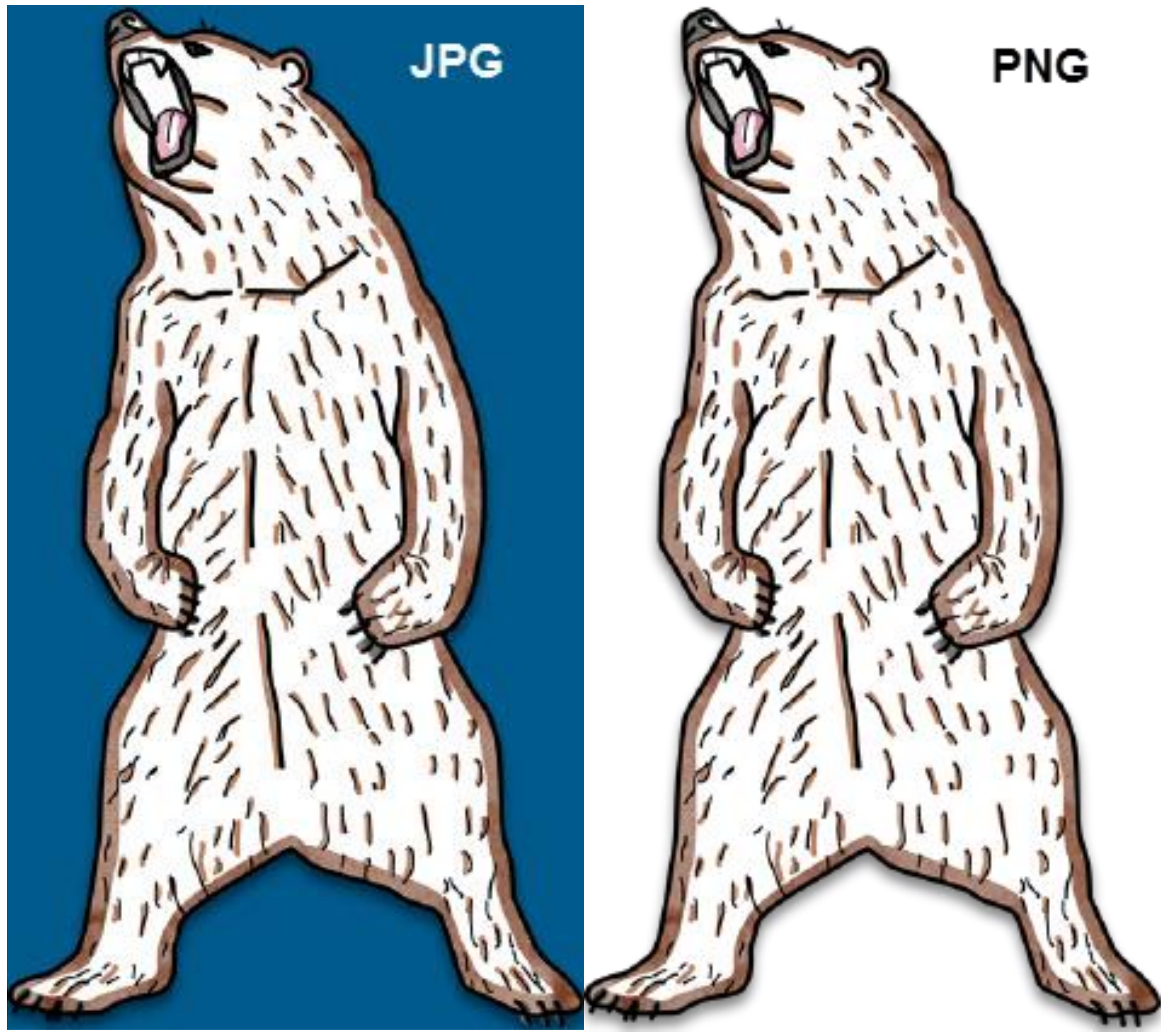
- PNG image and its encoding/decoding process
- Key optimizations in PNG encoder
  - Run Length Encoding
  - Huffman encoding
- Key optimizations in PNG decoder
  - De-Filter
- Performance
- Summary

PNG Image and Its Encoding/Decoding Process

What is PNG image?

- Portable Network Graphics(PNG)
- Lossless compression
- Transparency support





PNG Image and Its Encoding/Decoding Process

PNG file structure

ILSVRC2012_val_00031727,
From ImageNet

Start offset	Raw bytes	Chunk inside
0	89 50 4e 47 0d 0a 1a 0a	**Special: File signature** **Length: 8 bytes**
8	00 00 00 0d 49 48 44 52 00 00 02 d0 00 00 05 00 08 02 00 00 00 e1 ac f2 6a	**Data length(4bytes): 13 bytes** **Type(4bytes): IHDR** **Width(4bytes): 720 pixels** **Height(4bytes): 1280 pixels** **Bit depth(1byte): 8 bits per channel** **Color type(1byte): RGB (2)** **Compression method(1byte): DEFLATE (0)** **Filter method(1byte): Adaptive (0)** **Interlace method(1byte): None (0)** **CRC-32(4bytes): E1ACF26A**
33	00 00 20 00 49 44 41 54 78 01 94 c1 81 b2 24 47 b2 5d d7 bd … 92 1e 49 be fc f4 b5 6f eb 6c db 91	**Data length(4bytes): 8192 bytes** **Type(4bytes): IDAT** **Pixel data(8192bytes): …** **CRC-32(4bytes): EB6CDB91**
…	…	…
869220	00 00 00 00 49 45 4e 44 ae 42 60 82	**Data length(4): 0 bytes** **Type(4bytes): IEND** **CRC-32(4): AE426082**

PNG header, sole

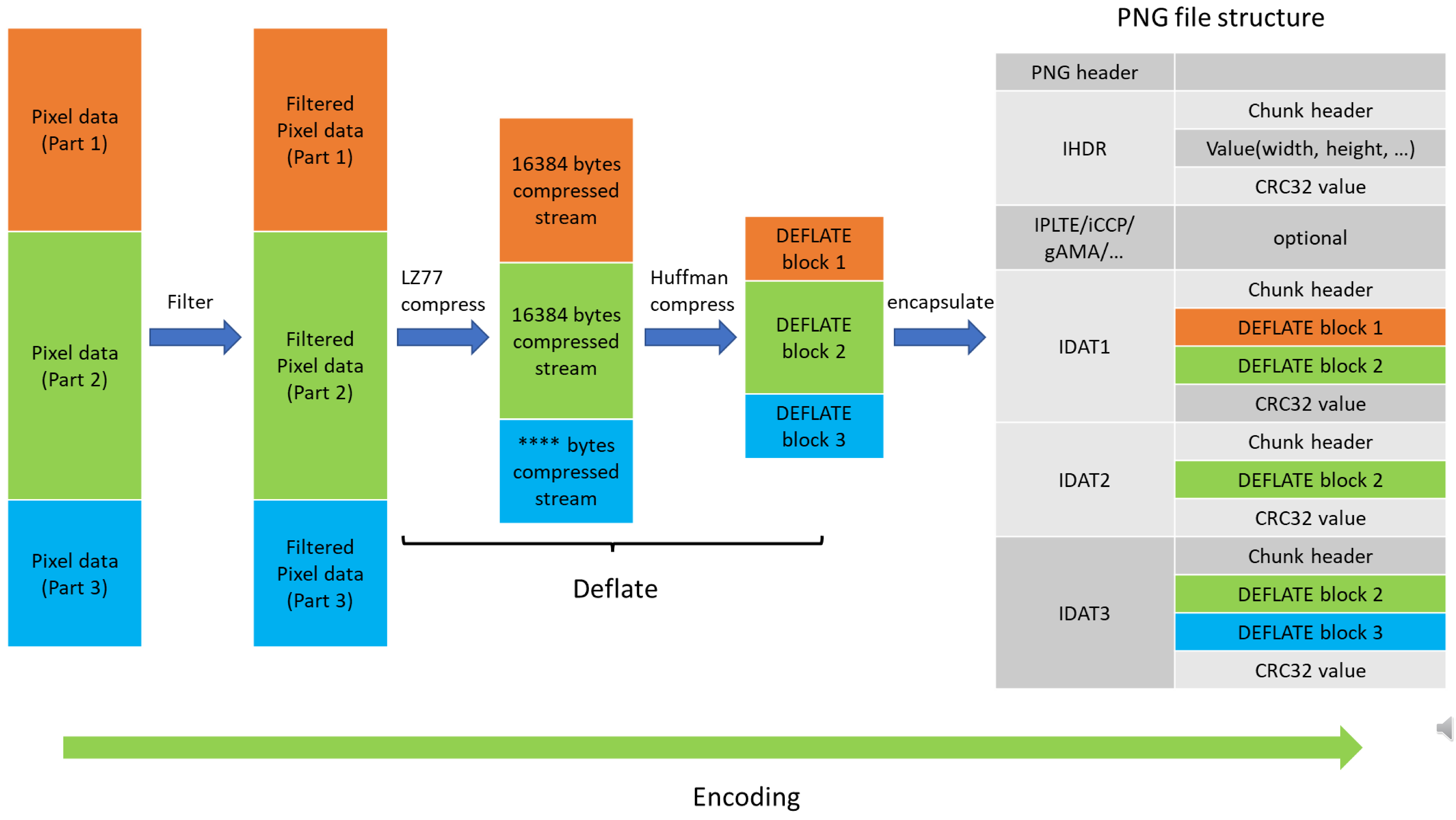
IHDR chunk, sole and the first chunk

A series of IDAT chunks and other ancillary chunks

IEND chunk, sole and the last chunk

PNG Image and Its Encoding/Decoding Process

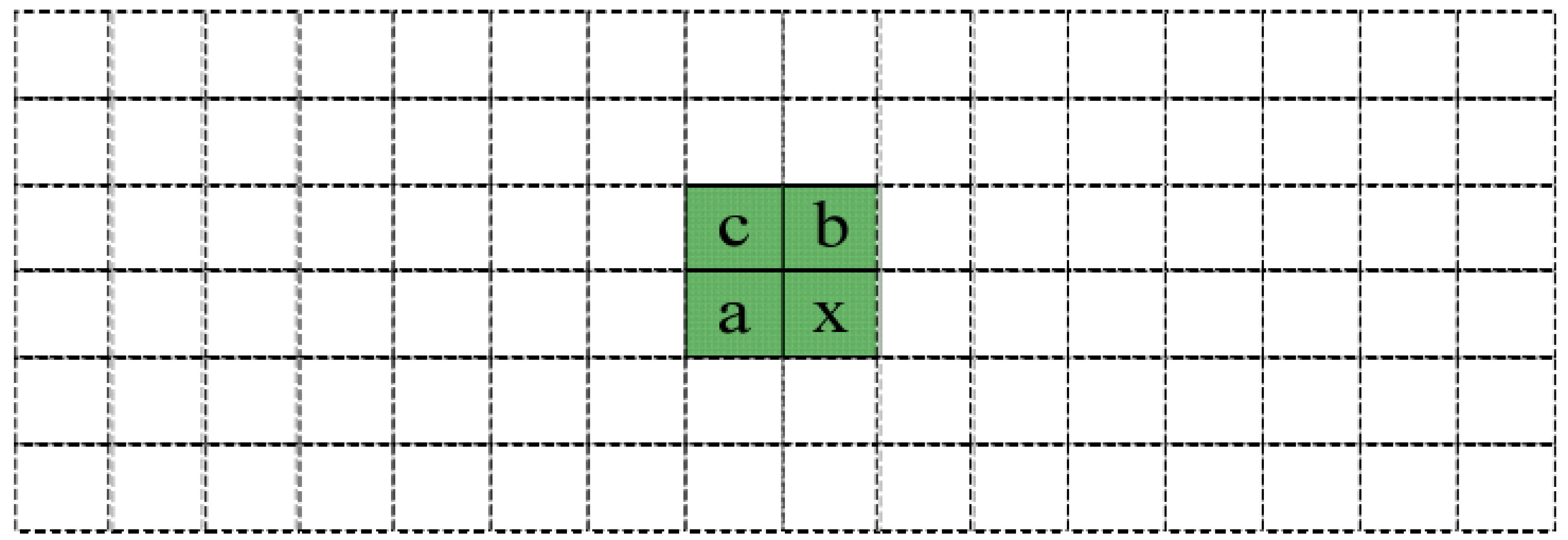
Encoding Process

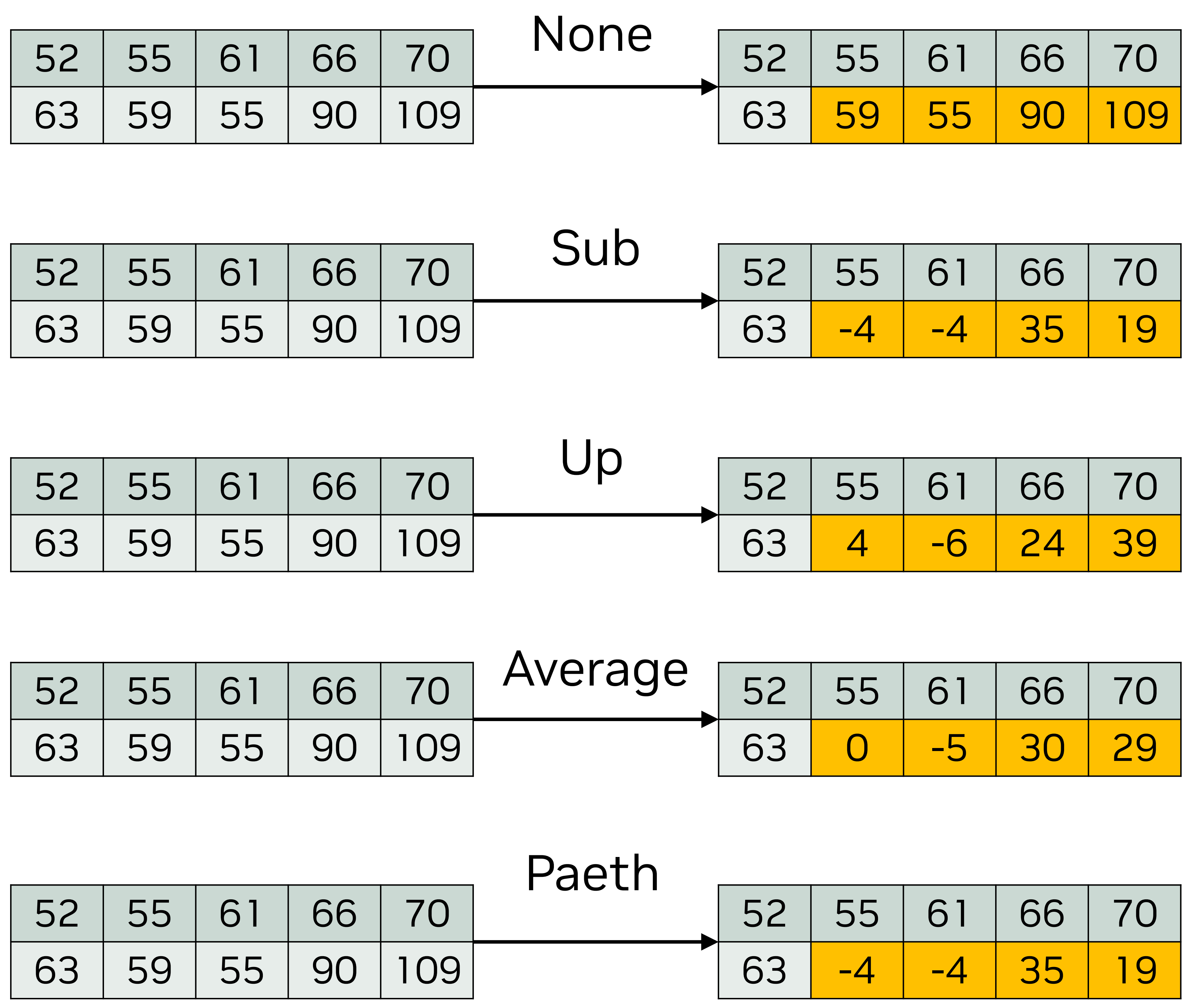


PNG Image and Its Encoding/Decoding Process

Filter algorithm

- Five filter types
- Each row has its own filter type
- Lossless and element-wise
- Improve the compressibility of the data





None

52	55	61	66	70
63	59	55	90	109

$\rightarrow$

52	55	61	66	70
63	59	55	90	109

Sub

52	55	61	66	70
63	59	55	90	109

$\rightarrow$

52	55	61	66	70
63	-4	-4	35	19

Up

52	55	61	66	70
63	59	55	90	109

$\rightarrow$

52	55	61	66	70
63	4	-6	24	39

Average

52	55	61	66	70
63	59	55	90	109

$\rightarrow$

52	55	61	66	70
63	0	-5	30	29

Paeth

52	55	61	66	70
63	59	55	90	109

$\rightarrow$

52	55	61	66	70
63	-4	-4	35	19

Type byte	**Filter name**	**Filter function**
0	None	x'=x
1	Sub	x'=x-a
2	Up	x'=x-b
3	Average	x'=x-(a+b)/2
4	Paeth	x'=Paeth(x, a, b, c)

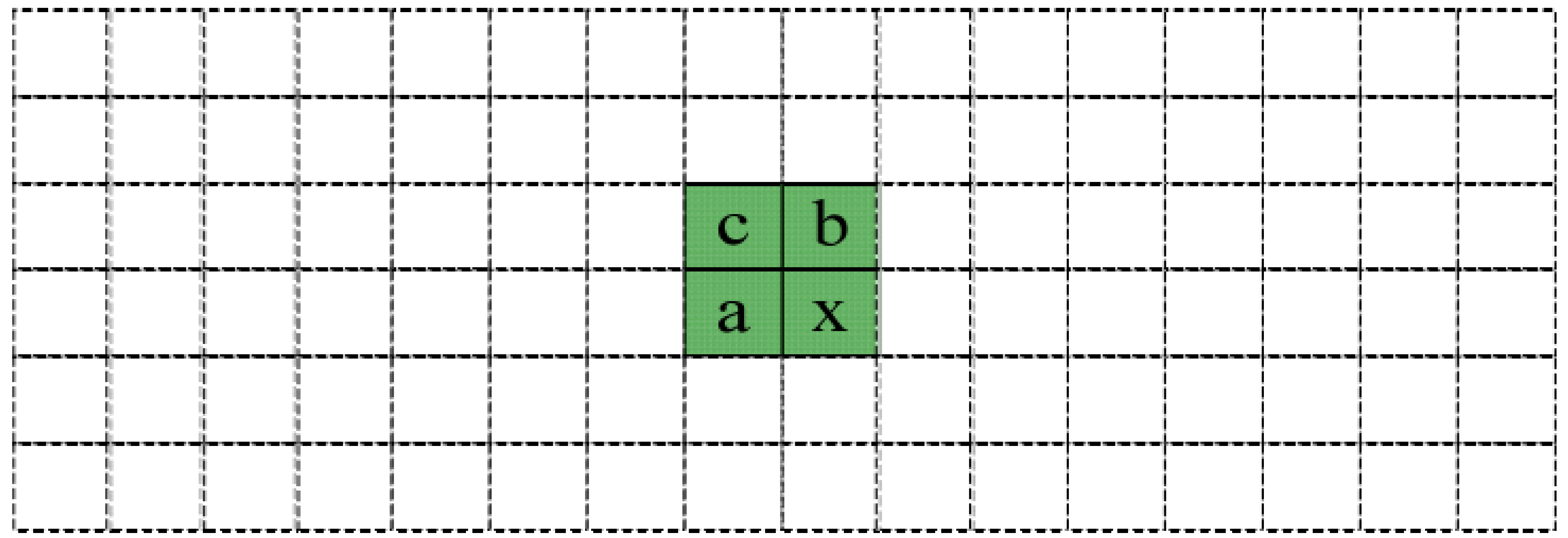
The superscript " ' " indicates the filtered value.

