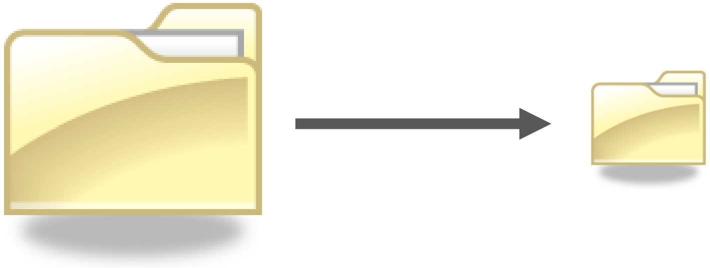
Lossless Data Compression on GPUs

Ritesh Patel
University of California, Davis

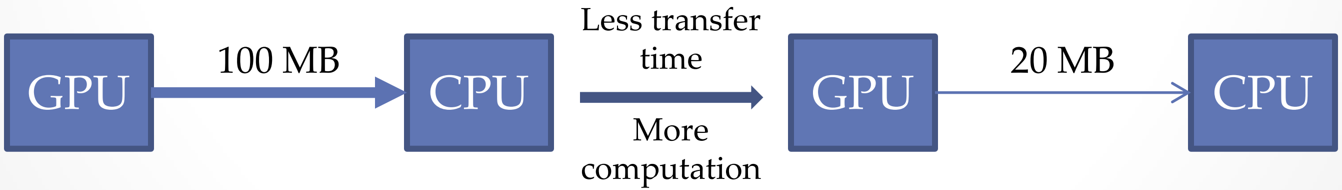
Jason Mak
University of California, Davis

Motivation

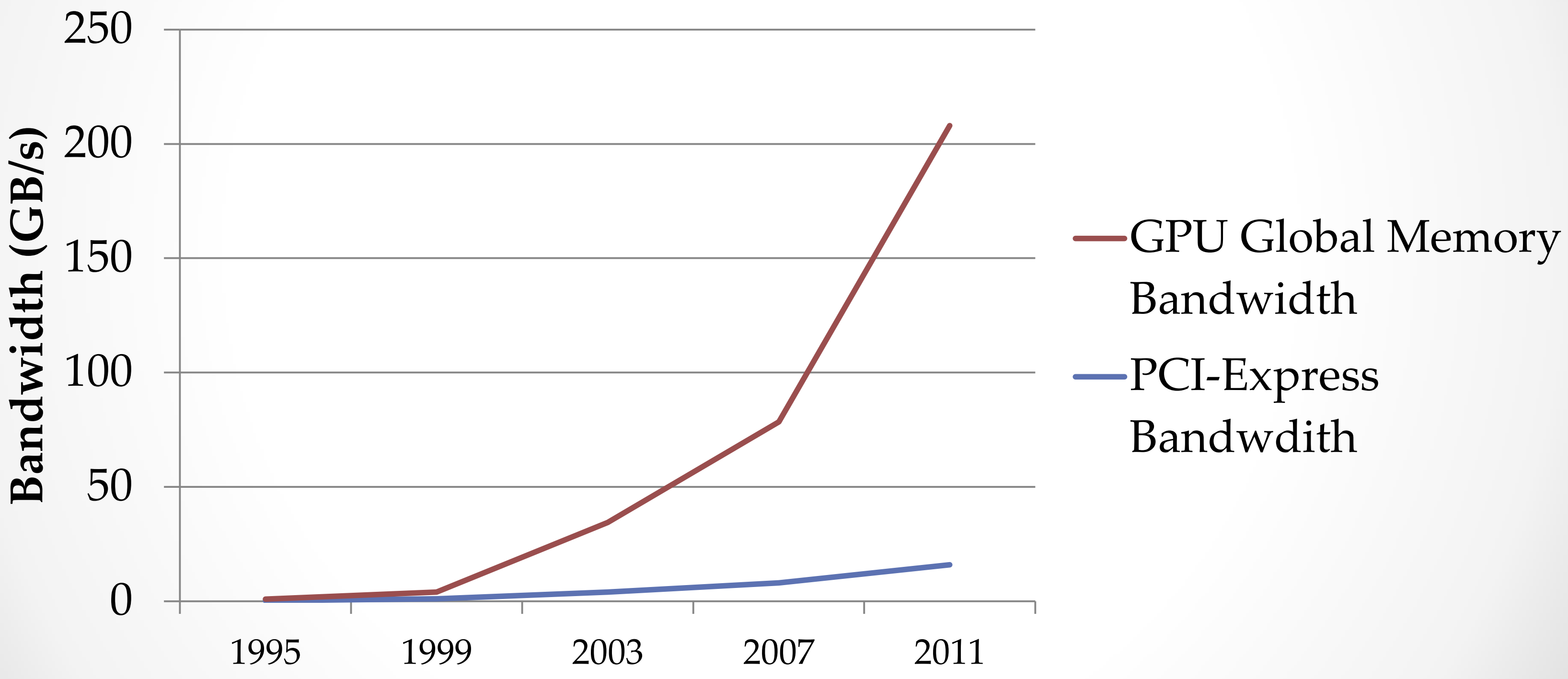
- Data storage



- Computation vs. Data Transfer
  - Is compress-then-send worthwhile?