PNG Image and Its Encoding/Decoding Process

How to De-Filter?

- For each scanline, identify the filter type;
- Process the first pixel since it's boundary;
- Decode the other pixels with the inverse function;
- Move to the next scanline

- For each update, use the updated value of neighbors
- "Implicit" stencil problem
- Strong data dependency leads to bad parallelism

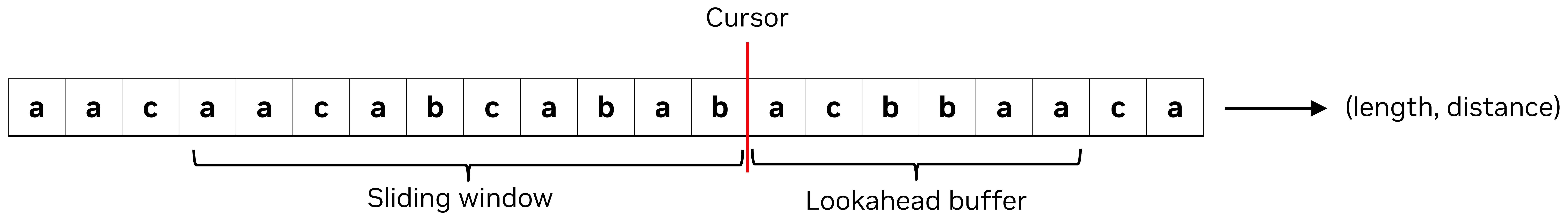


Type byte	Filter name	De-Filter function
0	None	x=x'
1	Sub	x=x'+a
2	Up	x=x'+b
3	Average	x=x'+(a+b)/2
4	Paeth	x=Paeth^{-1}(x', a, b, c)

x' and x denote the filtered and original value respectively,
And the same as a'/a, b'/b, c'/c

PNG Image and Its Encoding/Decoding Process

LZ77 compress



- Sliding window: a window of some width immediately preceding the current position in the data stream
- Cursor: the current position waiting to be encoded
- Lookahead buffer: a string buffer of some width waiting to find the longest match sub-string
- length: length of the longest match sub-string
- distance: distance between the cursor and the start position of matched sub-string

- In this case, length=2, distance=9

PNG Image and Its Encoding/Decoding Process

A special variant of LZ77 compress: Run-length encoding

a	a	c	a	a	c	a	b	c	a	b	a	b	a	c	b	b	a	a	c	a

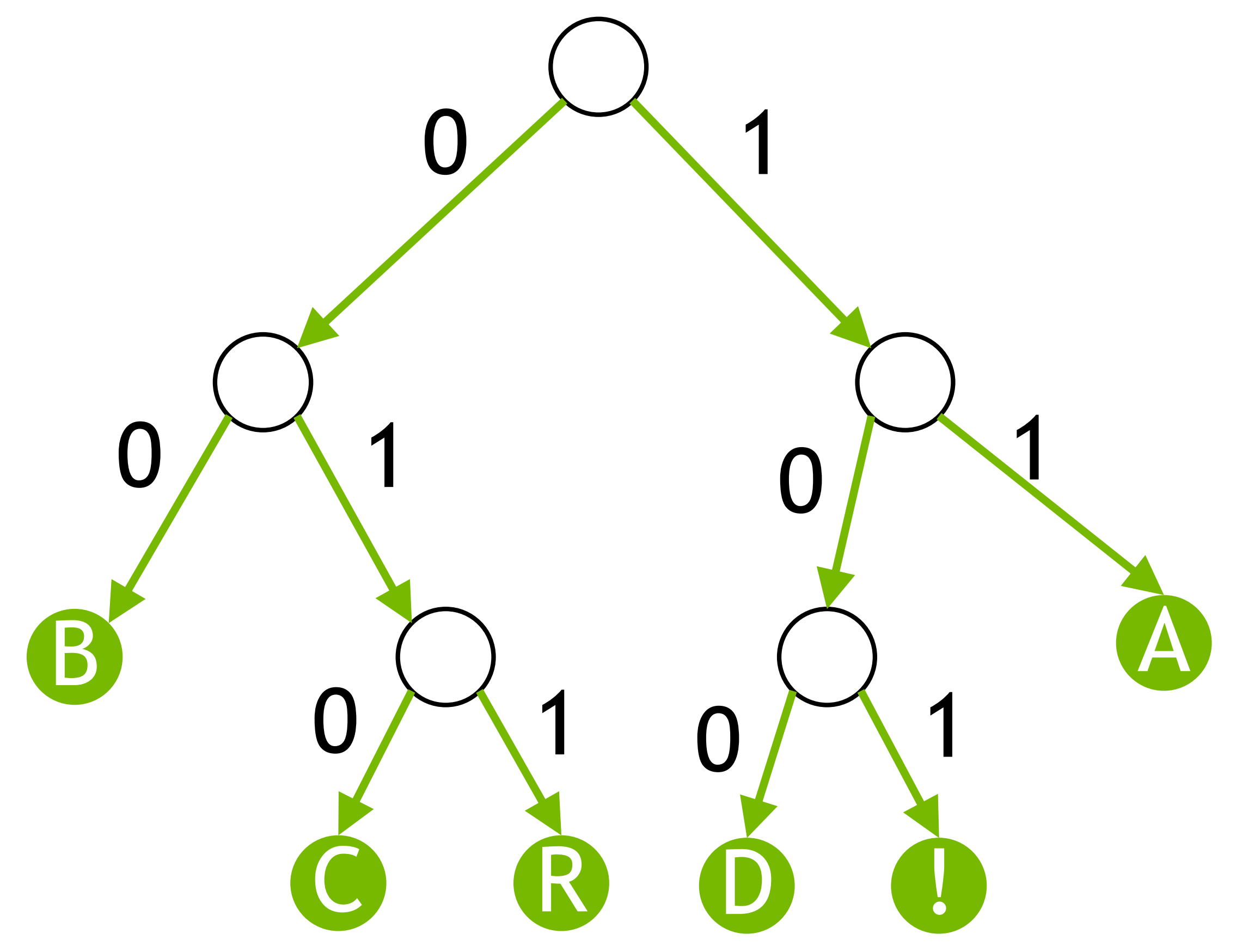
a	1	c	a	1	c	a	b	c	a	b	a	b	a	c	b	1	a	1	c	a	**Length**
0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	**Distance**

- A special case of LZ77 where only the cases of distance=1 are considered
- Worse compression ratio than LZ77, but suitable for the filtered pixels
- Better compression efficiency
- Default deflate strategy for PNG image in OpenCV
- Minimum length of matched sub-string is 3, so the literal will be outputted directly if length < 3

PNG Image and Its Encoding/Decoding Process

Huffman encoding

- A variable-length code table for encoding a source symbol
- label the edge to the left child as 0 and the edge to the right child as 1
- Only the leaf node store the source symbol
- traverse the Huffman tree to get to codeword table
- Two main features for Huffman tree
  - Frequency-related
  - Prefix-free



Key	A	B	C	D	R	!
Value	11	00	010	100	011	101

A A B C R D A B ! 72bits with ASCII table

11 11 00 010 011 100 11 00 101 22bits with above Huffman table

Key Optimizations in PNG Encoder

Run-length Encoding

CPU algorithm

a	a	a	a	a	c	c	c	c	b	b	b	b	a	c	b	b	b	a	a	a

a	4	c	3	b	a	c	b	b	b	a	a	a	Length
0	1	0	1	0	0	0	0	0	0	0	0	0	Distance

```
for (;;) {
  /* See how many times the previous byte repeats */
  int match_length = 0;
  int scan = cursor - 1;
  int prev = *scan;
  /* If there are three continuous same */
  if (prev == *++scan && prev == *++scan
    && prev == *++scan) {
    int end = cursor + MAX_MATCH;
    do {
    } while (prev == *++scan && scan < strend);
    /* Record the repeat times */
    match_length = MAX_MATCH - (end - scan);
  }
```

```
  /* Emit match if have run of MIN_MATCH or longer,
else emit literal */
  if (match_length >= MIN_MATCH) {
     emit_match_pair(1, match_length - MIN_MATCH);
     /* Move forward the cursor and sliding window*/
     cursor += match_length;
     match_length = 0;
  } else {
     /* No match, output the symbol*/
     emit_literal(*++prev);
     /* Move forward the cursor and sliding window*/
     s->strstart++;
  }
}
```

The algorithm is inherently serial
and has strong data dependency.

Run-length Encoding

Problem description

a	a	a	a	a	c	c	c	c	b	b	b	b	a	c	b	b	b	a	a	a

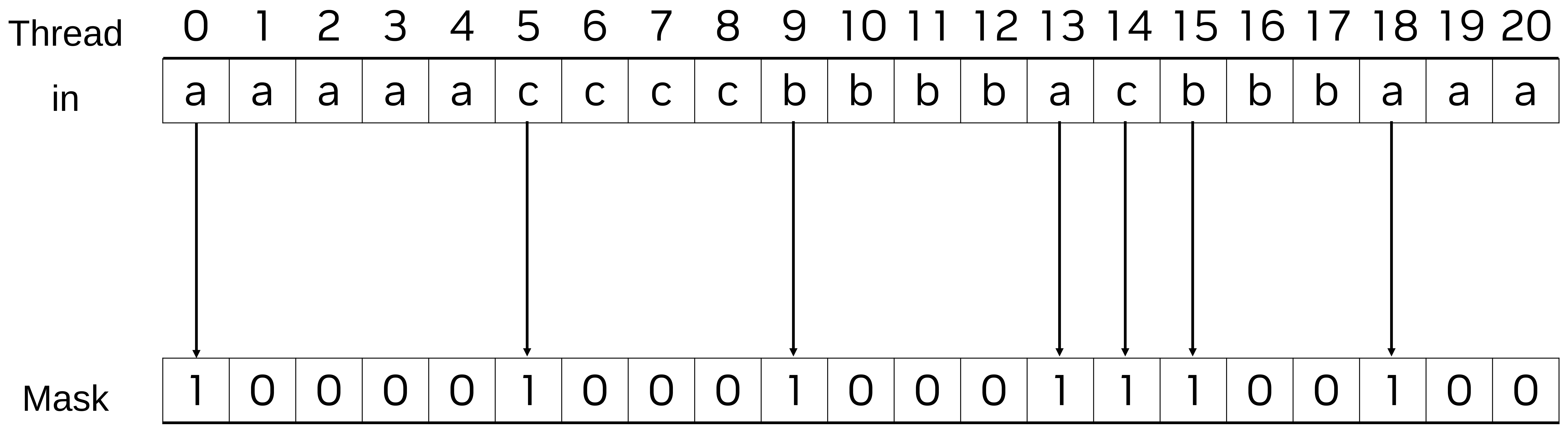
a	4	c	3	b	3	a	c	b	b	b	a	a	a	Length
0	1	0	1	0	1	0	0	0	0	0	0	0	0	Distance

- The input is a continuous 1-D buffer
- Adjacent elements are coherent
- Match boundary: (3, 258)
- Results should be bit-wise equivalent with the CPU algorithm
- Output position for each thread is unpredictable
- Straight divide-and-conquer algorithm doesn't work in this case

Run-length Encoding

Mask

- The first element will always be set to 1
- Each thread compare the current with the previous element for equality, mask will be set to 1 if not equal
- The mask encodes the beginning of all the runs



Thread	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
in	a	a	a	a	a	c	c	c	c	b	b	b	b	a	c	b	b	b	a	a	a
Mask	1	0	0	0	0	1	0	0	0	1	0	0	0	1	1	1	0	0	1	0	0

Run-length Encoding

Prefix sum

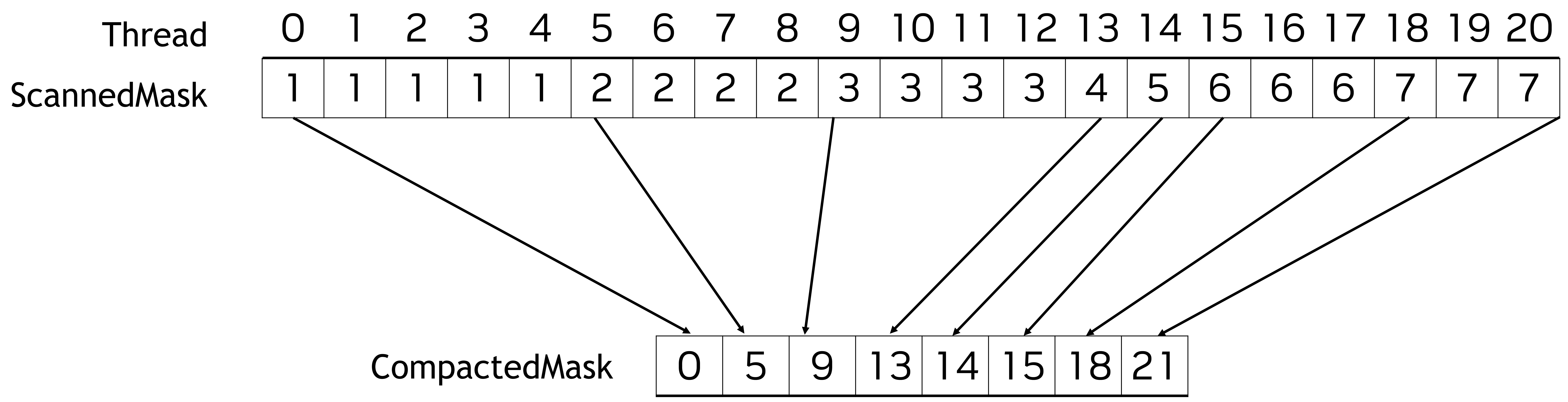
- Prefix sum array is another array of the same size, such that the value of prefixSum[i] is arr[0] + arr[1] + arr[2] … arr[i]
- The scannedMask encodes the output location of all the compressed pairs
- The CUDA implementation of prefix-sum(Scan) algorithm could be found in the CUB library

Thread	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
in	a	a	a	a	a	c	c	c	c	b	b	b	b	a	c	b	b	b	a	a	a
Mask	1	0	0	0	0	1	0	0	0	1	0	0	0	1	1	1	0	0	1	0	0
ScannedMask	1	1	1	1	1	2	2	2	2	3	3	3	3	4	5	6	6	6	7	7	7

Run-length Encoding

Accumulated Runs length

- We get the output location of the output pairs, but the run lengths are never touched
- The first element will always be set to 0
- If the current and the previous elements are not equal, output the thread index at scannedMask[i]-1
- The last element will be set to the total size of input array



Run-length Encoding

Overall GPU implementation

Run-length
5 4 4 1 1 3 3
CompactedMask
0 5 9 13 14 15 18 21
In
a a a a a c c c c b b b b a c b b b a a a
Symbol
a c b a c b a
Transform kernel
Transform kernel
Out
Length a 4 c 3 b 3 a c b b b a a a
Distance 0 1 0 1 0 1 0 0 0 0 0 0 0 0

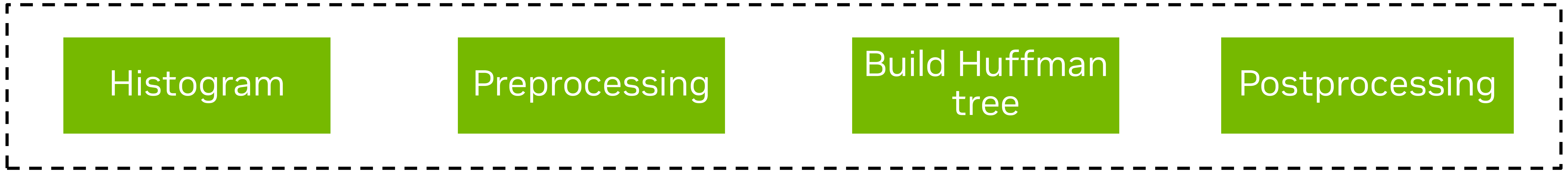
Huffman encoding and Flush to file

Flow of GPU implementation

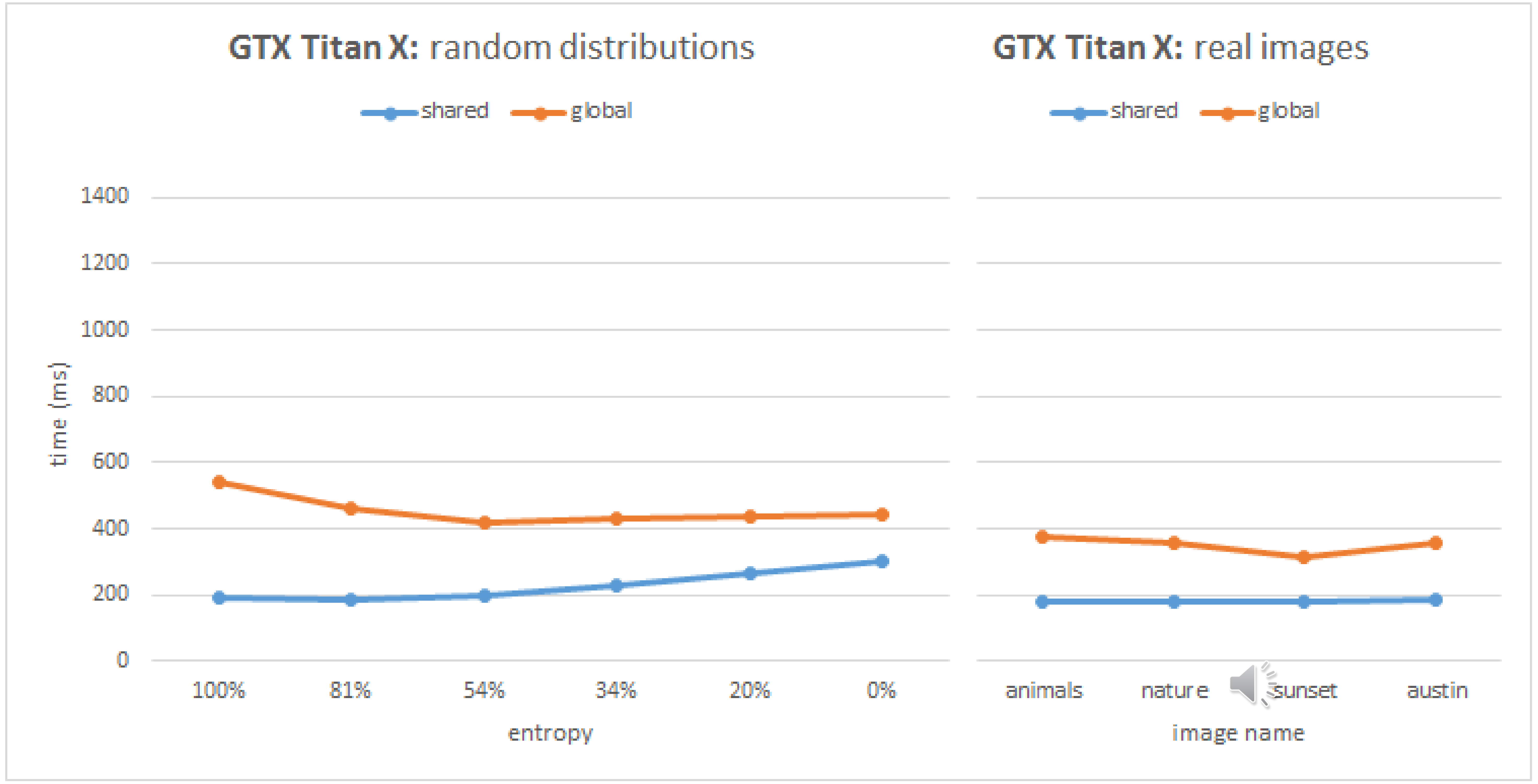
Huffman encoding
Histogram
Preprocessing
Build Huffman tree
Postprocessing
Literal/Length
Source array
Codeword table
Codeword length
Distance
Source array
Codeword table
Codeword length
Code length
Codeword table
Codeword length
Compressed data flow
Compressed codeword length
Stage 1
Stage 2
Stage 3
HLIT
HDIST
HCLEN
code lengths
coded code lengths for literal/length tree
coded code lengths for distance tree
coded data
256
One specific DEFLATE block