PCI-E vs. GPU bandwidth



Topics Covered

- Related Work
- Three Algorithms in the Compression Pipeline
  1. Burrows-Wheeler Transform
  2. Move-to-Front Transform
  3. Huffman Coding
- Results
- Future Work

Related Work

- Domain-specific compression
  - Floating-point data compression
  - Texture compression
- Parallel bzip2 (pbzip2)
  - Uses pthread library
  - bzip2 not data-parallel-friendly

Compression Pipeline

Input Data: **n characters**

Burrows-Wheeler Transform

n characters | **Many runs of repeated characters**

Move-to-Front Transform

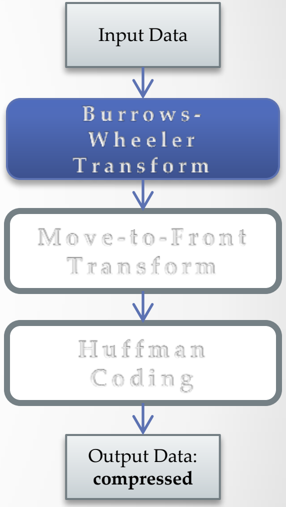
n characters | **Low entropy → Lots of 0s, 1s, etc.**

Huffman Coding

Output Data: **compressed**

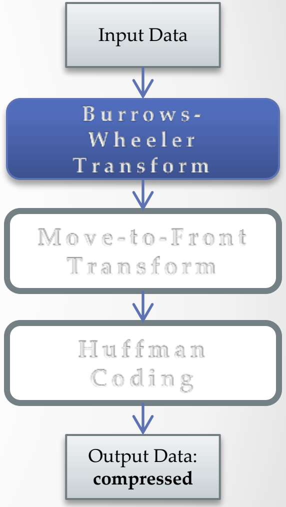
Burrows-Wheeler Transform

- Transforms a string and gives has many runs of repeated characters
- Same characters get grouped



Burrows-Wheeler Transform

- Transforms a string and gives has many runs of repeated characters
- Same characters get grouped

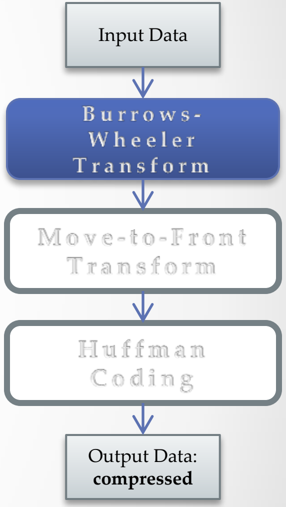


Burrows-Wheeler Transform

1) Cyclical Rotations

BWT Input:
ababacabac

a	**b**	**a**	**b**	**a**	**c**	**a**	**b**	**a**	**c**
c	a	b	a	b	a	c	a	b	a
a	c	a	b	a	b	a	c	a	b
b	a	c	a	b	a	b	a	c	a
a	b	a	c	a	b	a	b	a	c
c	a	b	a	c	a	b	a	b	a
a	c	a	b	a	c	a	b	a	b
b	a	c	a	b	a	c	a	b	a
a	b	a	c	a	b	a	c	a	b
b	a	b	a	c	a	b	a	c	a



Burrows-Wheeler Transform

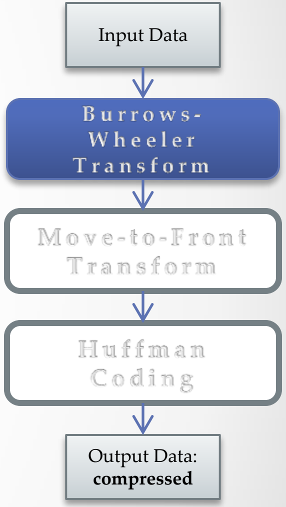
BWT Input:
ababacabac

1) Cyclical Rotations

a	**b**	**a**	**b**	**a**	**c**	**a**	**b**	**a**	**c**
c	a	b	a	b	a	c	a	b	a
a	c	a	b	a	b	a	c	a	b
b	a	c	a	b	a	b	a	c	a
a	b	a	c	a	b	a	b	a	c
c	a	b	a	c	a	b	a	b	a
a	c	a	b	a	c	a	b	a	b
b	a	c	a	b	a	c	a	b	a
a	b	a	c	a	b	a	c	a	b
b	a	b	a	c	a	b	a	c	a

2) Sort

a	**b**	**a**	**b**	**a**	**c**	**a**	**b**	**a**	**c**
a	b	a	c	a	b	a	b	a	c
a	b	a	c	a	b	a	c	a	b
a	c	a	b	a	b	a	c	a	b
a	c	a	b	a	c	a	b	a	b
b	a	b	a	c	a	b	a	c	a
b	a	c	a	b	a	b	a	c	a
b	a	c	a	b	a	c	a	b	a
c	a	b	a	b	a	c	a	b	a
c	a	b	a	c	a	b	a	b	a



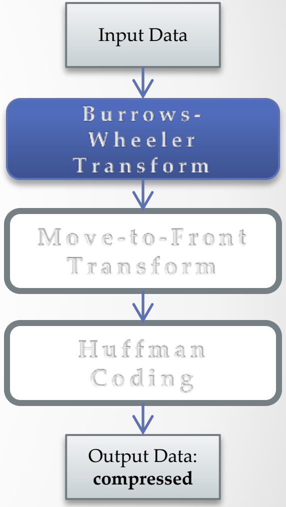
Burrows-Wheeler Transform

Sorted Cyclical Rotations

a b a b a c a b a **c**
a b a c a b a b a **c**
a b a c a b a c a **b**
a c a b a b a c a **b**
a c a b a c a b a **b**
b a b a c a b a c **a**
b a c a b a b a c **a**
b a c a b a c a b **a**
c a b a b a c a b **a**
c a b a c a b a b **a**

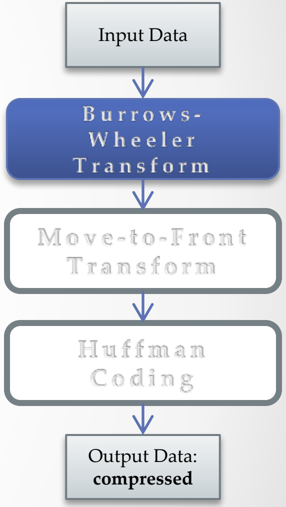
→

Index: 0
BWT: **ccbbbaaaaa**



BWT with Merge Sort

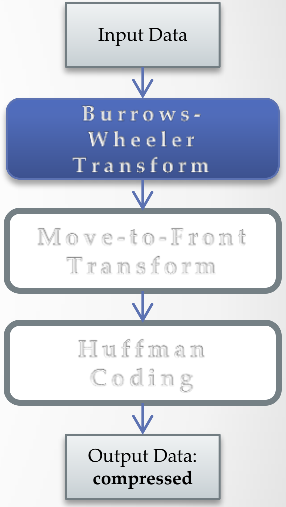
- Sorting **n** strings → each **n** characters long
  - **n** = 1 million per block in our implementation
- Radix sort not a good fit
- Break ties on the fly



BWT with Merge Sort

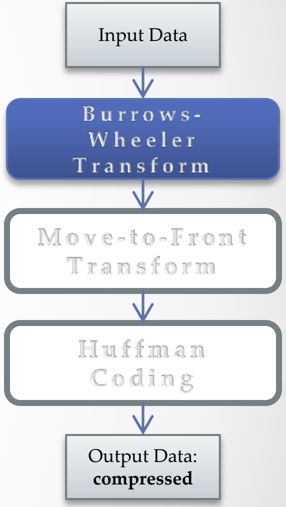
- Sorting **n** strings → each **n** characters long
  - **n** = 1 million per block in our implementation
- Radix sort not a good fit
- Break ties on the fly
- String sorting on GPU
  - Non-uniform

	a	b	a	b	a	c	a	b	a	c
Pair 1: 7 ties	**a**	**b**	**a**	**c**	**a**	**b**	**a**	b	a	c
	a	**b**	**a**	**c**	**a**	**b**	**a**	c	a	b
	a	c	a	b	a	b	a	c	a	b
Pair 2: 0 ties	a	c	a	b	a	c	a	b	a	b
	b	a	b	a	c	a	b	a	c	a
	b	a	c	a	b	a	b	a	c	a
	b	a	c	a	b	a	c	a	b	a
Pair 3: 4 ties	**c**	**a**	**b**	**a**	b	a	c	a	b	a
	c	**a**	**b**	**a**	c	a	b	a	b	a



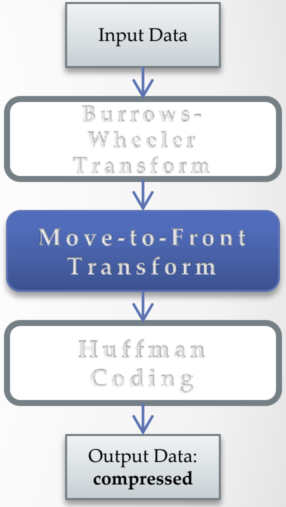
BWT with Merge Sort

- Sorting **n** strings → each **n** characters long
  - **n** = 1 million per block in our implementation
- Radix sort not a good fit
- Break ties on the fly
- String sorting on GPU
  - Non-uniform
  - Parallelize BWT with string sorting algorithm based on merge sort
  - Currently fastest string sort on GPU



Move-to-Front Transform

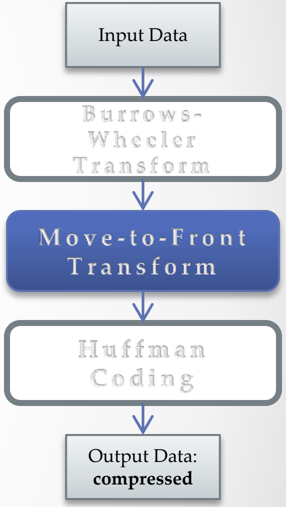
- Exploit clumpy characters from BWT
  - BWT: `ababacabac` → **ccbbbaaaaa**
- Improves effectiveness of entropy encoding



Move-to-Front Transform

- Each symbol in the data is replaced by its index in the list
  - Initial list: …abcd… (ASCII)
- Recently seen characters are kept at front of the list
  - Long sequences of identical symbols replaced by many zeros

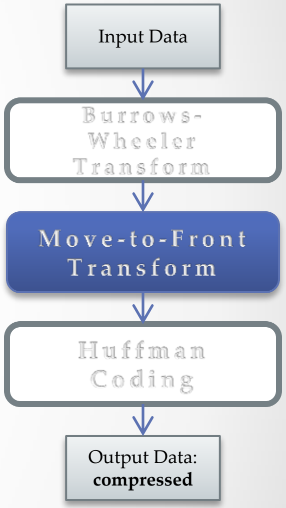
`ccbbbaaaaa` → Move-to-Front Transform → `99,0,99,0,0,99,0,0,0,0`