Huffman encoding

Histogram



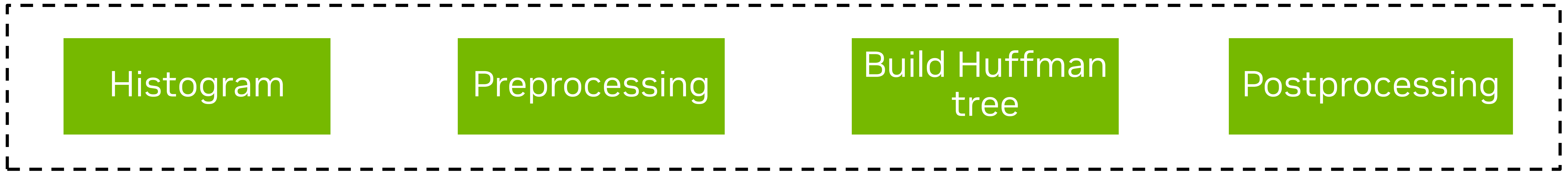
- Hardware support for shared memory atomics since Maxwell
- Shared atomics histogram implementation show 2x speedup
- Reference: Shared Atomics



Performance of histogram algorithm comparing global memory atomics to shared memory atomics on a Maxwell architecture NVIDIA GeForce GTX TITAN X GPU

Huffman encoding

Preprocessing



- For literal/length tree, total number of all symbols is 286 (256 literal symbols + 1 end symbol + 31 length symbols);
- Before building Huffman tree, the last symbol with non-zero frequency should be determined for efficiency;

- The process could be abstracted to such an algorithm:
- Given an integer array, find the index with the last non-zero value;
- For example, the input array is 1, 2, 0, 6, 5, 0, 12, 0, 4, 0, 0, 0, the output should be 8(value = 4);

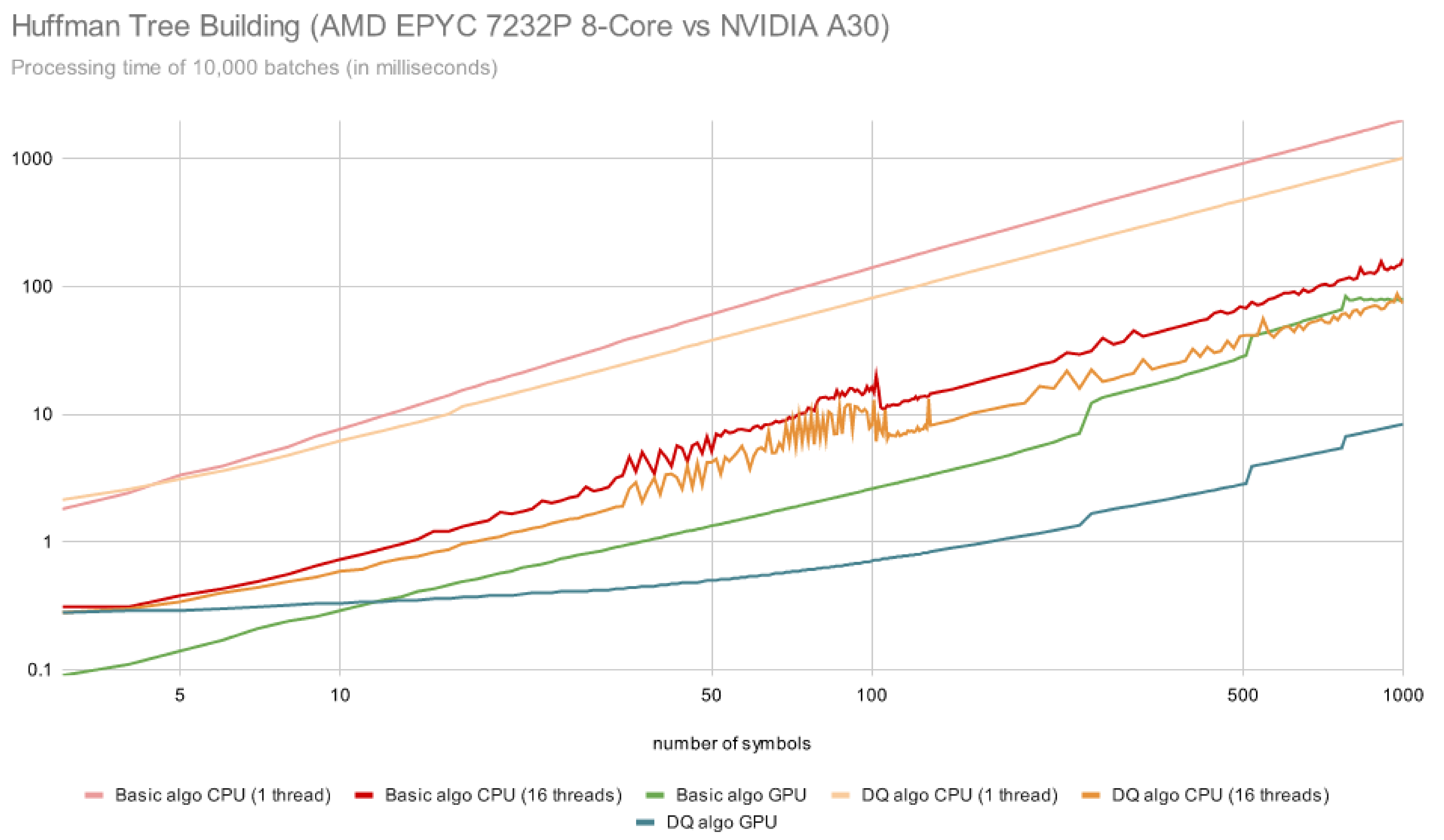
- GPU-accelerated solution:
- Declare a mask array with the same length, set the value to thread index if the input value is non-zero, or set it to 0;
- Block-reduce to get the maximum value of the mask array;

Huffman encoding

Build Huffman tree

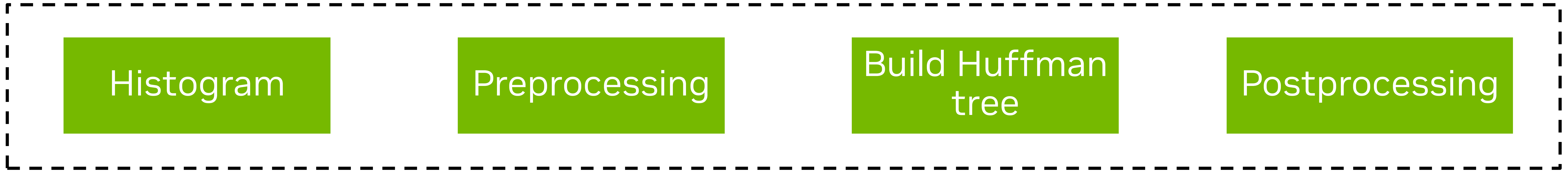
Histogram | Preprocessing | Build Huffman tree | Postprocessing

- Two implementations: Priority queue(Basic) and Two-queues(DQ)
- DQ improves the perf. no matter for CPU or GPU
- DQ can’t produce the same results with baseline
  - Different strategies for symbols with same frequency
- Basic algo is chosen as the GPU implementation of building Huffman tree in encoder

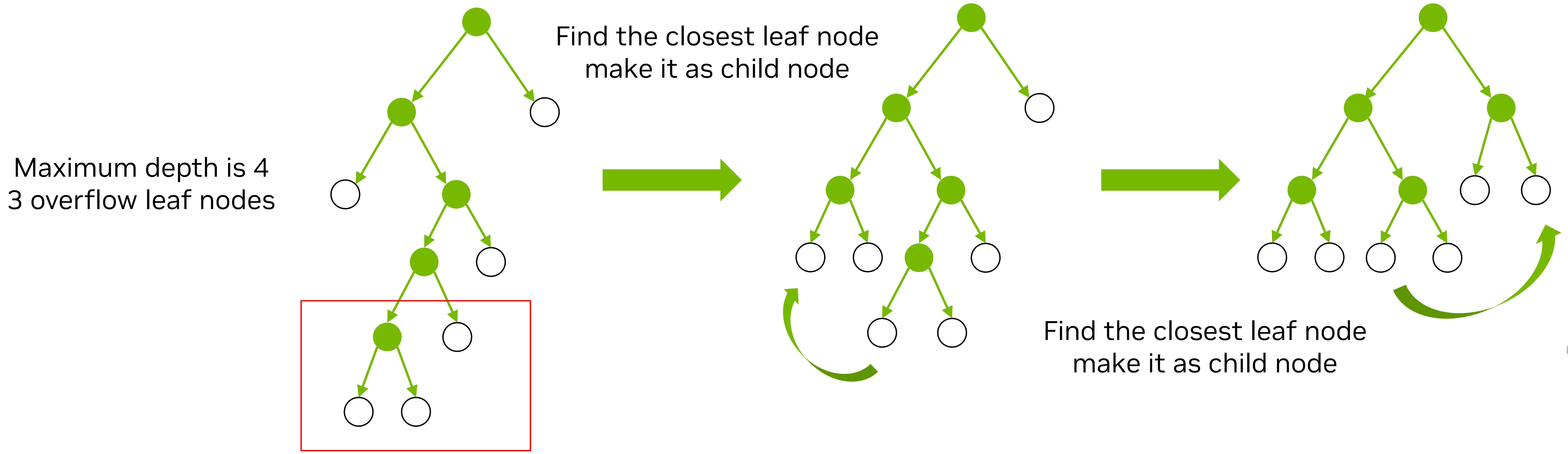


Huffman encoding

Postprocessing



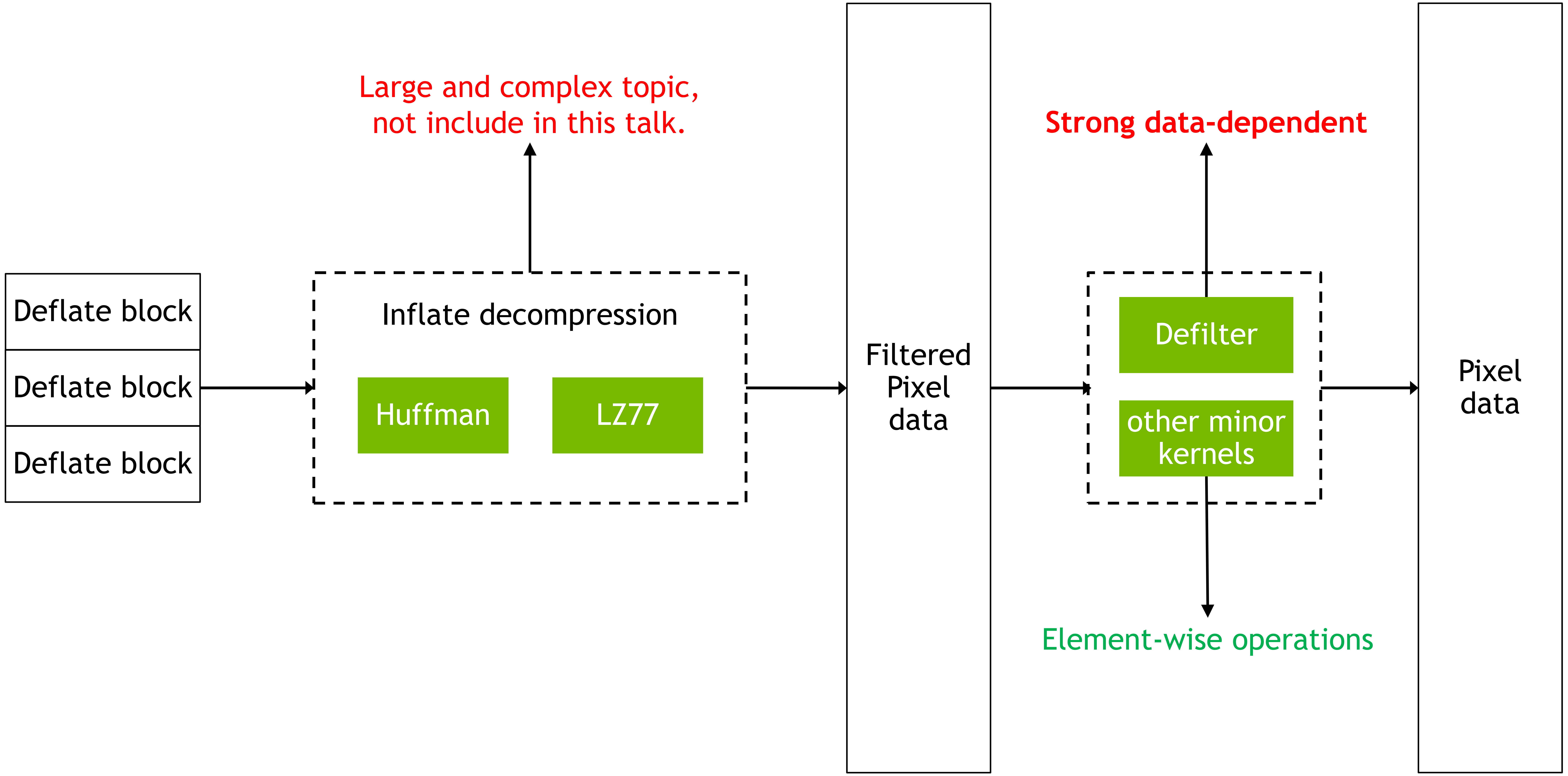
- The maximum depth of Huffman tree is limited in 7 for code length tree, and 15 for literal/length tree and distance tree.
- The generated Huffman tree may need to be pruned.


Maximum depth is 4
3 overflow leaf nodes
Find the closest leaf node make it as child node
Find the closest leaf node make it as child node

Key Optimizations in PNG Decoder

Key Optimizations in PNG Decoder

Decoding process



Key Optimizations of De-Filter in PNG Decoder

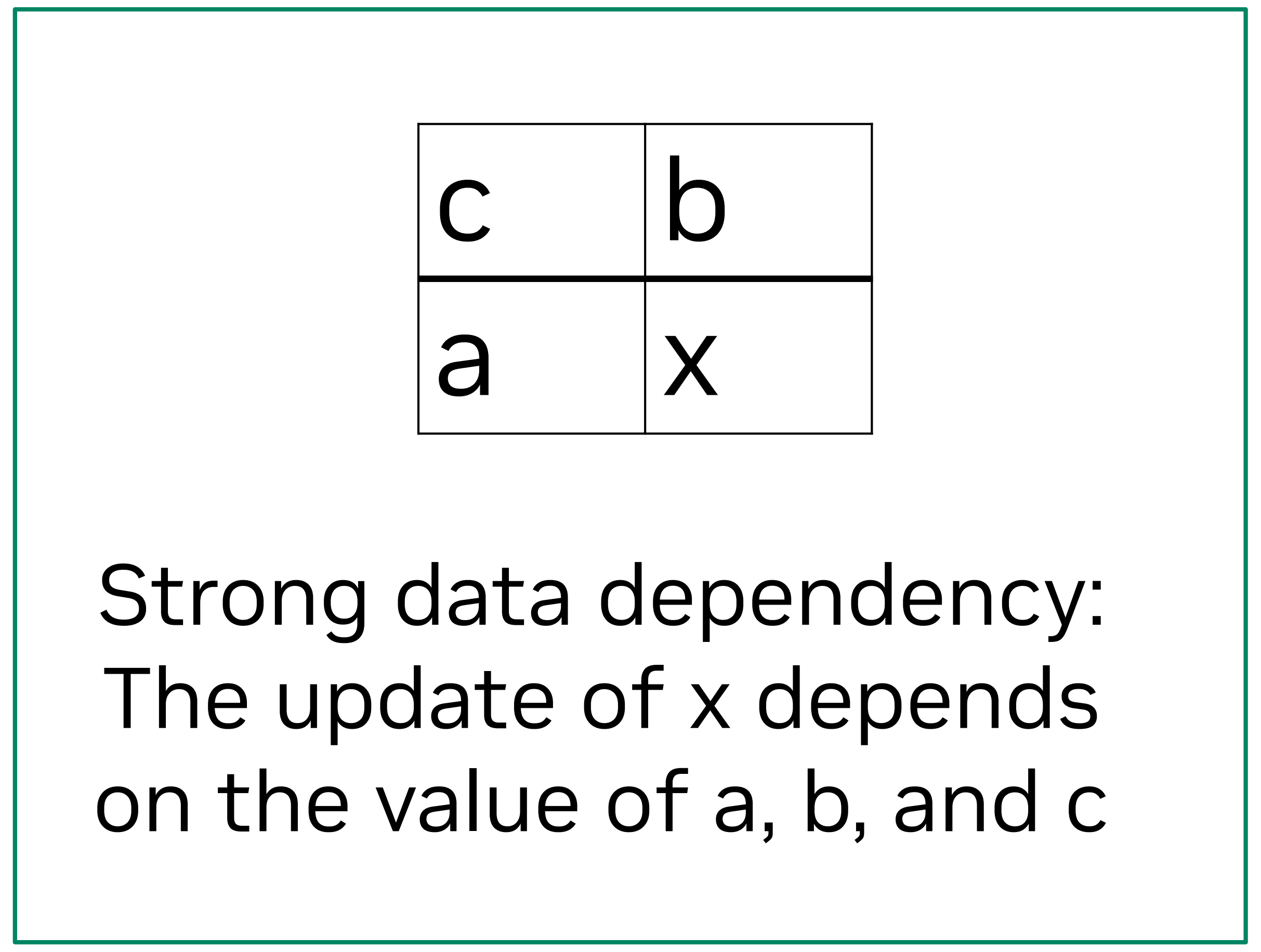
outline

- Wave solver
- Tile wave solver
  - Divide image into tiles
  - Register reuse
- Lock-based dispatch
- Hide the Latency of data access