Move-to-Front Transform

Input	MTF List	Output
ccbbbaaaaa	…abc… (ASCII)	[99]

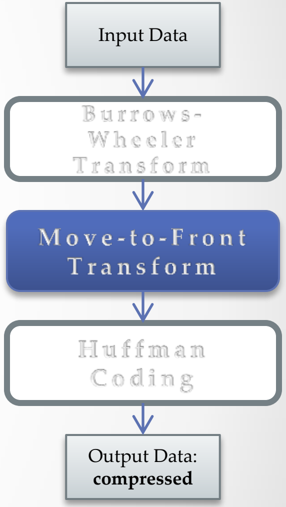
- 'c' occurs at 99^{th} index → Output '**99**'



Move-to-Front Transform

Input	MTF List	Output
ccbbbaaaaa	c...ab... (ASCII)	[99]

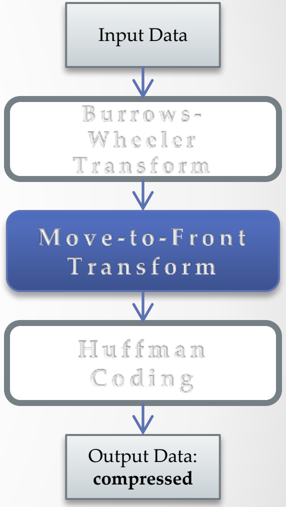
- ‘c’ occurs at 99th index → Output ’99’
- Move ‘c’ to front of the MTF list
- Shift all previous elements to the right



Move-to-Front Transform

Input	MTF List	Output
ccbbbaaaaa	...ab**c**... (ASCII)	[99]
c**c**bbbaaaaa	**c**...ab...	[99,**0**]

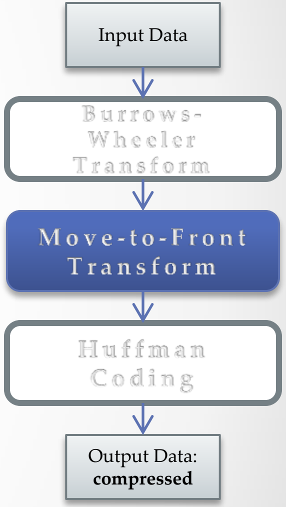
- ‘**c**’ occurs at 0^{th} index → Output ’**0**’



Move-to-Front Transform

Input	MTF List	Output
ccbbbaaaaa	...abc... (ASCII)	[99]
ccbbbaaaaa	c...ab...	[99,0]

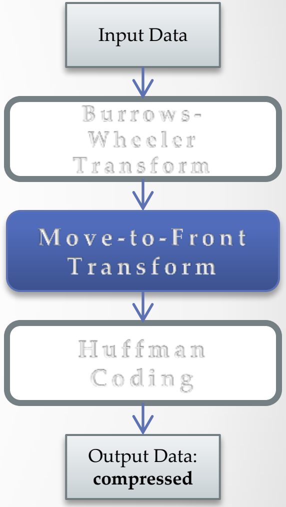
- ‘c’ occurs at 0th index → Output ’0’
- ‘c’ already at front, no shifts



Move-to-Front Transform

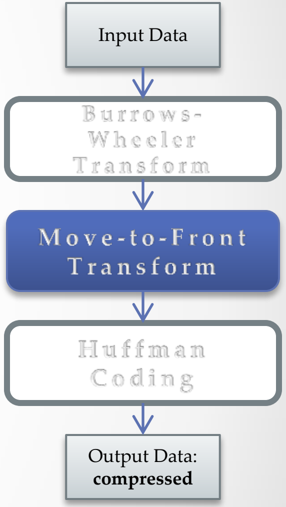
Input	MTF List	Output
ccbbbaaaaa	...abc... (ASCII)	[99]
ccbbbaaaaa	c...ab...	[99,0]

- 'c' occurs at 0^{th} index → Output '0'
- 'c' already at front, no shifts
- Repeat for all elements



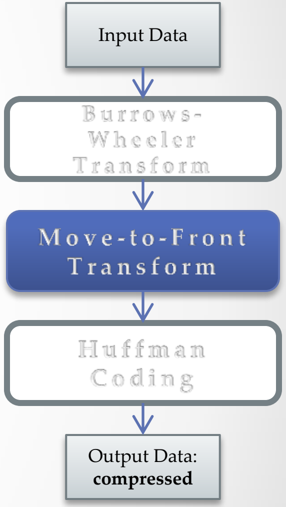
Move-to-Front Transform

Iteration	MTF List	Transformed String
ccbbbaaaaa	...abc... (ASCII)	[**99**]
c**c**bbbaaaaa	c...ab...	[99,**0**]
cc**b**bbaaaaa	c...ab...	[99,0,**99**]
ccb**b**baaaaa	bc...a...	[99,0,99,**0**]
ccbb**b**aaaaa	bc...a...	[99,0,99,0,**0**]
ccbbb**a**aaaa	bc...a...	[99,0,99,0,0,**99**]
ccbbba**a**aaa	abc...	[99,0,99,0,0,99,**0**]
ccbbbaa**a**aa	abc...	[99,0,99,0,0,99,0,**0**]
ccbbbaaa**a**a	abc...	[99,0,99,0,0,99,0,0,**0**]
ccbbbaaaa**a**	abc...	**[99,0,99,0,0,99,0,0,0,0]**