Wave solver

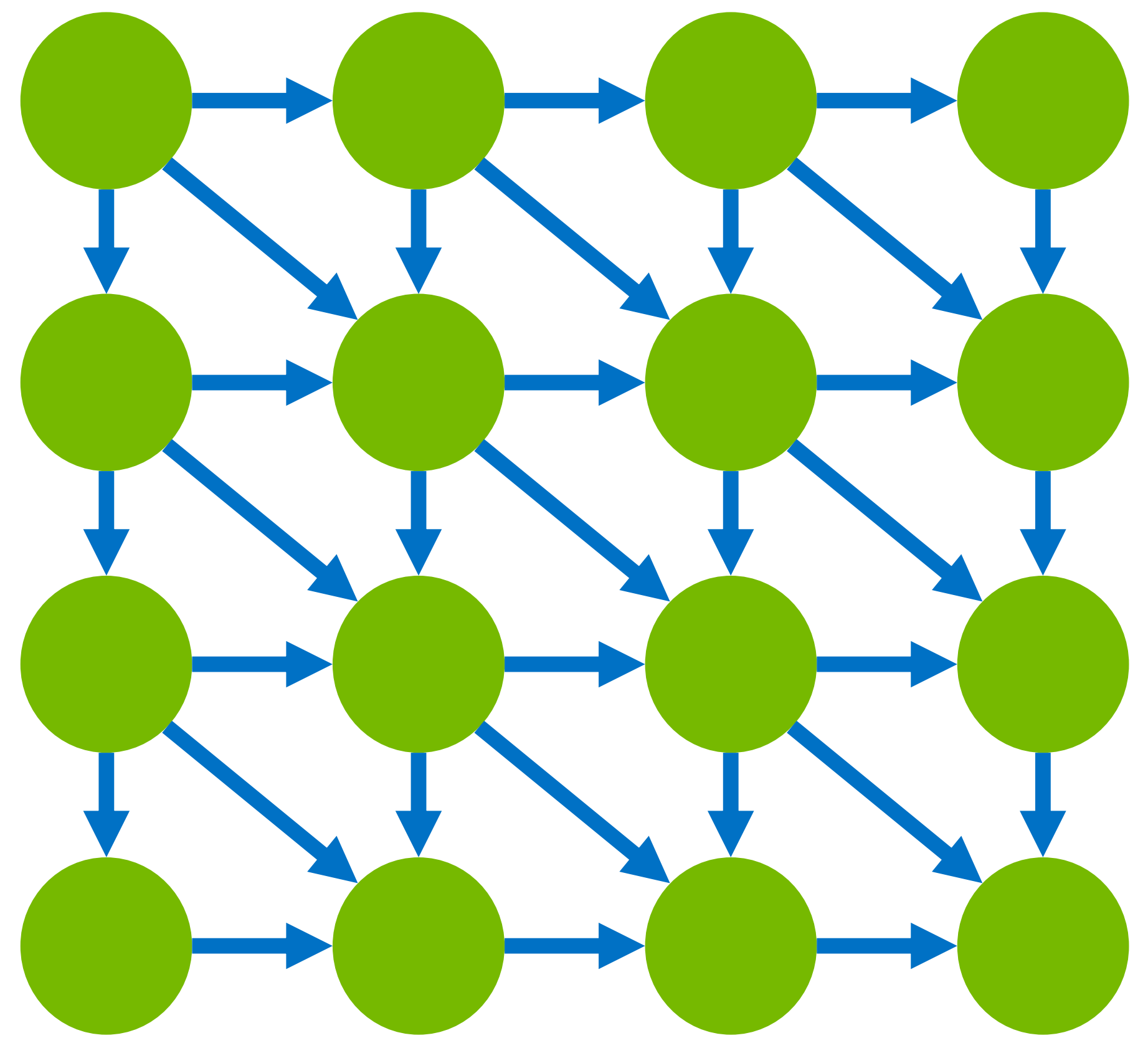
How to parallel the de-filter process

- **Parallelize the de-filter with the "wave solver" strategy.**
  - The update of current pixel depends on the left, up, upper left pixels.
  - The pixels at diagonals can be processed in parallel, which we called "wave solver"

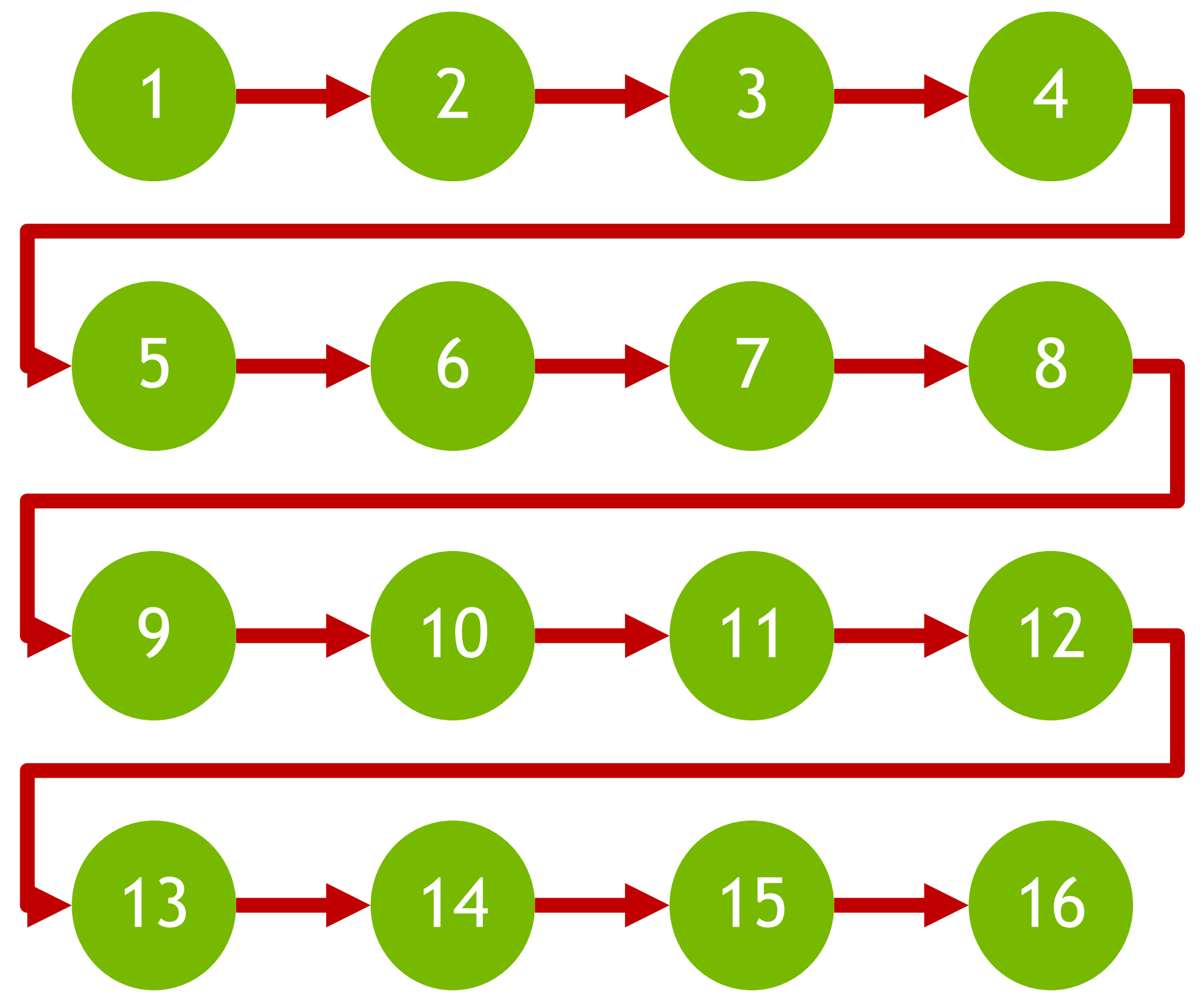


c	b
a	x

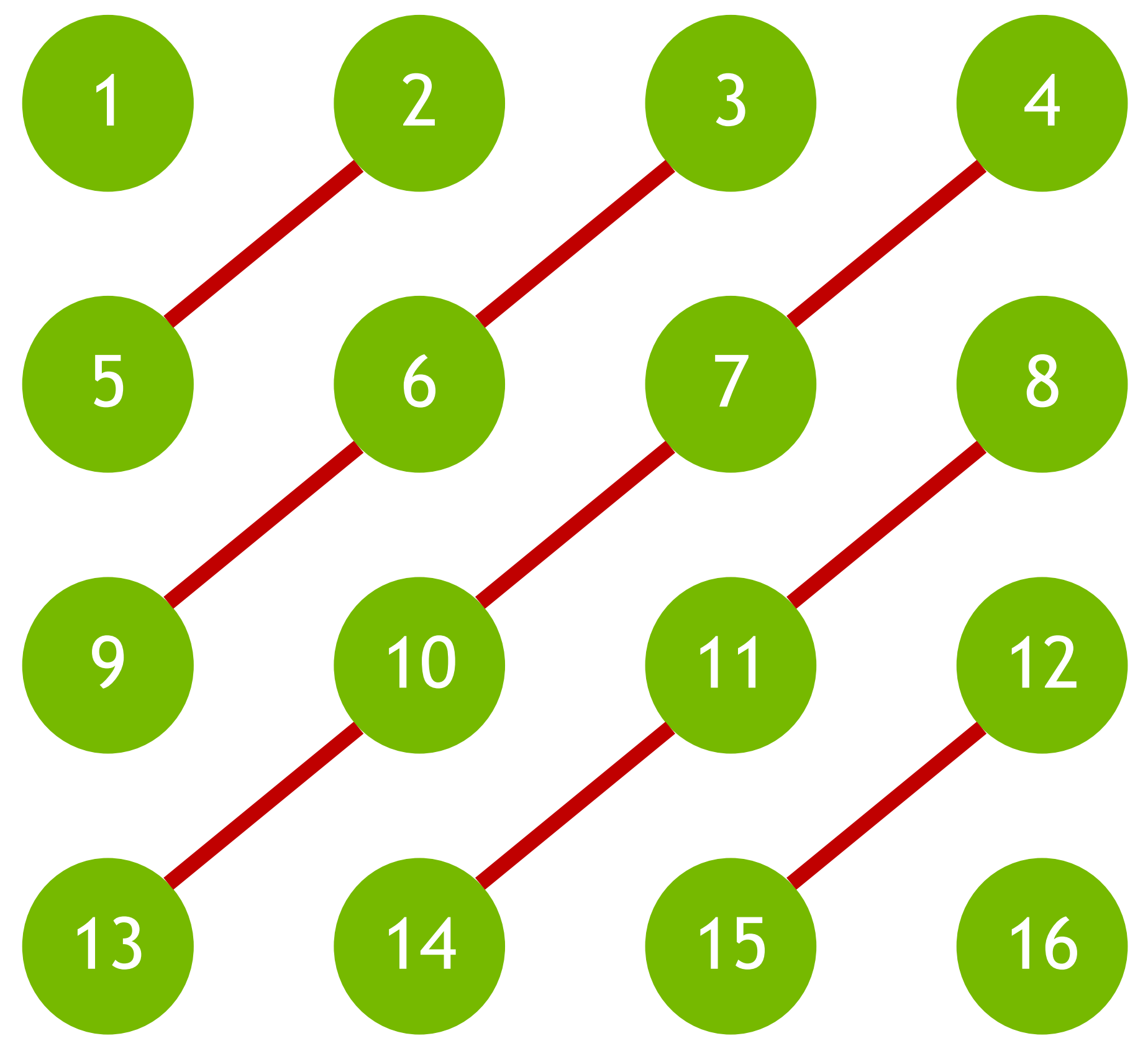
Strong data dependency: The update of x depends on the value of a, b, and c



The dependencies in 4 × 4 pixels matrix



All pixels are processed sequentially in CPU.



Pixels in the diagonal can be parallel. Synchronization is required between diagonals

- The order of updates in CPU

  1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11 -> 12 -> 13 -> 14 -> 15 -> 16

- The order of updates in wave solver

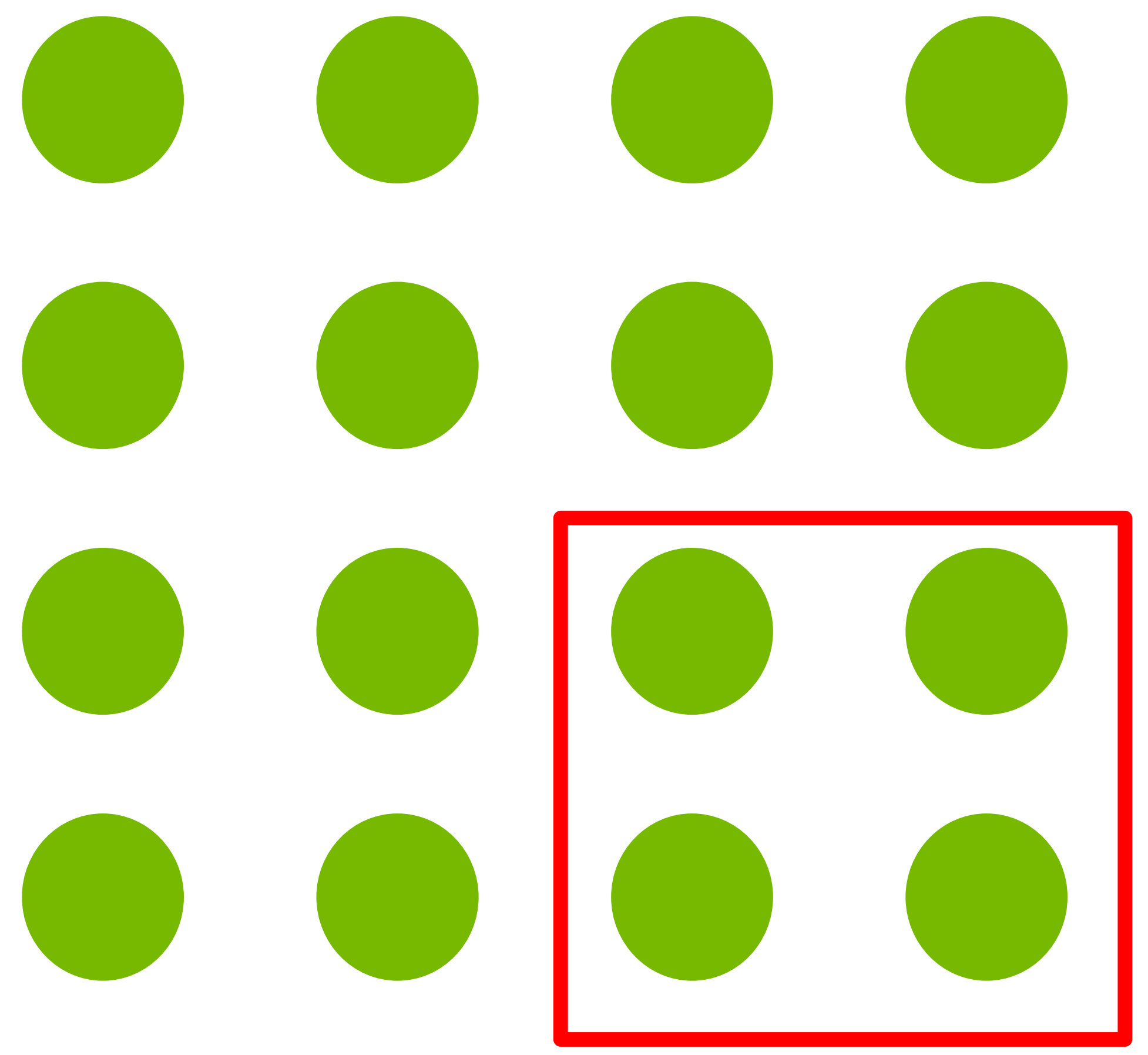
  (1) -> (2,5) -> (3,6,9) -> (4,7,10,13) -> (8,11,14) -> (12,15) -> (16)

  -The pixels in "()" can be processed in parallel.