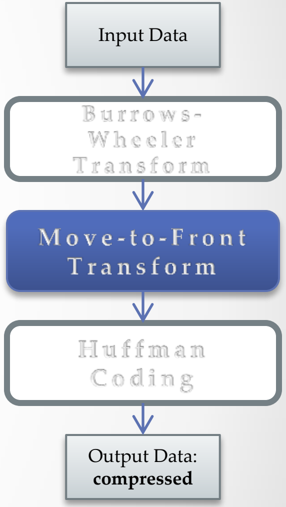
Parallel MTF

- MTF appears to be highly serial
  - Character-by-character dependency



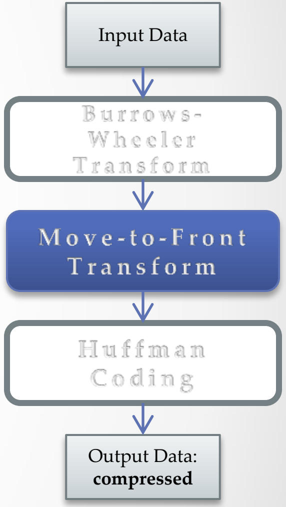
Parallel MTF

- MTF appears to be highly serial
  - Character-by-character dependency
- Goal: generate multiple MTF lists at different indices
  - Compute MTF output in parallel



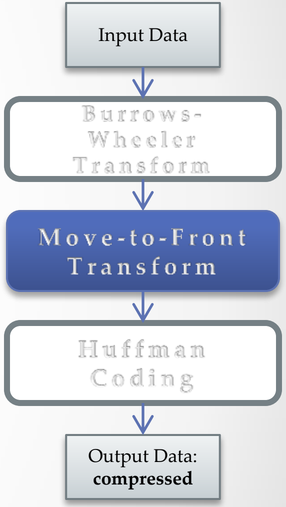
Parallel MTF

- MTF appears to be highly serial
  - Character-by-character dependency
- Goal: generate multiple MTF lists at different indices
  - Compute MTF output in parallel
- How to parallelize?
  - Two key insights
    1. Generate partial MTF list of a substring s
    2. Combine two adjacent partial MTF lists (using "concatenation")



Example Parallel MTF

B A D A D D D A C C C B B A A A



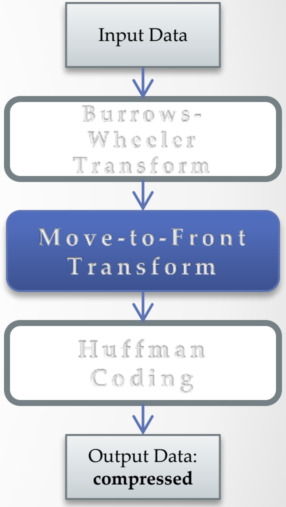
Example Parallel MTF

Part 1

B A D A D D D A C C C B B A A A

End of part 1

MTF List	**A D B**



Example Parallel MTF

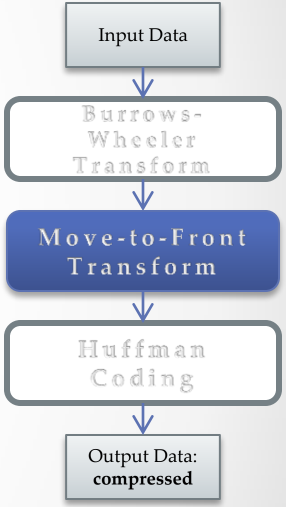
Part 1 Part 2

B A D A D D D A C C C B B A A A

End of part 2

Final MTF List

A B C D



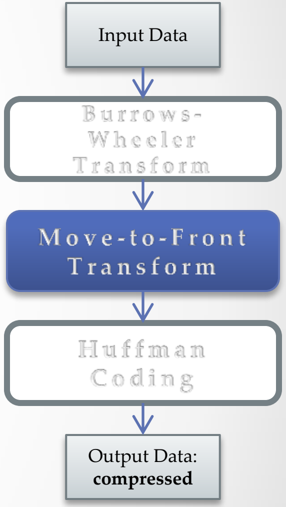
Example Parallel MTF

Break input into 2 chunks

B A D A D D D A	C C C B B A A A

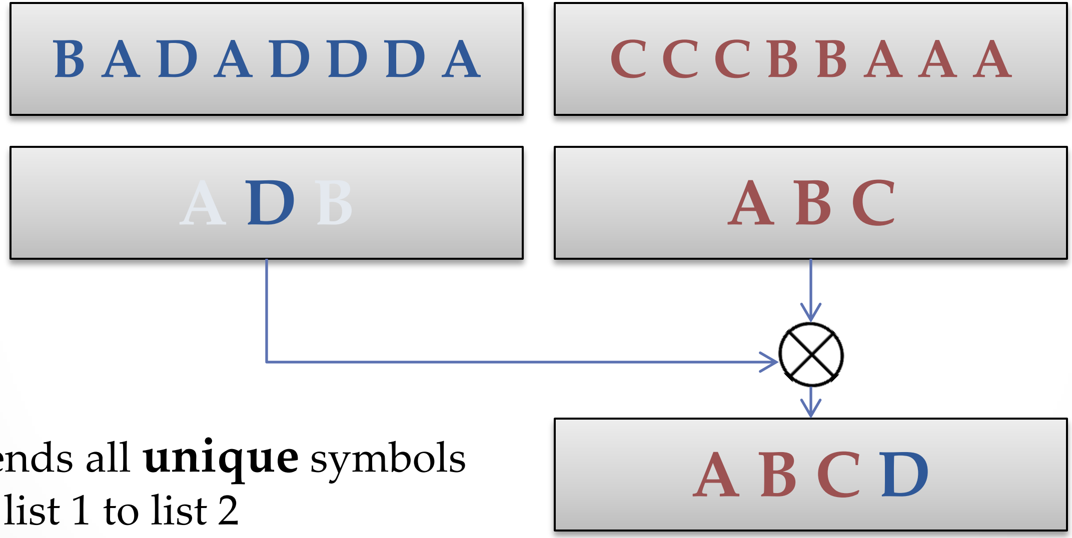
List 1	List 2
A D B	A B C

2 partial MTF lists
(in parallel)

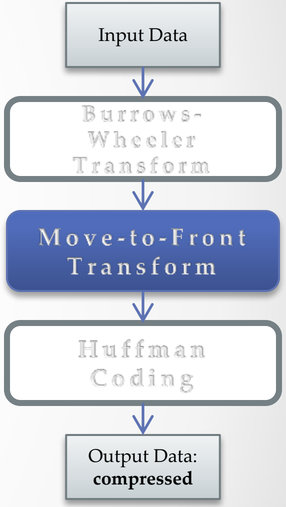


Example Parallel MTF

2 partial MTF lists
(in parallel)

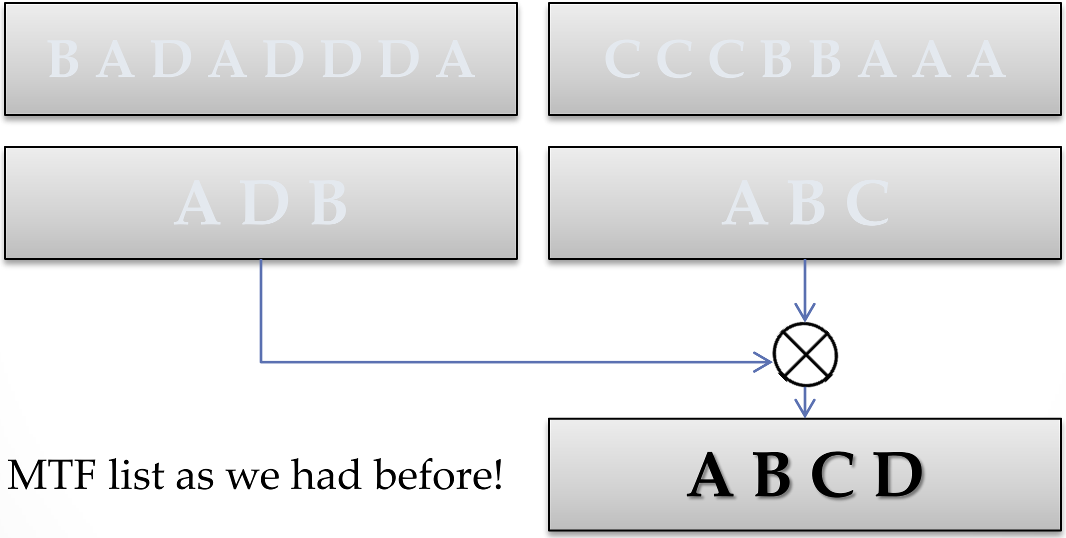


$\otimes$ Appends all **unique** symbols from list 1 to list 2

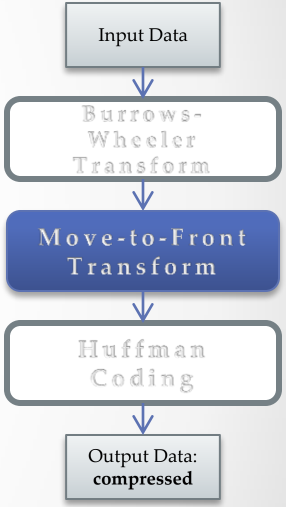


Example Parallel MTF

2 partial MTF lists
(in parallel)



Same MTF list as we had before!