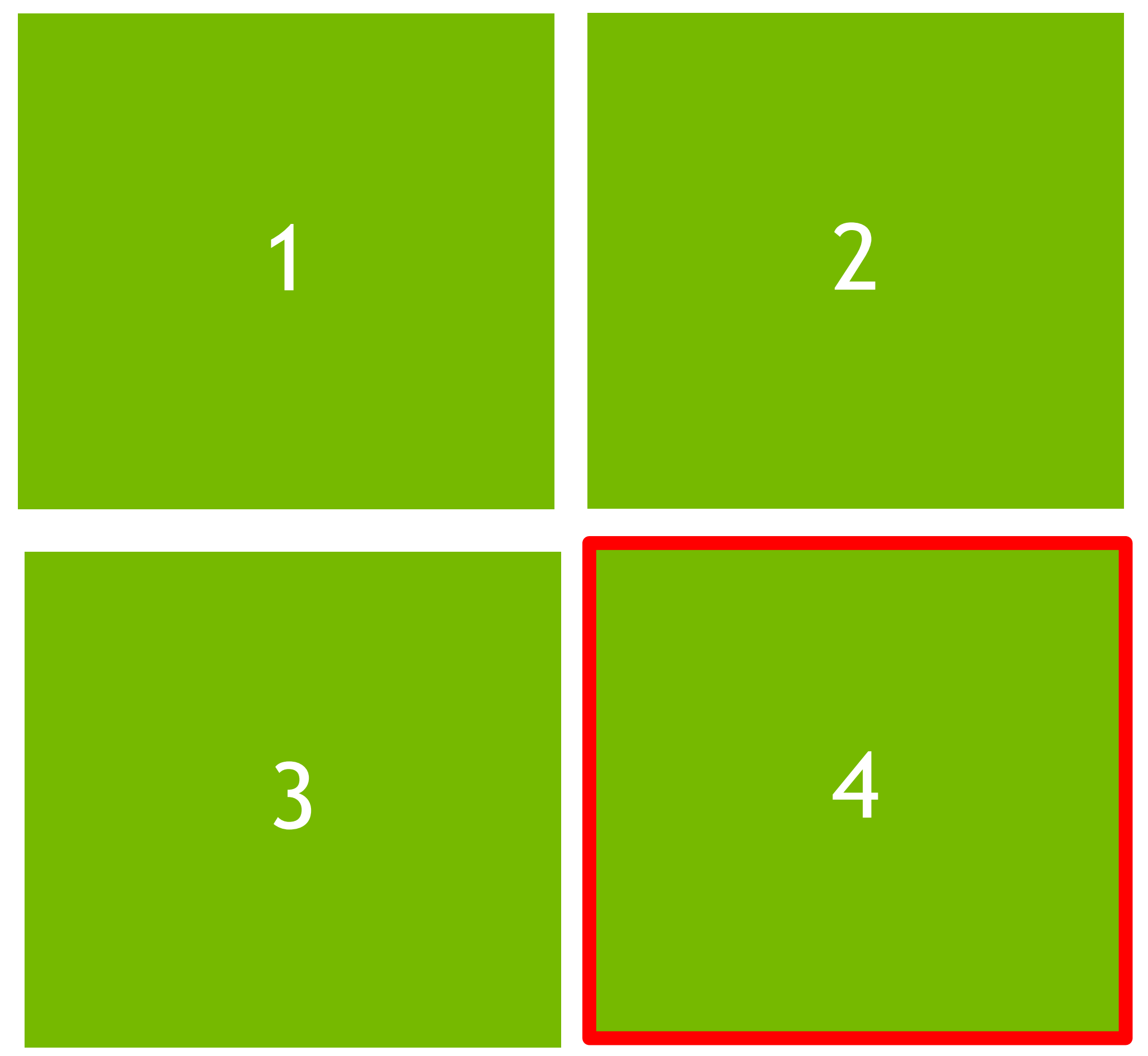
Tile wave solver

How to avoid uncoalesced access

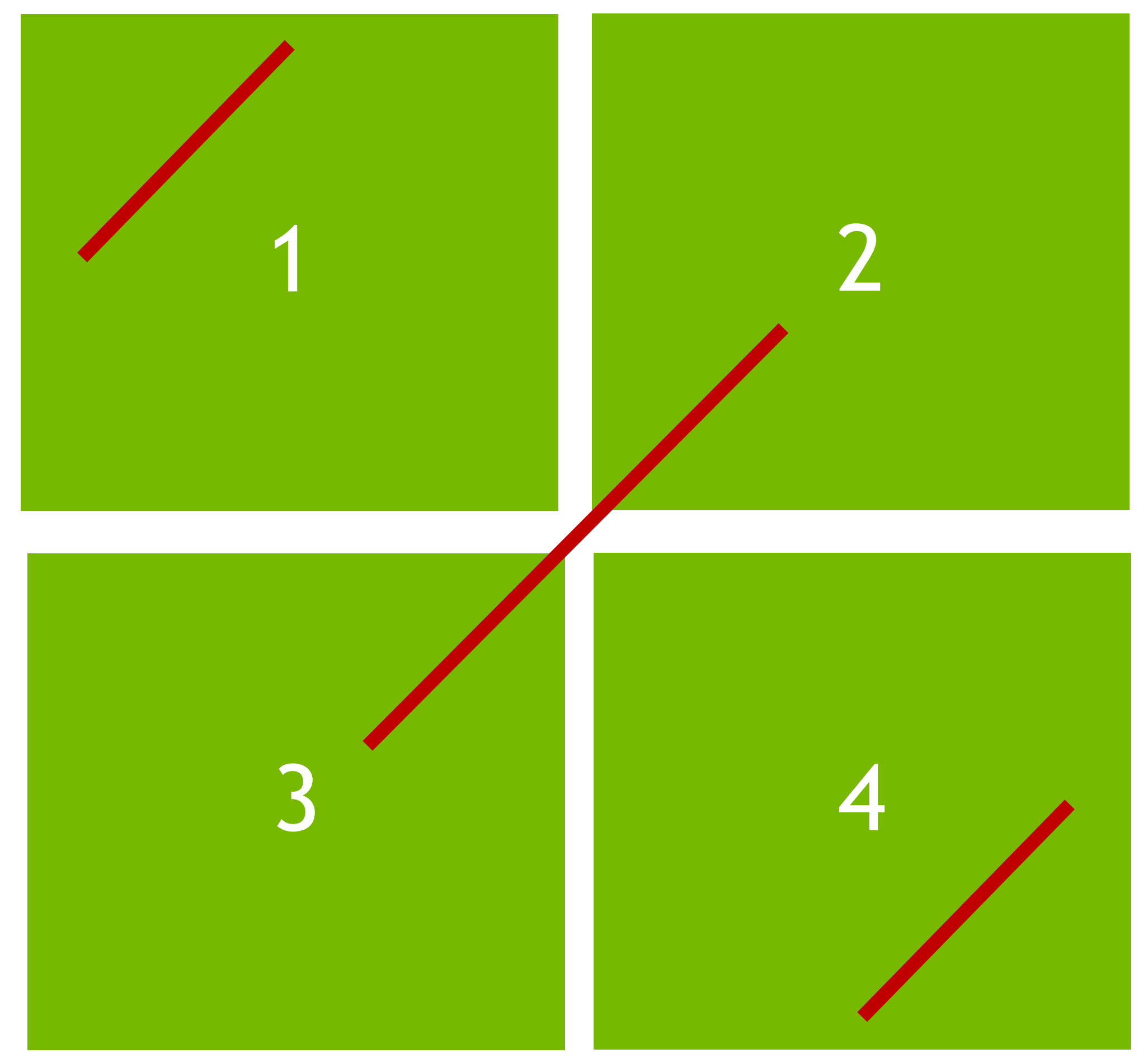
- **Tile wave solver**
  - The whole image can not be put in shared memory, the diagonal update will cause uncoalesced access to global memory.
  - Divide the pixel matrix into several tiles, each block handle one at "tile diagonal".
  - **The synchronization at block-level is required at tile wave solver**



For example, each 2×2 submatrix is a tile



The 4×4 pixel matrix can be seen as a 2×2 tile matrix



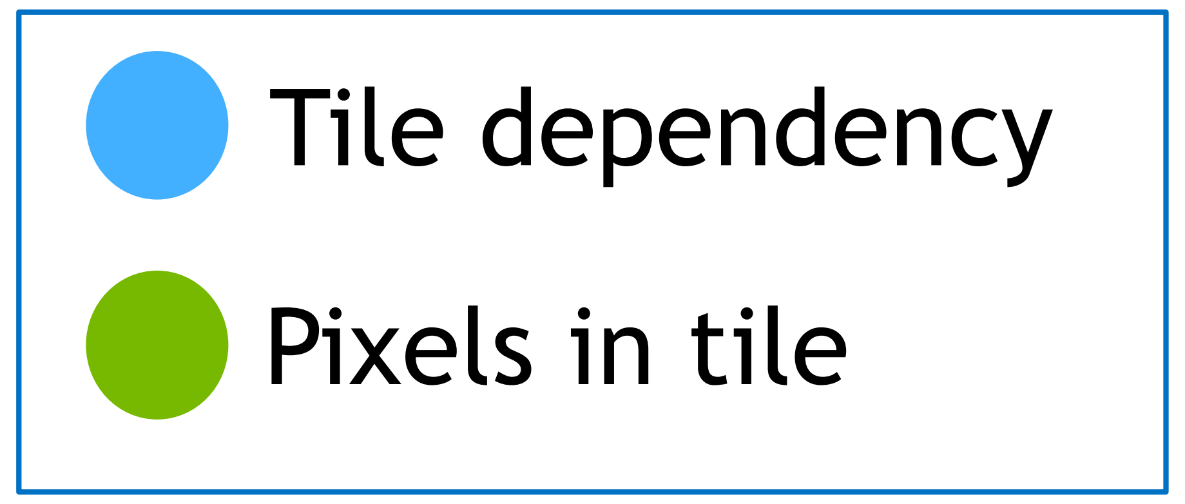
3 diagonals
2 synchronizations

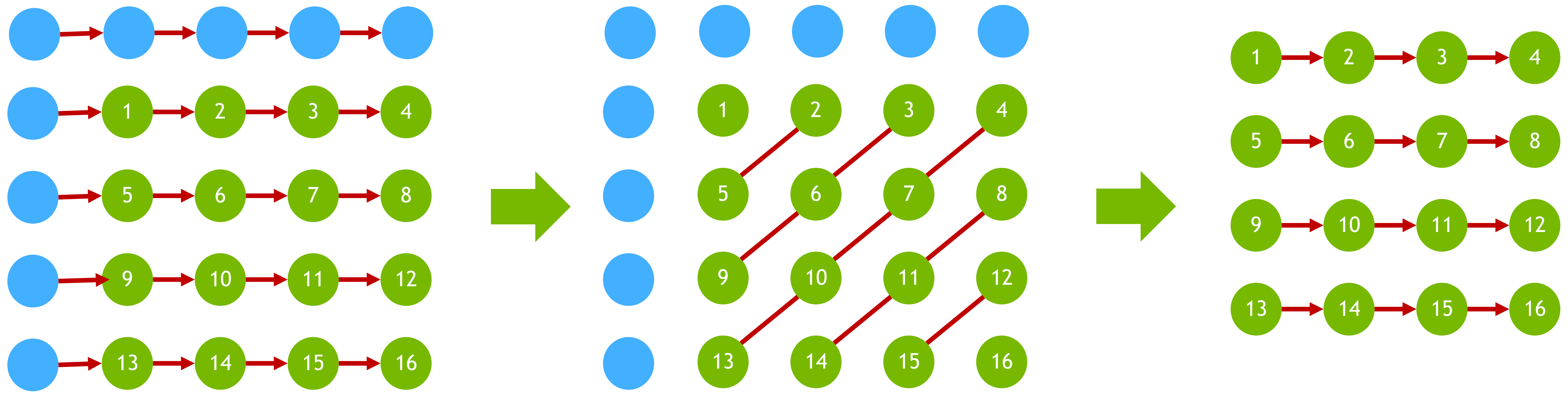
- At the sample, there are 4 tiles, 3 tile diagonals.
  - At each tile diagonal , each block handle one tile
  - This case need 2 blocks
- Block-level synchronization between different diagonals

Tile wave solver

Data load/store at each tile

- **Pixel update inside tile (in block)**
  - Load the whole tile in shared memory (coalesced access)
  - Update pixels diagonal by diagonal
  - Write the result to global memory (coalesced access)
  - For a N*N tile, (N+1)*(N+1) tile is loaded to satisfy the dependency





For 4 × 4 matrix, 5 × 5 pixels is loaded to shared memory row by row (**Vectorized coalesced access**)

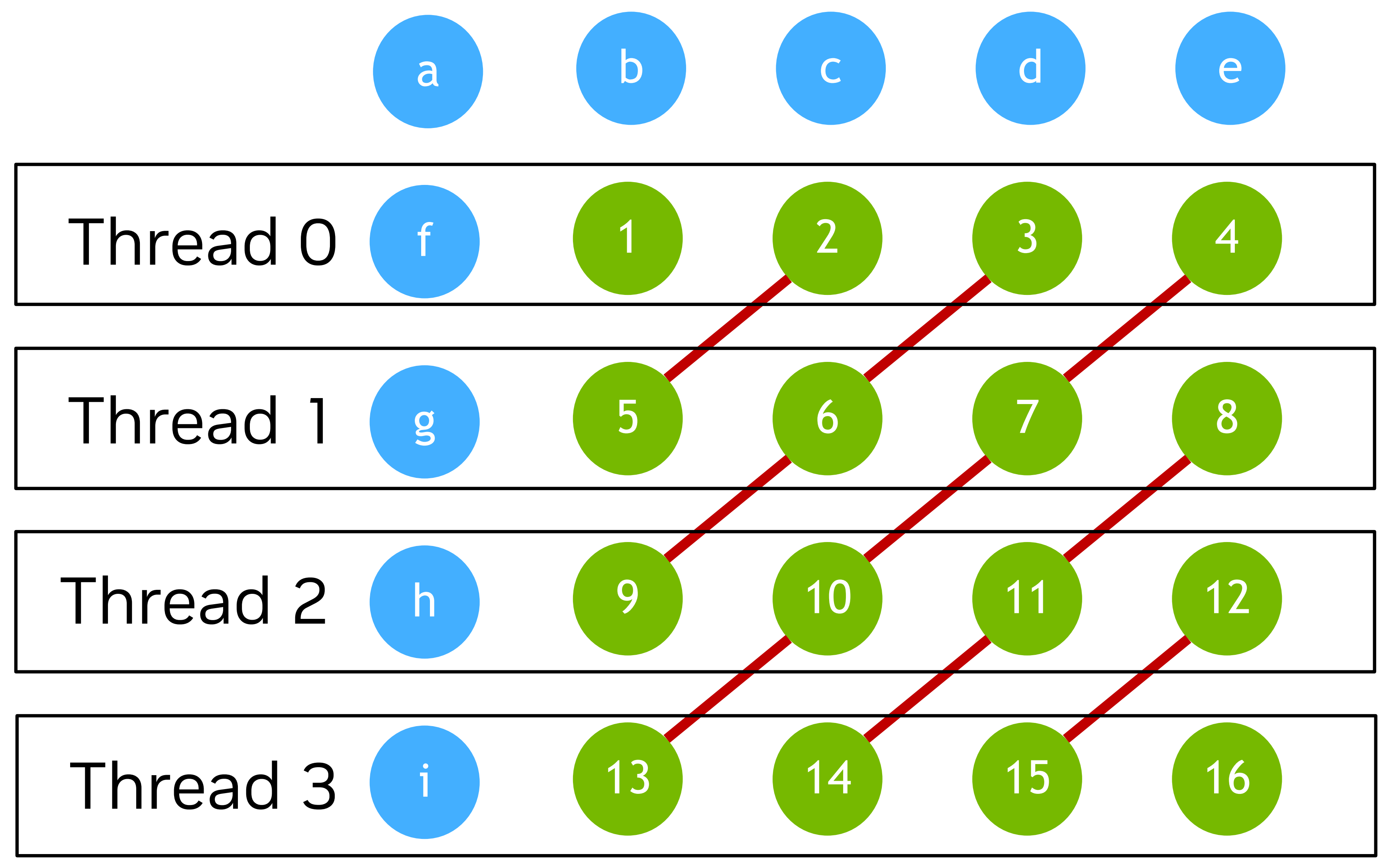
Update pixels on diagonals in shared memory

4 × 4 matrix is written to global memory (**Vectorized coalesced access**)

Tile wave solver

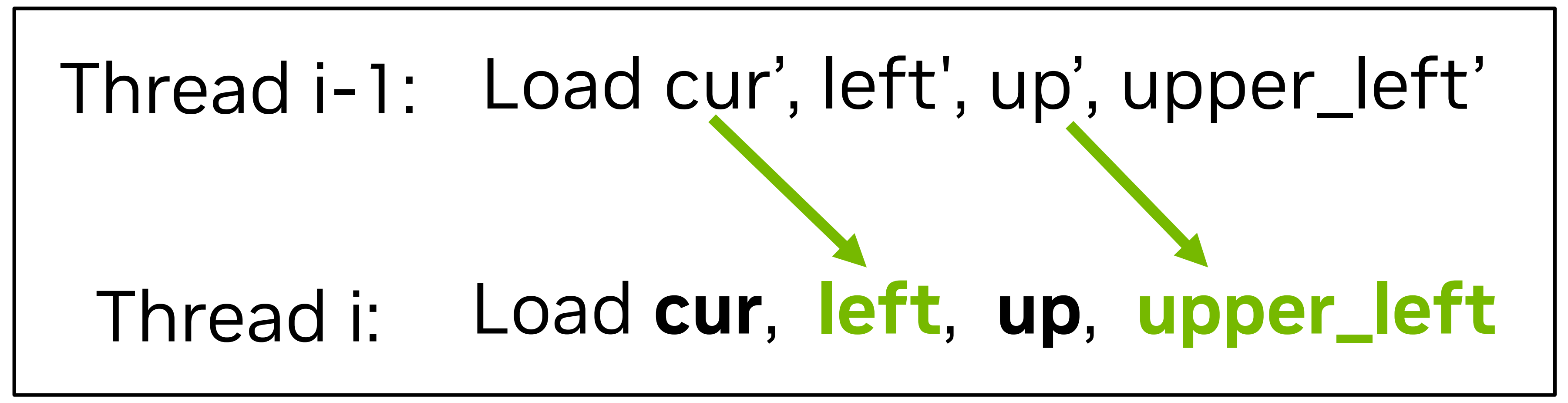
Register reuse

- **Pixel update inside tile (in block)**
  - Each thread handle one row at a tile
  - The "left" of current pixel is the "current" of the last iteration
  - The "upper left" of current pixel is the "up" of the last iteration



Data load in thread 0:

	Pixel id to load
Iteration 0	Load 1,f,b,a
Iteration 1	Load 2,1,c,b
Iteration 2	Load 3,2,d,c
Iteration 3	Load 4,3,e,d
Iteration 4+	None

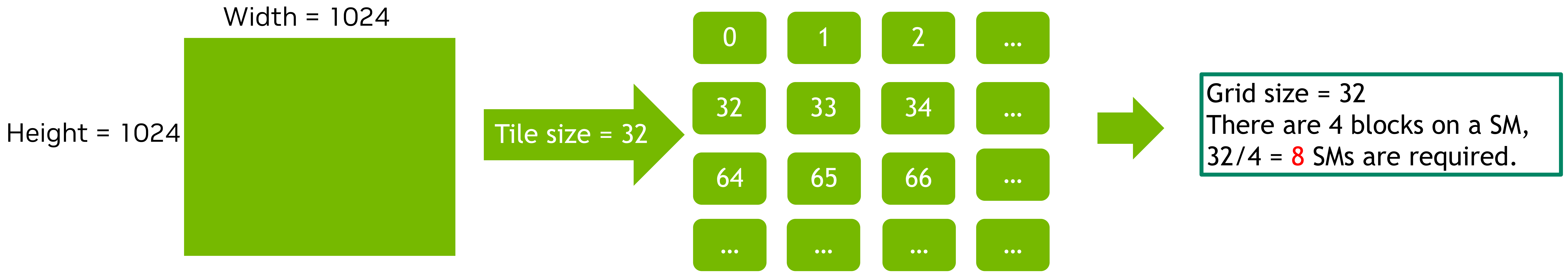

Thread i-1: Load cur', left', up', upper_left'
Thread i: Load **cur**, **left**, **up**, **upper_left**

In single thread i, the "**cur**" and the "**up**" should be loaded from global memory. The "**left**" and "**upper_left**" should be loaded from register at last register.

Tile wave solver

The SM occupancy of tile wave solver

- **Tile update and synchronization**
  - Grid size := the number of blocks
  - Large tile => can not be put into shared memory
  - Small tile => low parallelism in block



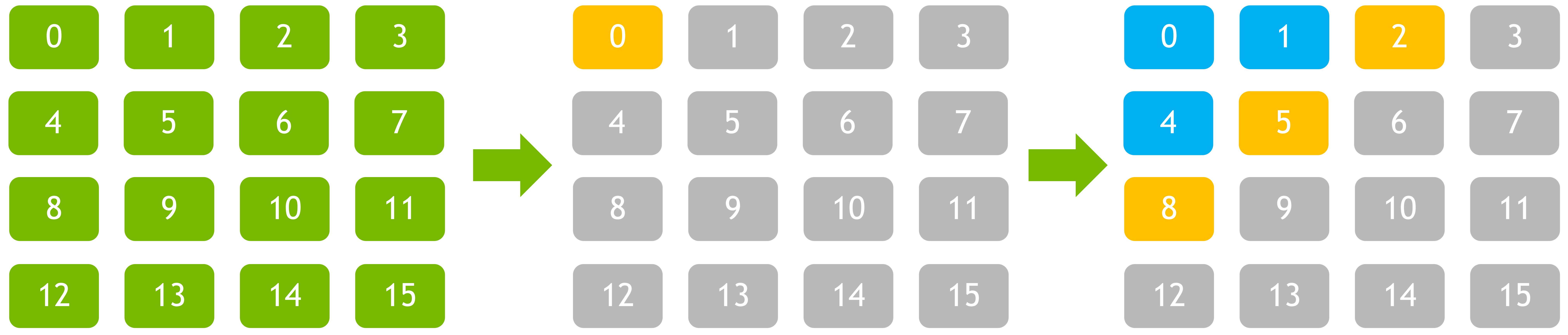
Grid size = 32
There are 4 blocks on a SM,
32/4 = 8 SMs are required.

- The occupancy of tile wave solver need to be improved.
  - A $1k \times 1k$ PNG image requires 8 SMs to decode.
  - The A100-80GB has 108 SMs (occ = 8/108 = 7.14%), A30 has 56 SMs(occ = 8/56 = 14.29%).
- There are many block-level synchronizations
  - Number of diagonals = 63
  - Number of block-level synchronizations = 62
- New lock-based dispatch strategy is proposed

Lock-based dispatch

How to use more blocks in tile wave solver

- **Tile update and synchronization**
  - Lock-based dispatch
  - Each tile is processed on a block
  - Only active block should be processed



Divide the whole image into several tiles. (4×4 tiles in this case)

All tile is blocked at initialization except tile 0. The tile 0 is active.