Example Parallel MTF

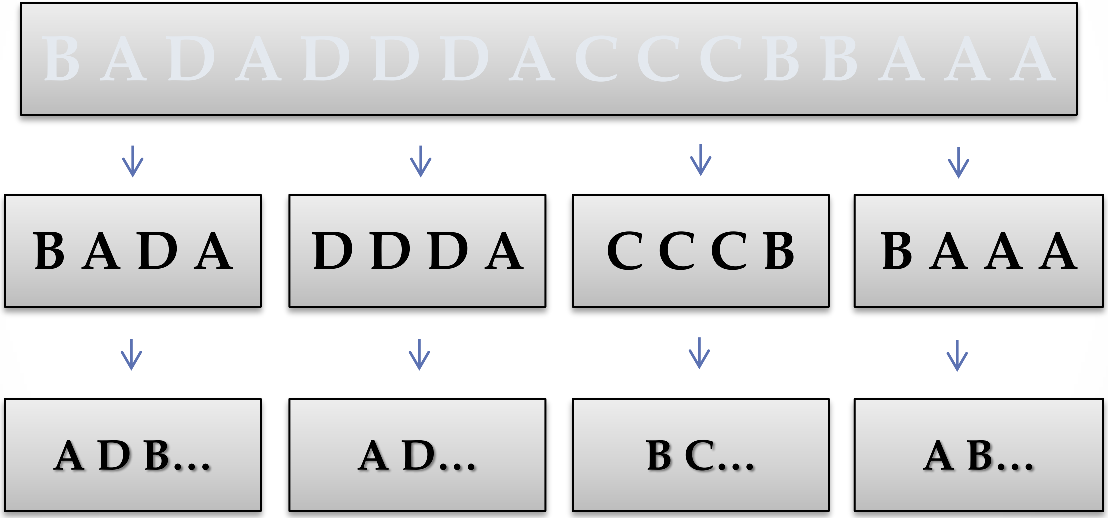
B A D A D D D A C C C B B A A A

Break input into 4 chunks

B A D A	D D D A	C C C B	B A A A

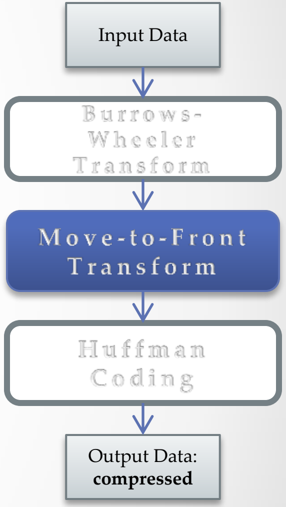
Input Data → Burrows-Wheeler Transform → **Move-to-Front Transform** → Huffman Coding → Output Data: **compressed**