Blocked tiles can be active at some conditions

🟨	Active tile
⬜	Blocked tile
🟦	Finished tile

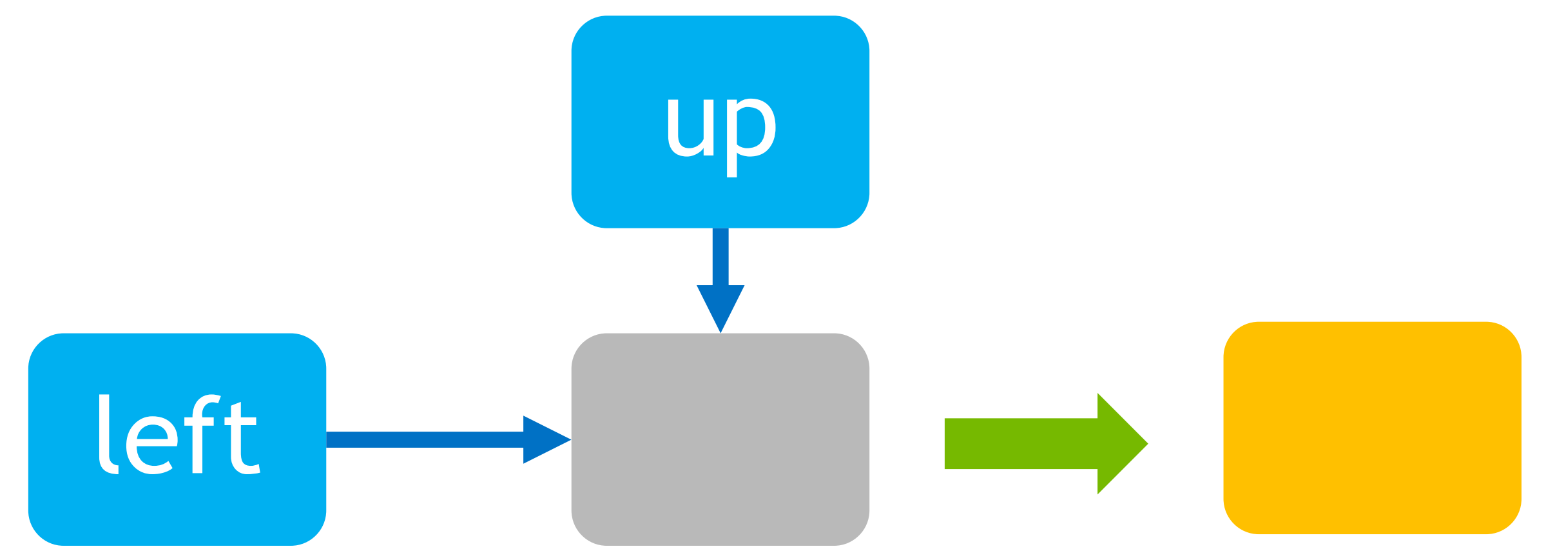
Lock-based dispatch

How to update the state of lock

1. Blocked -> Active

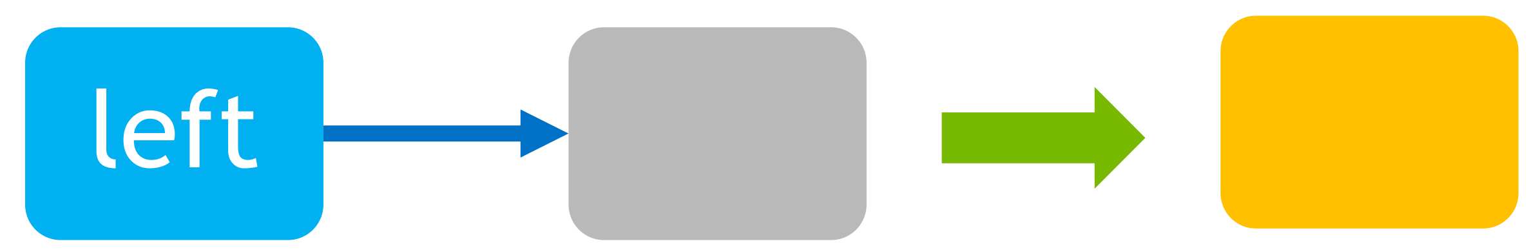
T := de-filter type of the first line in tile

(1) T in {2,3,4}



The lock will be released when the left, up tile have finished their work.

(2) T in {0,1}



The lock will be released when the left tile has finished the work.

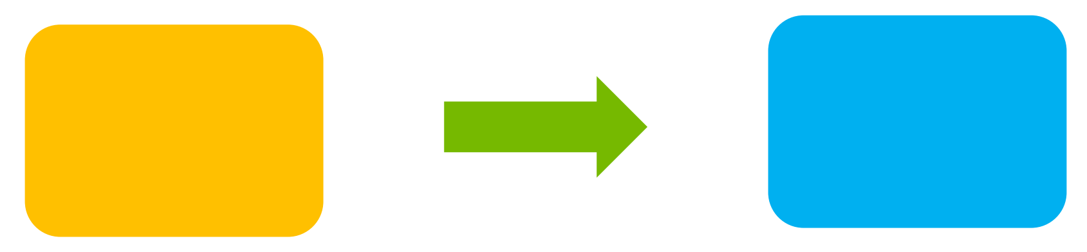
Type byte	Filter name	Inverse function
0	None	x'=x
1	Sub	x'=x+a'
2	Up	x'=x+b'
3	Average	x'=x+(a'+b')/2
4	Paeth	x'=Paeth'(x, a', b', c')

- The x, a, b, c are corresponding to current , left, up, upper left pixel
- x' and x denote the latest and original value respectively, and the same as a'/a, b'/b, c'/c

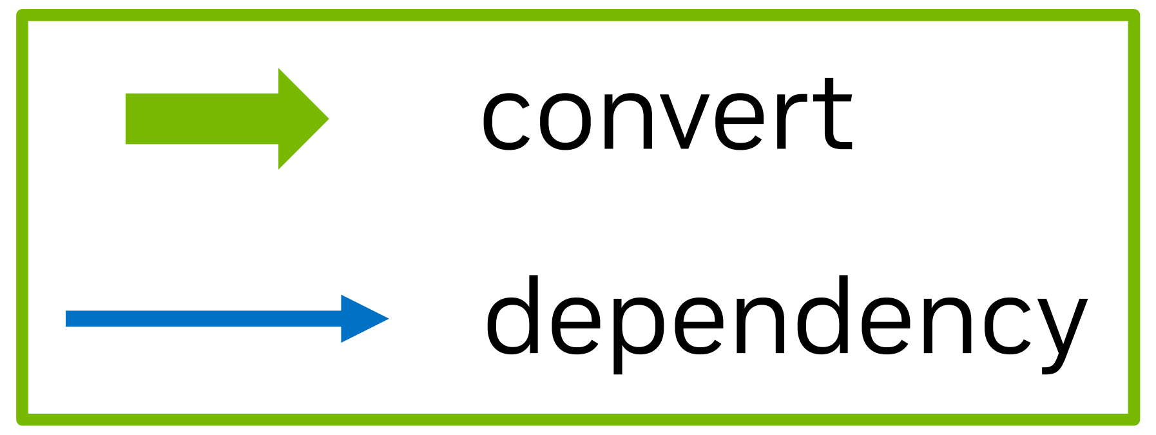
c	b
a	x

- **Tile update and synchronization**
  - Release lock at special case
  - Update rules
    - Active -> Finished
    - Blocked -> Active

2. Active -> Finished



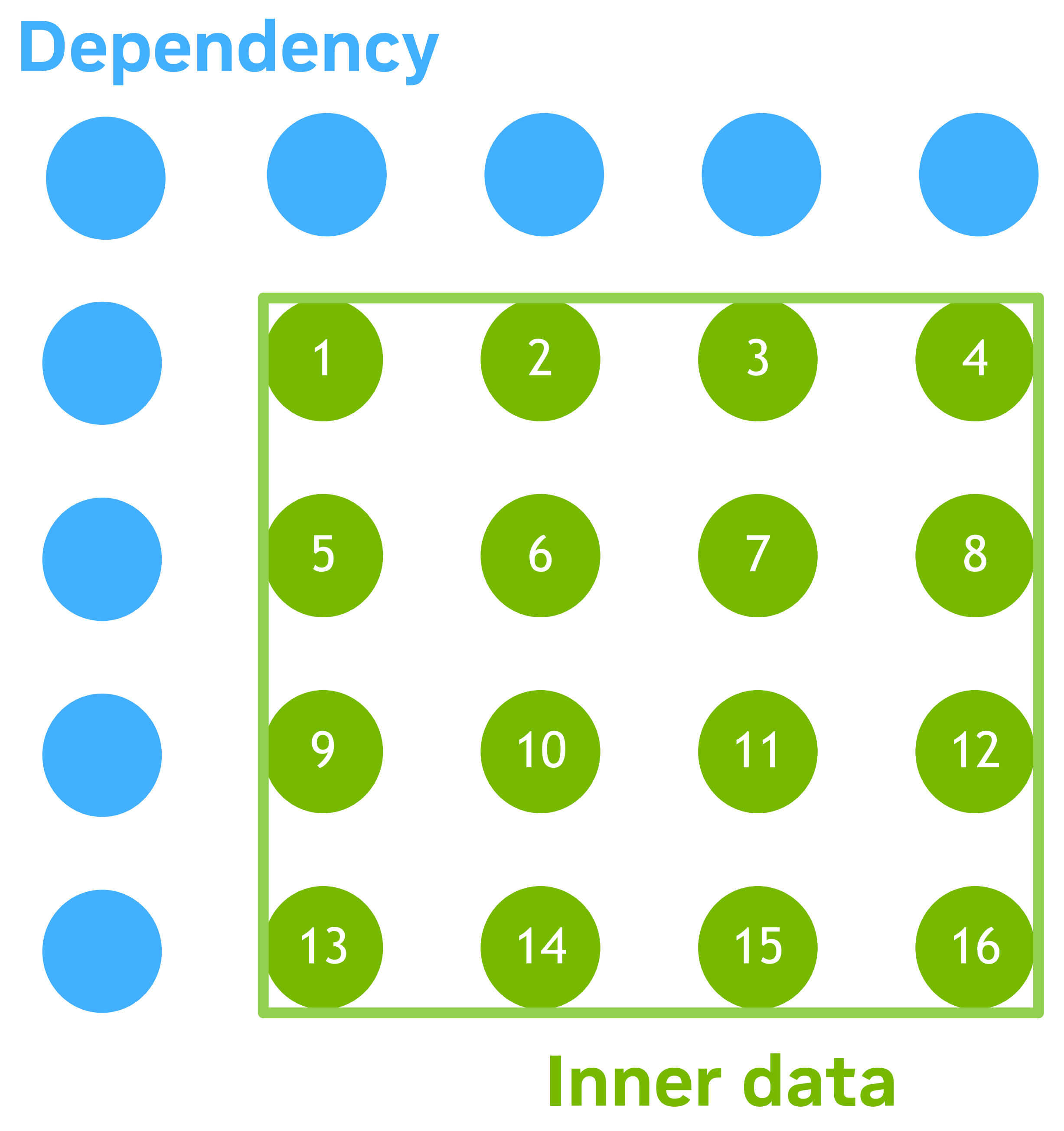
The active block will be finished after writing tile to global memory

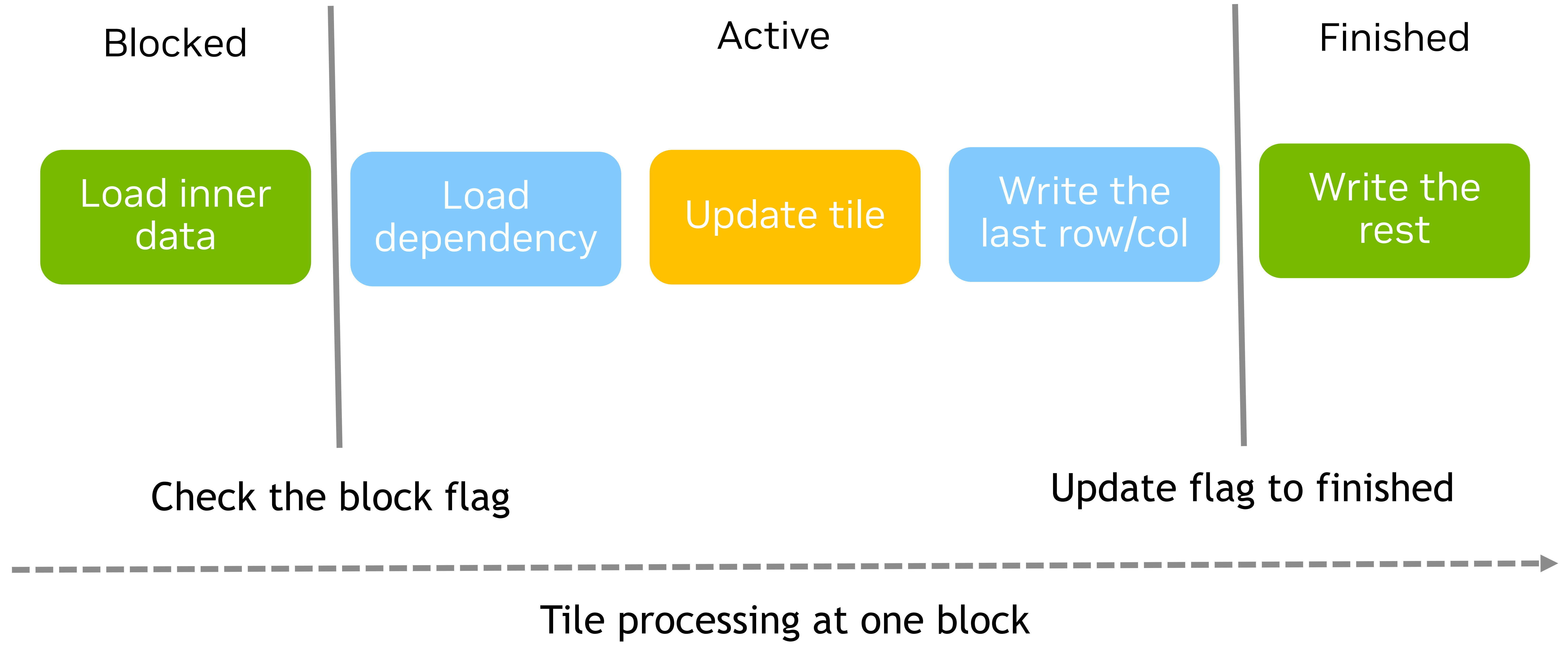


Hide the Latency of data access

How to hide the latency

- **Data prefetch & delayed data store**
  - The data loaded into shared memory can be divided into 2 parts: 1) dependency , 2) inner data
  - Only the dependency should be blocked, the latency of loading/storing the inner data can be hidden



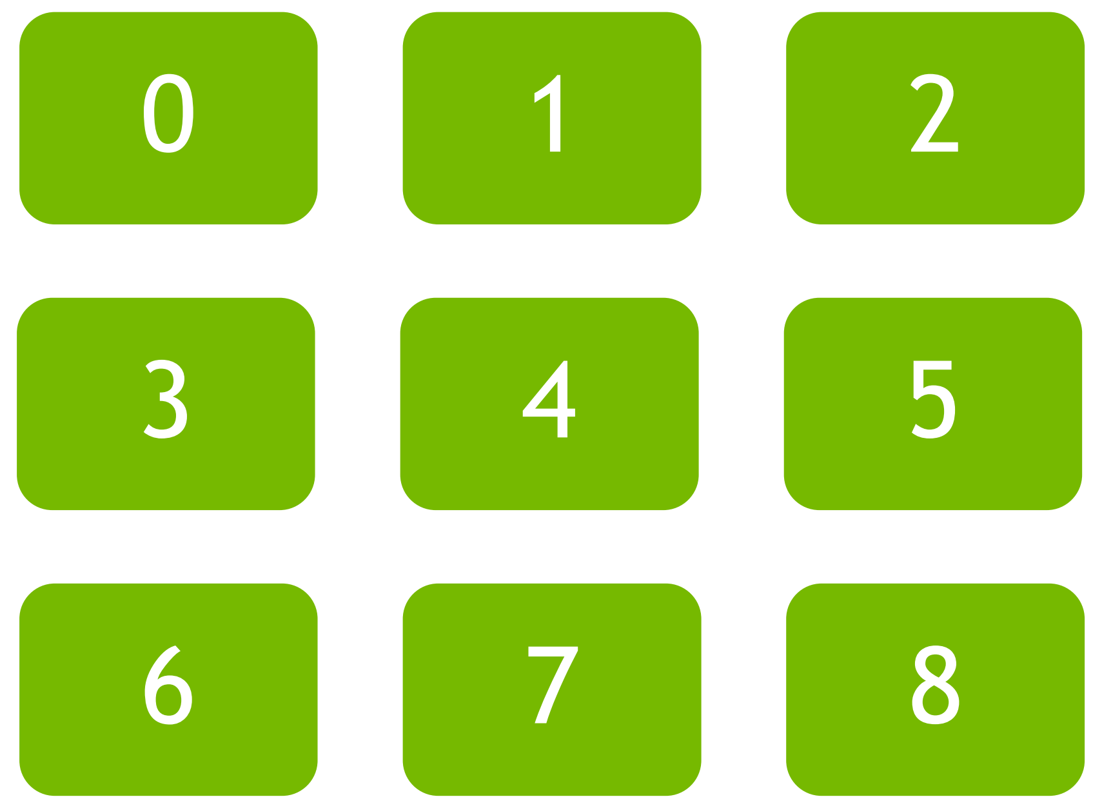


Hide the Latency of data access

The sample to hide the latency

- **Timeline of a sample**
  - 9 tiles
  - 6 blocks

🟩	Load/store inner data
⬜	Check/update flag
🟨	Load/store dependency, update tile



Timeline
Block 0
Tile 0
Tile 5
Block 1
Tile 1
Tile 7
Block 2
Tile 3
Tile 8
Block 3
Block 4
Block 5
Tile 2
Tile 4
Tile 6
Diag 0
Diag 1
Diag 2
Diag 3
Diag 4
Block id

Performance

Performance

Configuration

- **Hardware environment**
  - CPU
    - AMD EPYC 7232P 8-Core Processor
  - GPU
    - NVIDIA A30
    - NVIDIA A100
    - NVIDIA A10
    - NVIDIA Tesla T4
- **Software environment**
  - CUDA 11.3
  - C++ 17
  - Docker
- **Baseline**
  - libpng + openmp
- **Timing range**
  - Encoder
    - The interface receive the address of data to encode -> encoded data is written to the buffer of tensor
  - Decoder
    - The interface receive the path of file -> the data of decoded PNG image is written to the buffer

Performance

Datasets

- DIV2K[1]

  DIVerse 2K resolution high quality images as used for the challenges @ NTIRE (CVPR 2017 and CVPR 2018) and @ PIRM (ECCV 2018).
  - DIV2K/DIV2K_train_HR/ -- 0001.png, 0002.png, ..., 0800.png train HR images (provided to the participants)
    - Image size： ~ 1500*2000 RGB images
  - DIV2K/DIV2K_train_LR_bicubic/X2/ -- 0001x2.png, 0002x2.png, ..., 0800x2.png train LR images, downscale factor x2
    - Image size: ~ 600*800 RGB images
- imageNet[2]

  ImageNet is an image database organized according to the WordNet hierarchy (currently only the nouns), in which each node of the hierarchy is depicted by hundreds and thousands of images
  - Validation images: pick 10000 images
    - Image size: ~ 400*500 RGB images, some images reach 2500*3000

- High resolution dataset: DIV2K_train_HR
- Low resolution dataset: DIV2K_train_LR_bicubic/X2/
- HYBRID dataset: imageNet validation images

[1] E. Agustsson and R. Timofte, "NTIRE 2017 Challenge on Single Image Super-Resolution: Dataset and Study," 2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), Honolulu, HI, USA, 2017, pp. 1122-1131, doi: 10.1109/CVPRW.2017.150.
[2] Deng J , Dong W , Socher R , et al. ImageNet : A Large-Scale Hierarchical Image Database[J]. Proc. CVPR, 2009, 2009.

Performance

Datasets

- Information for different dataset

dataset	DIV_train_HR	DIV_train_LR/x2	ImageNet
Image size	~4M	~1M	200k - 20M
Image number	800	800	10000

- Several representative images

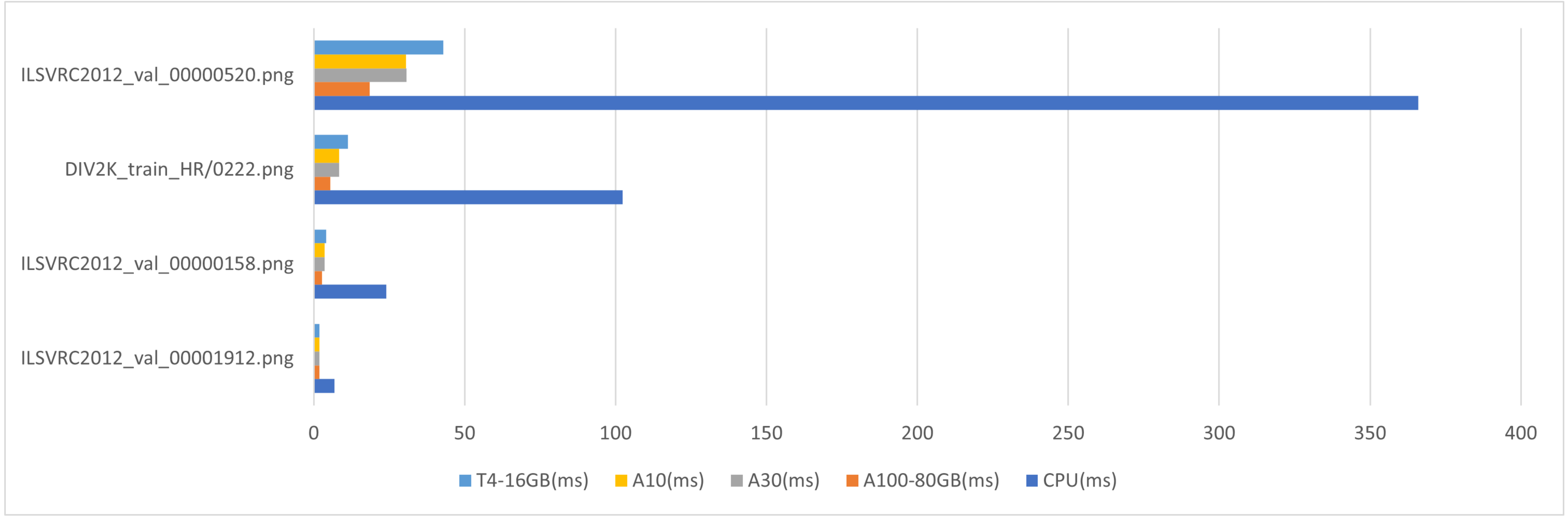
	ILSVRC2012_val_00001912.png	ILSVRC2012_val_00000158.png	DIV2K_train_HR/0222.png	ILSVRC2012_val_00000520.png
Image size	276K	1.15M	3.4M	15.6M
	(500, 370, 3)	(919, 560, 3)	(1160, 2040, 3)	(2448, 3264, 3)

Performance

Encoder

- **Test at single image**
  - The speedup at medium size image is about 6~10
  - The larger images tend to show better performance
  - The speedup of the encoder is related to the data pattern of pixel.

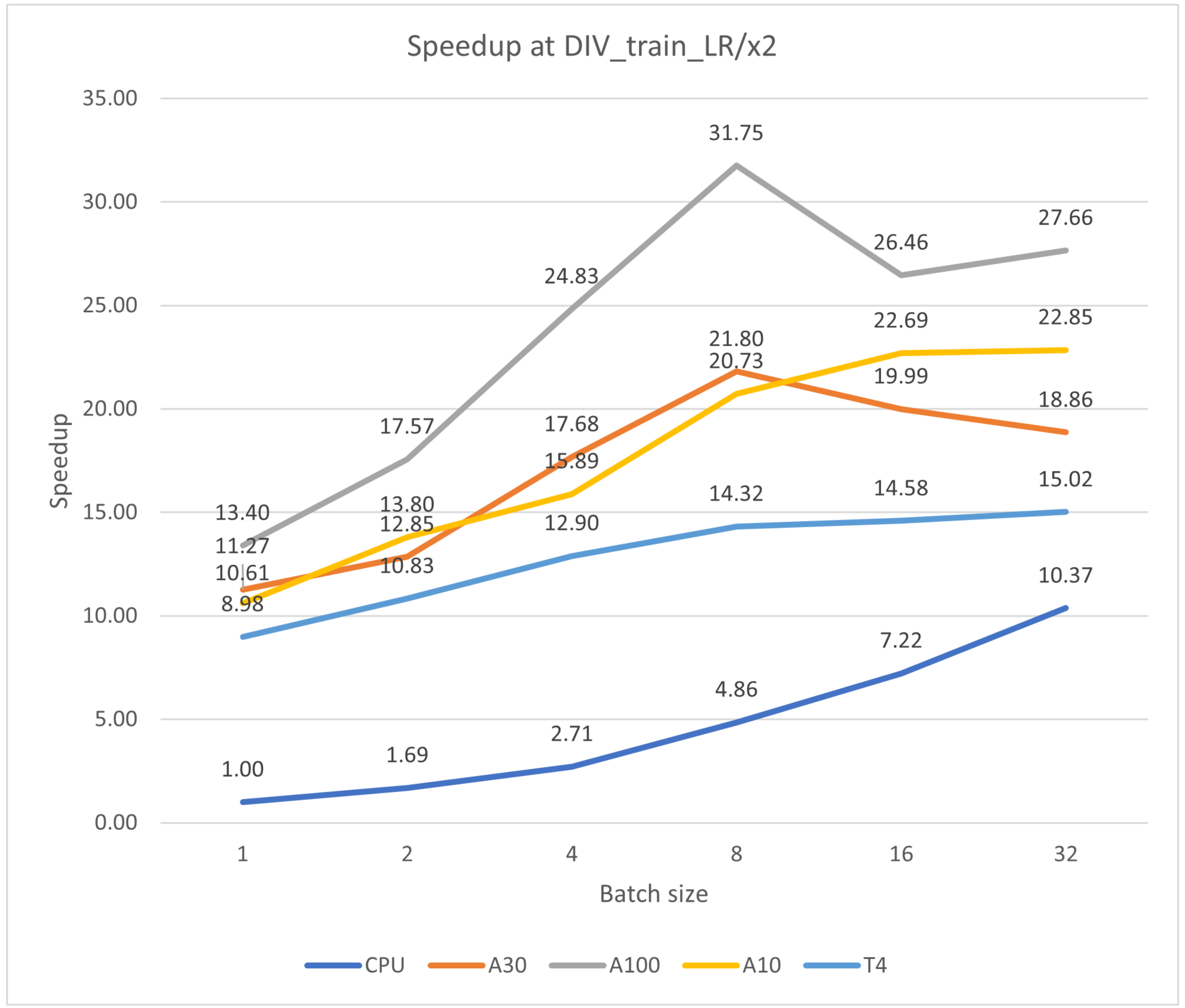
	ILSVRC2012_val_00001912.png	ILSVRC2012_val_00000158.png	DIV2K_train_HR/0222.png	ILSVRC2012_val_00000520.png
CPU(ms)	6.83	24.04	102.33	365.99
A100-80GB(ms)	1.86	2.67	5.56	18.50
A30(ms)	1.87	3.59	8.33	30.67
A10(ms)	1.89	3.56	8.38	30.59
T4-16GB(ms)	1.90	4.05	11.36	42.85

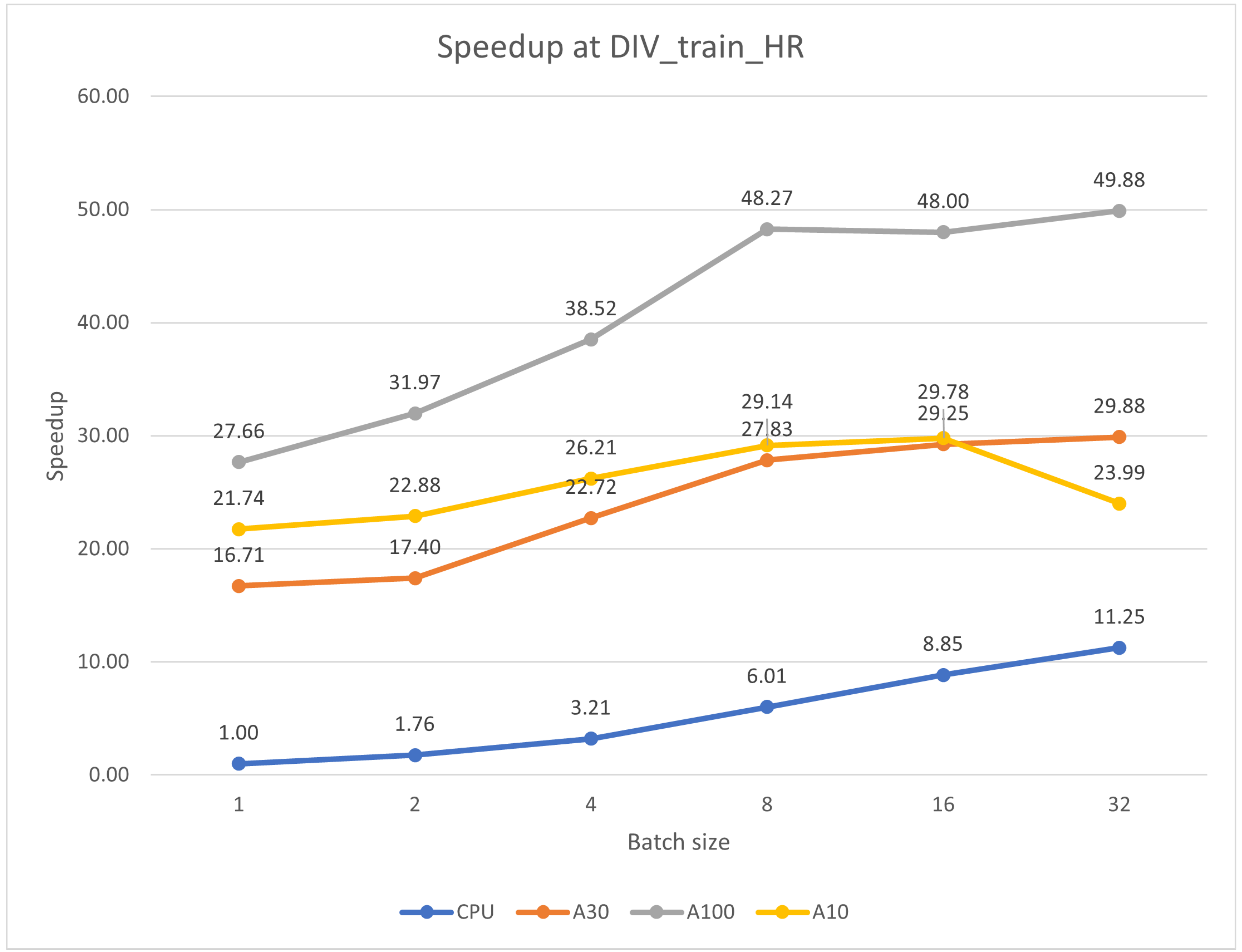


Performance

Encoder

- Speedup: compared to the CPU(batch size = 1)
  - Get best performance when batch size = 16/32
  - The scalability is not as good as baseline
    - SM assigned to each image is not enough
  - Perform better at HR dataset





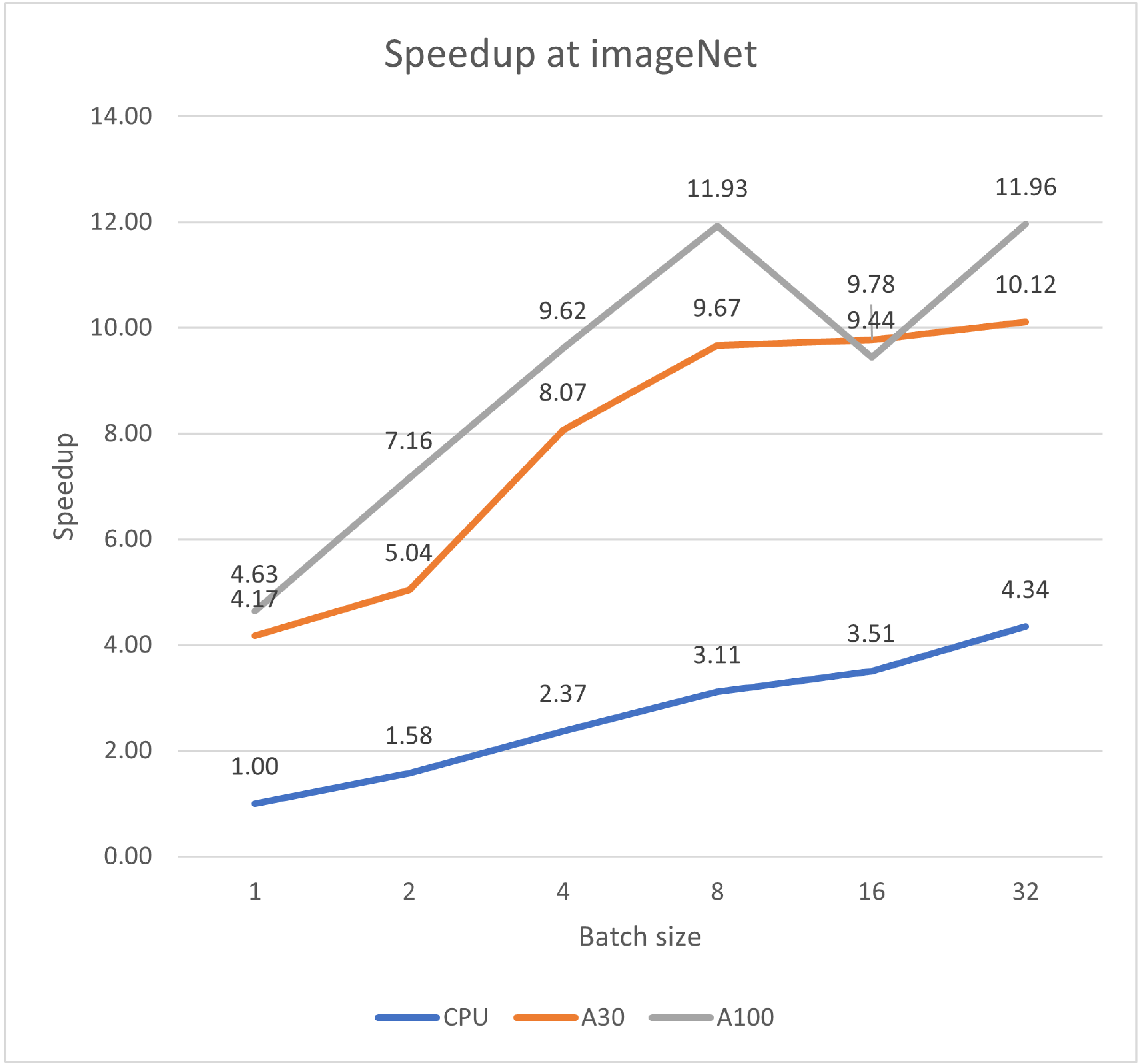
Performance

Encoder

- Performance on imageNet validation dataset
  - Speedup: compared to the CPU(batch size = 1)

Speedup at different batch size

batch size	**CPU**	**A30**	**A100**
1	1.00	4.17	4.63
2	1.58	5.04	7.16
4	2.37	8.07	9.62
8	3.11	9.67	11.93
16	3.51	9.78	9.44
32	4.34	10.12	11.96

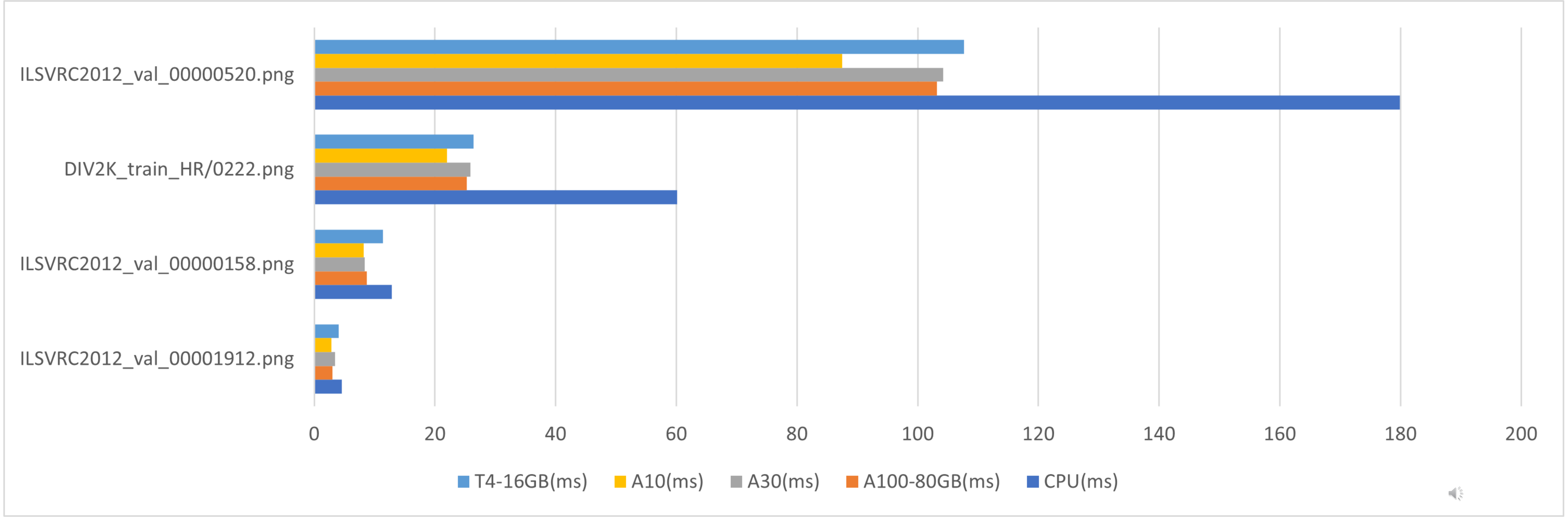


Performance

Decoder

- **Test at single image**
  - The speedup at medium size image is about 1.5~2
  - The larger images tend to show better performance
  - The speedup of the decoder is related to the data pattern of pixel.
  - The inflate compression and the defilter is hard to parallel because the data dependency, so the speedup is worse than encoder

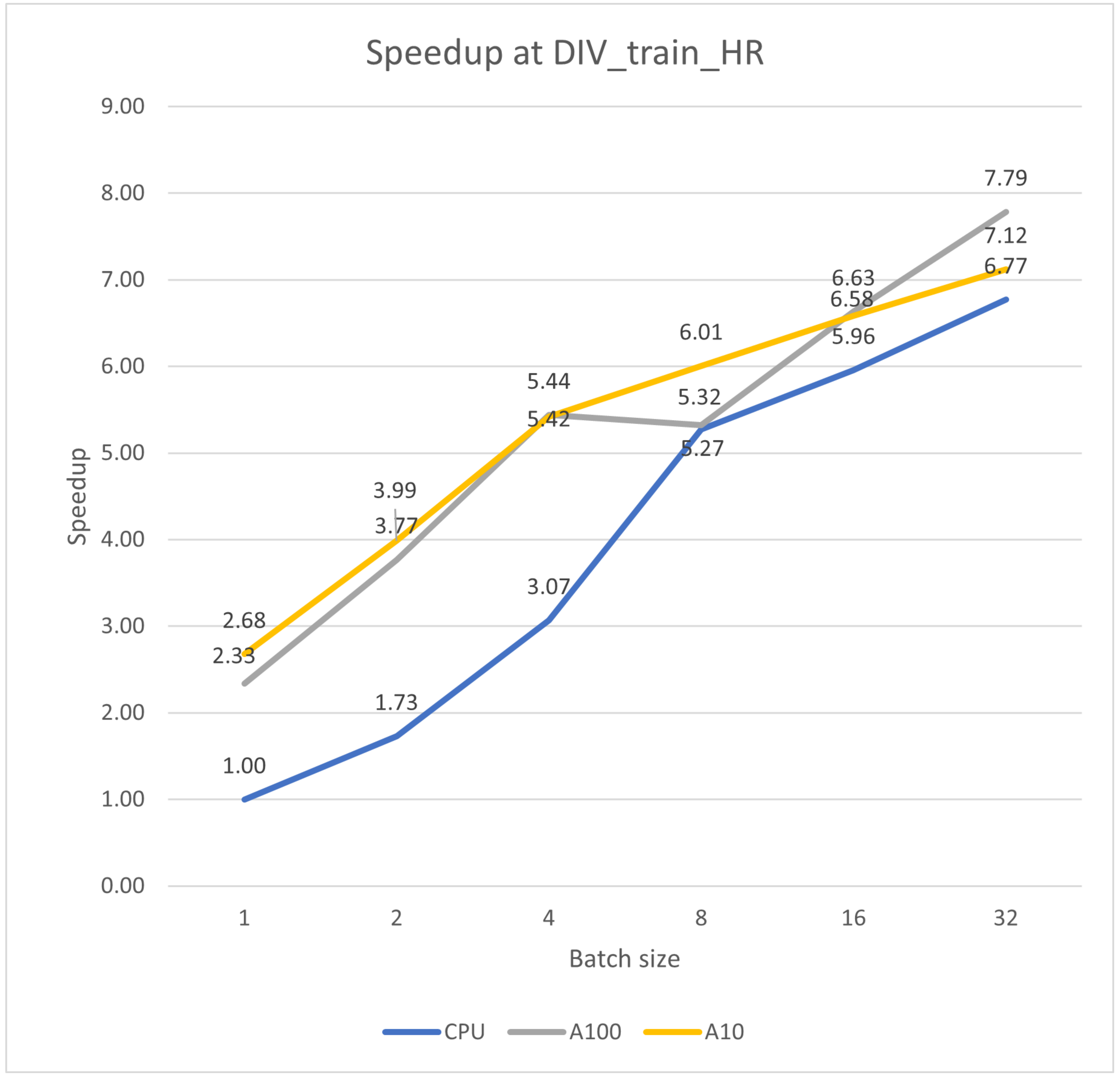
	ILSVRC2012_val_00001912.png	ILSVRC2012_val_00000158.png	DIV2K_train_HR/0222.png	ILSVRC2012_val_00000520.png
CPU(ms)	4.57	12.90	60.10	179.92
A100-80GB(ms)	3.03	8.73	25.26	103.18
A30(ms)	3.46	8.38	25.90	104.24
A10(ms)	2.82	8.16	22.00	87.47
T4-16GB(ms)	4.03	11.42	26.40	107.70

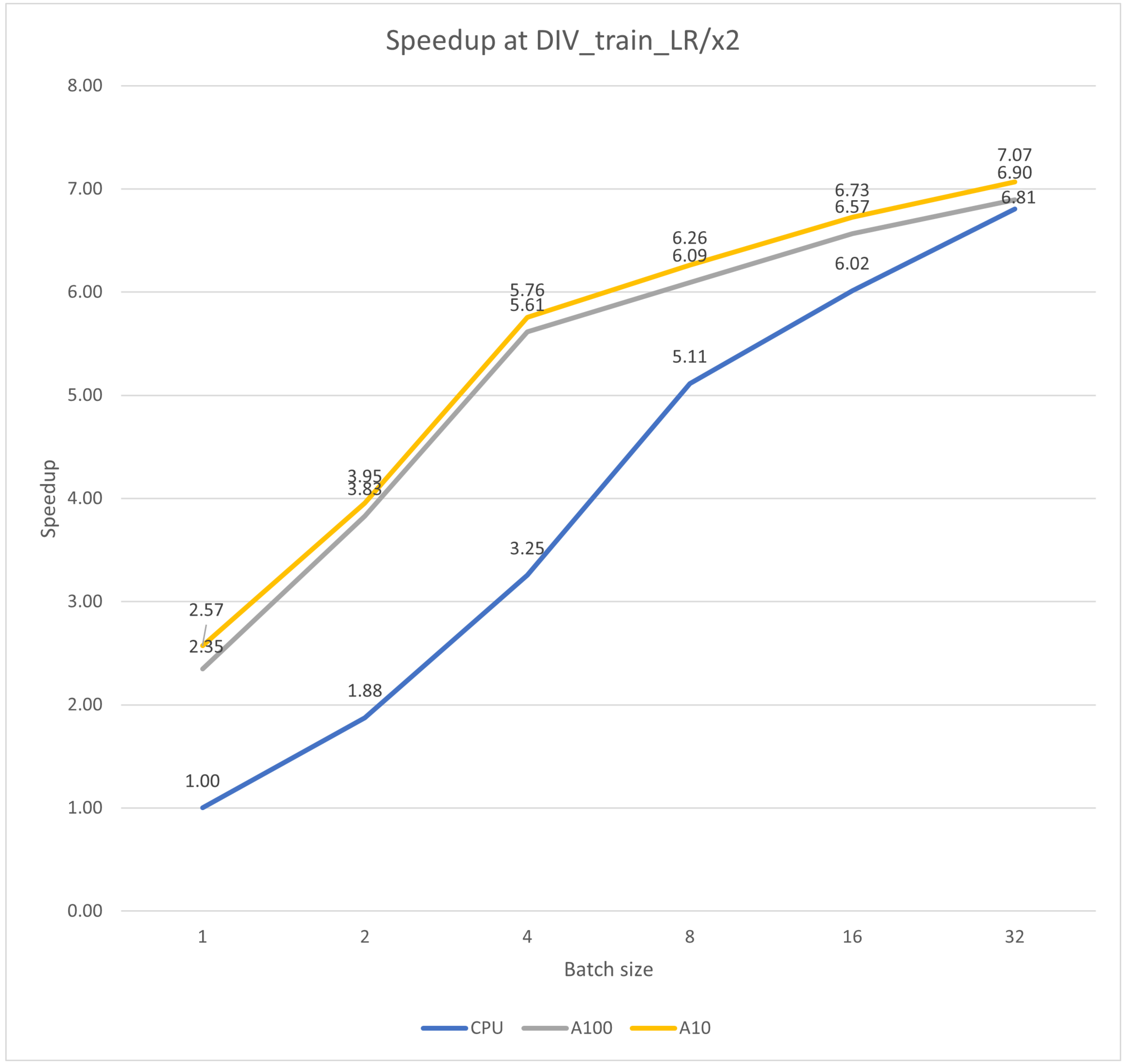


Performance

Decoder

- Speedup: compared to the CPU(batch size = 1)
  - GPU decoder suffers from low occupancy
  - Decoder works better on larger images





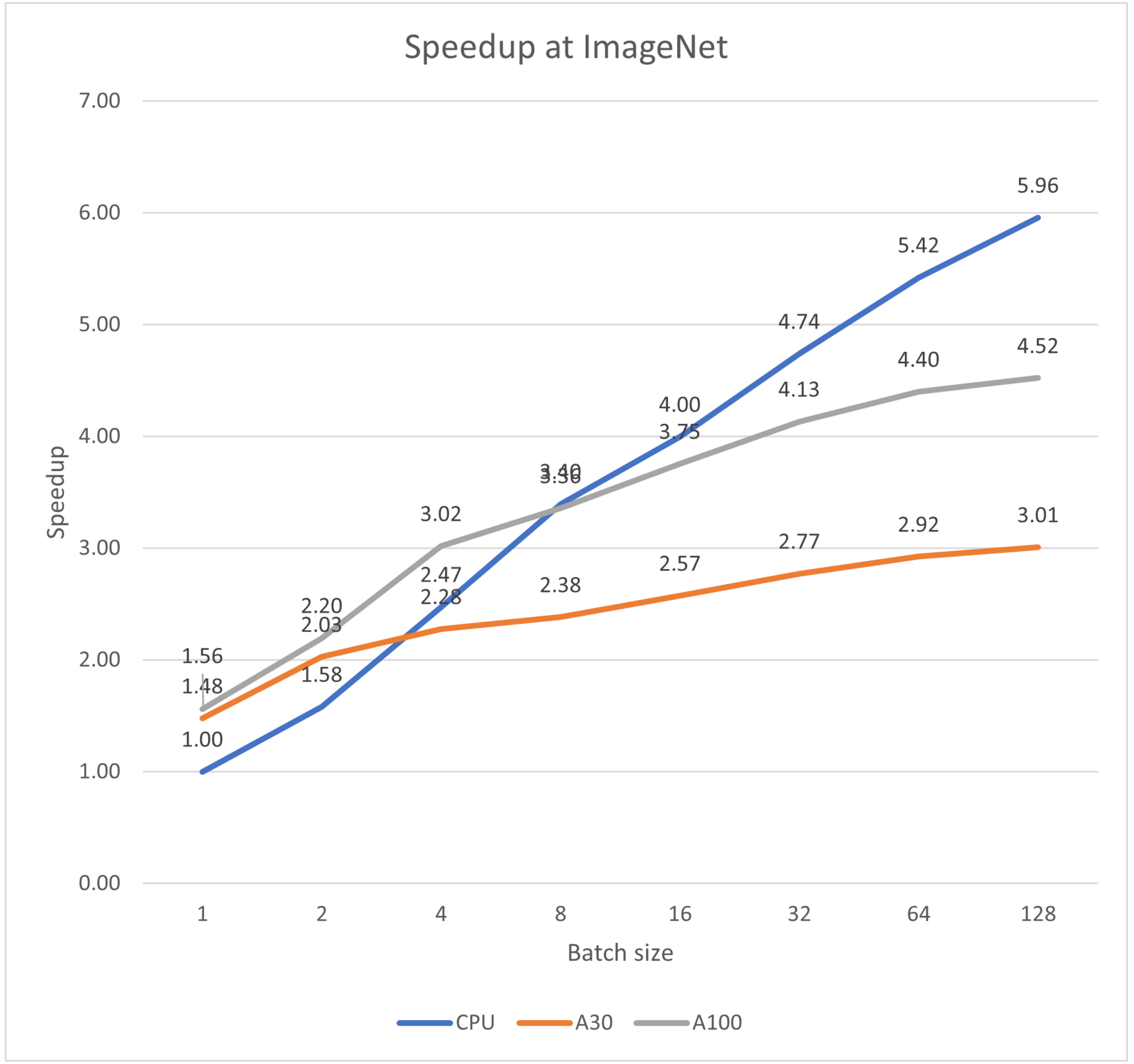
Performance

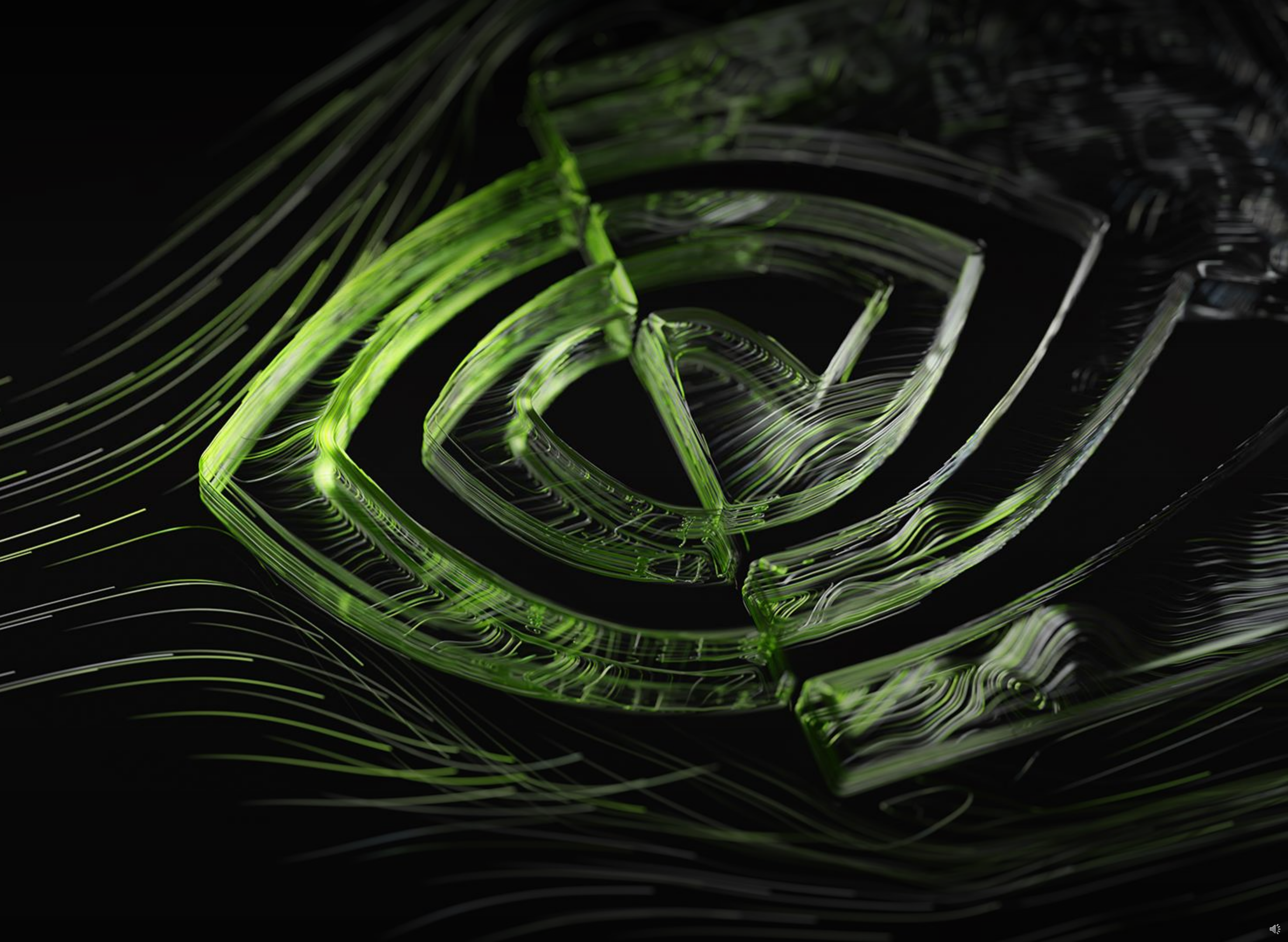
Decoder

- Performance on ImageNet validation dataset
  - Speedup: compared to the CPU(batch size = 1)
  - Much smaller image size(~300K)
  - Will improve the scalability continuously

Speedup at different batch size

batch size	**CPU**	**A30**	**A100**
1	1.00	1.48	1.56
2	1.58	2.03	2.20
4	2.47	2.28	3.02
8	3.40	2.38	3.36
16	4.00	2.57	3.75
32	4.74	2.77	4.13
64	5.42	2.92	4.40
128	5.96	3.01	4.52





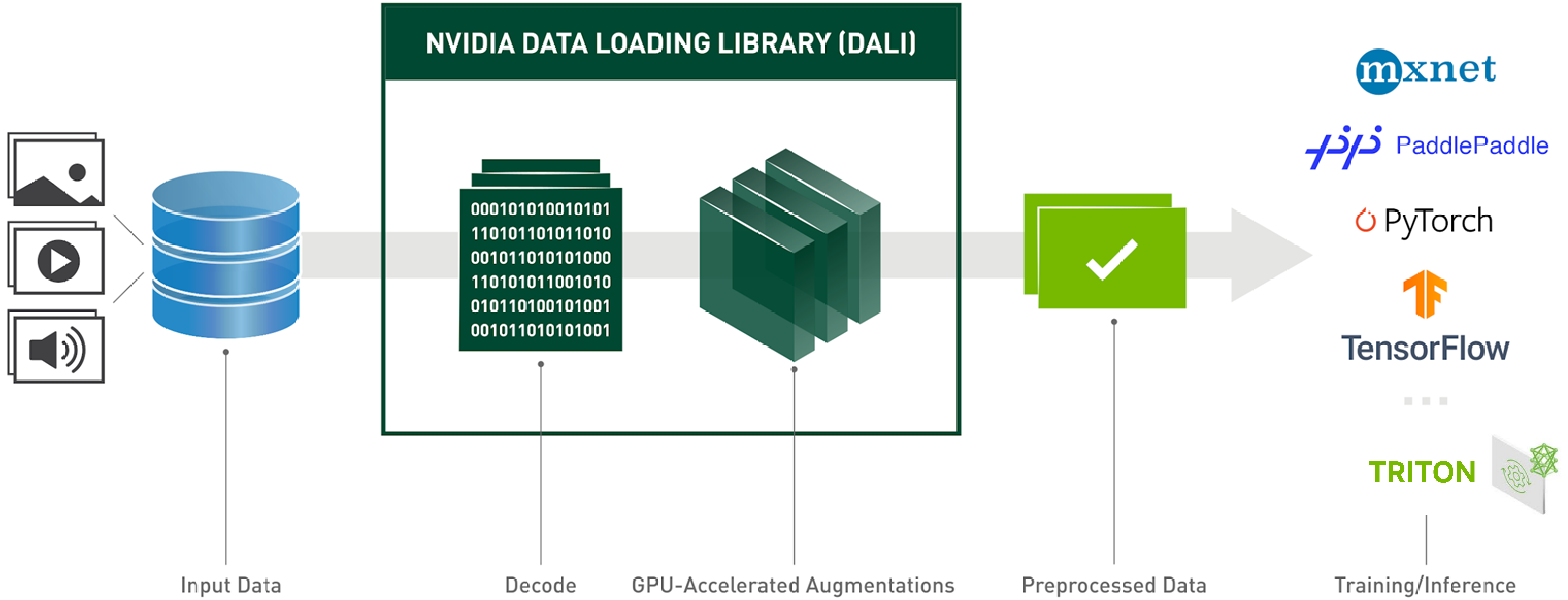
Summary

Summary

- Our work efficiently accelerates the key components of PNG encoder and decoder with CUDA;
- Our PNG encoder shows remarkable speedup over CPU and good Scalability;
- The decoder shows qualified performance at small batch size, while for large batch size, more efforts is needed to improve the scalability;
- The interfaces are concise and clear, and we would optimize them further based on the users' feedback;

Image Processing Ecosystem of NVIDIA

DALI



- Easy-to-use Functional API, drop-in framework integration, CPU & GPU pipelines
- Portable data processing pipeline – easily retarget to different frameworks for training and inference (via dedicated DALI backend - https://github.com/triton-inference-server/dali_backend)
- Provides GPU acceleration for different data formats – JPEG, JPEG2000, video; soon PNG and TIFF
- https://docs.nvidia.com/deeplearning/dali/user-guide/docs/ && https://github.com/NVIDIA/DALI

Acknowledge

- Thanks for Gems Guo’s work on Huffman encoding;
- Thanks for Mauro Bisson’s work on crc32 algorithm;
- Thanks for Eyal Soha’s work on inflate algorithm;

NVIDIA®