Example Parallel MTF



Break input into 4 chunks

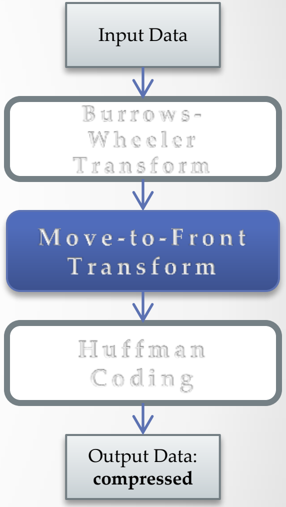
Generate partial MTF lists



Example Parallel MTF

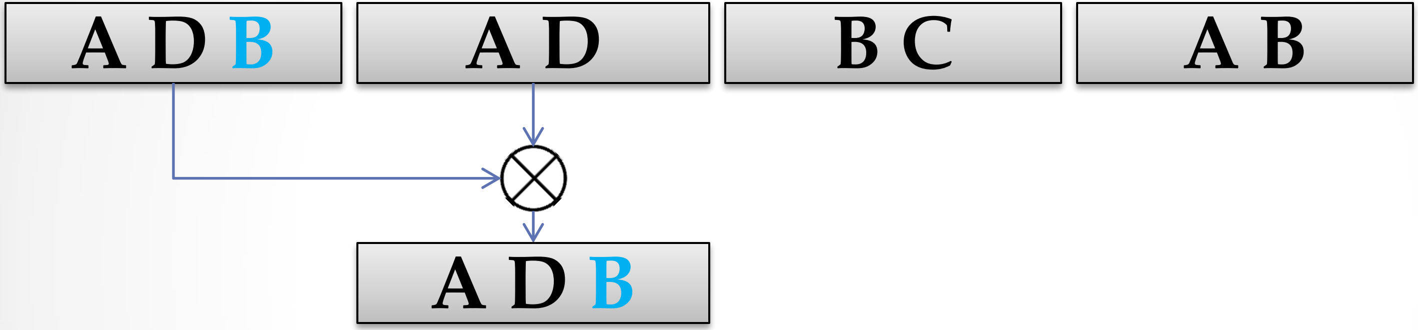
A D B	A D	B C	A B

MTF list at index 4

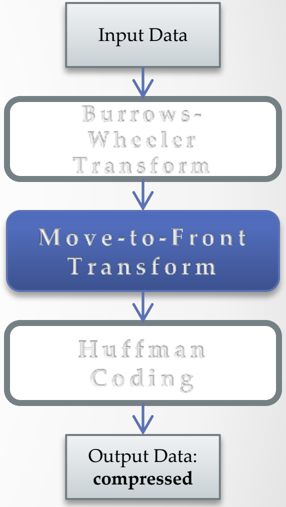


$\otimes$ Operator → Appends all unique symbols to current list

Example Parallel MTF

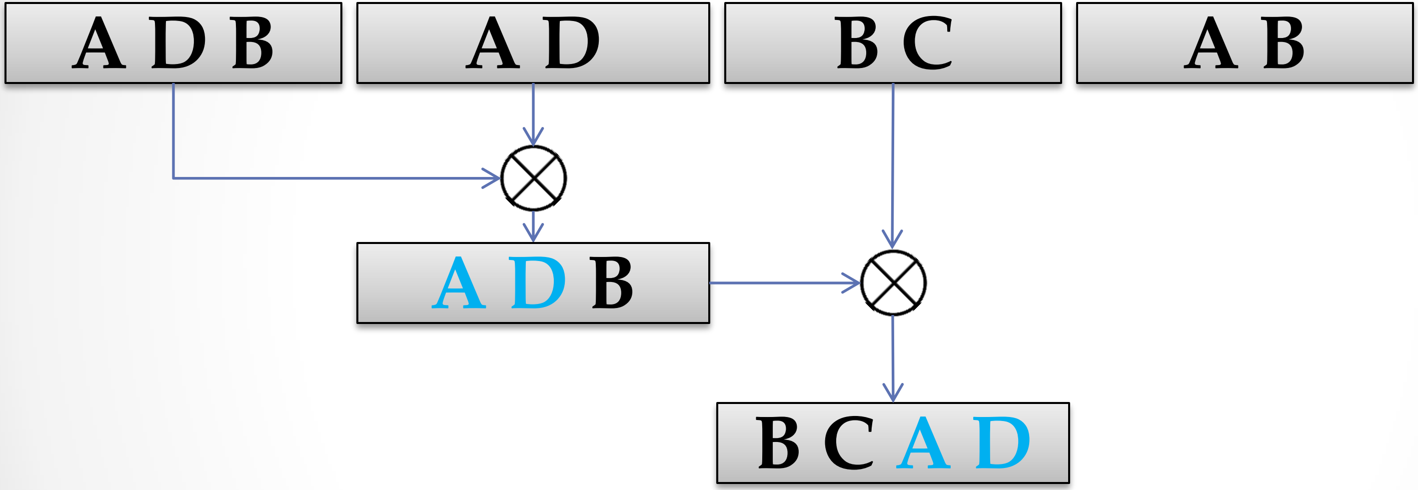


MTF list at index 8

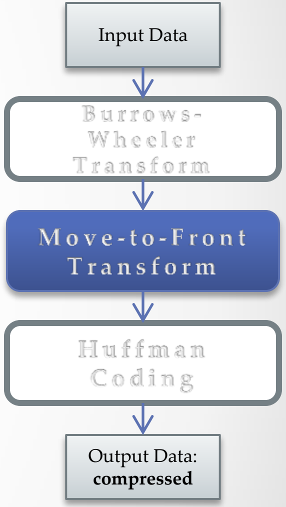


$\otimes$ Operator → Appends all unique symbols to current list

Example Parallel MTF

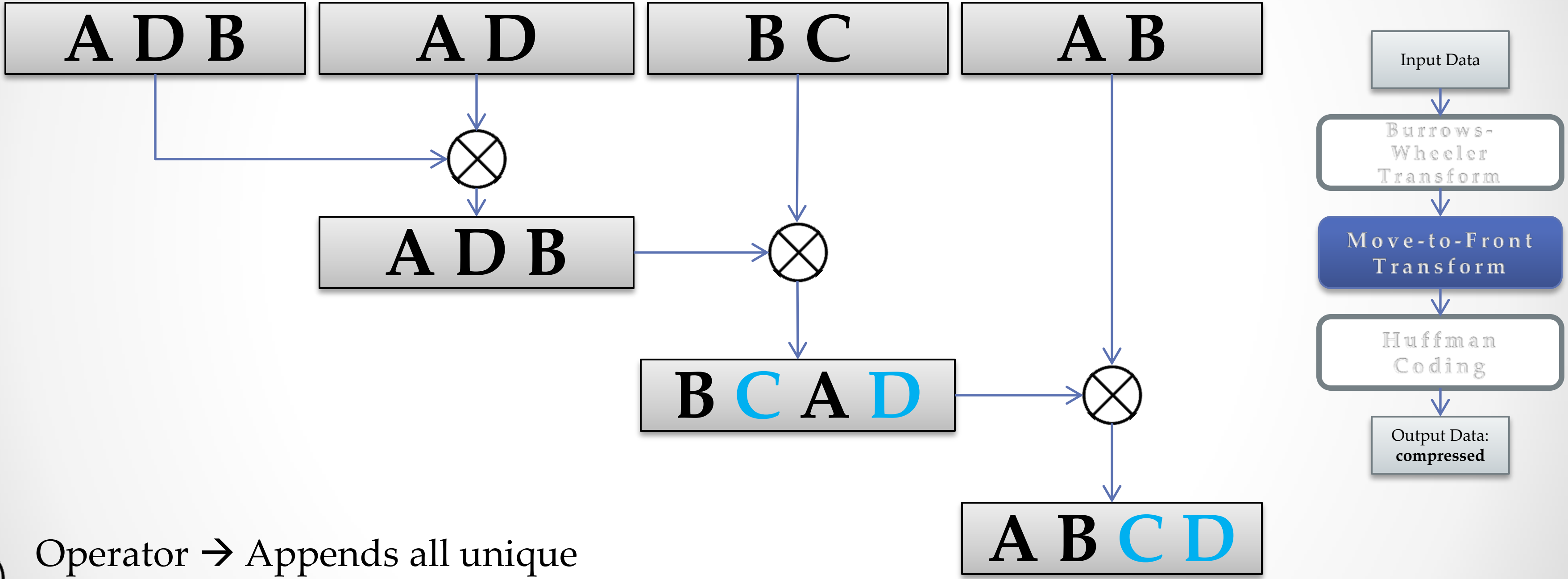


MTF list at index 12



$\otimes$ Operator → Appends all unique symbols to current list

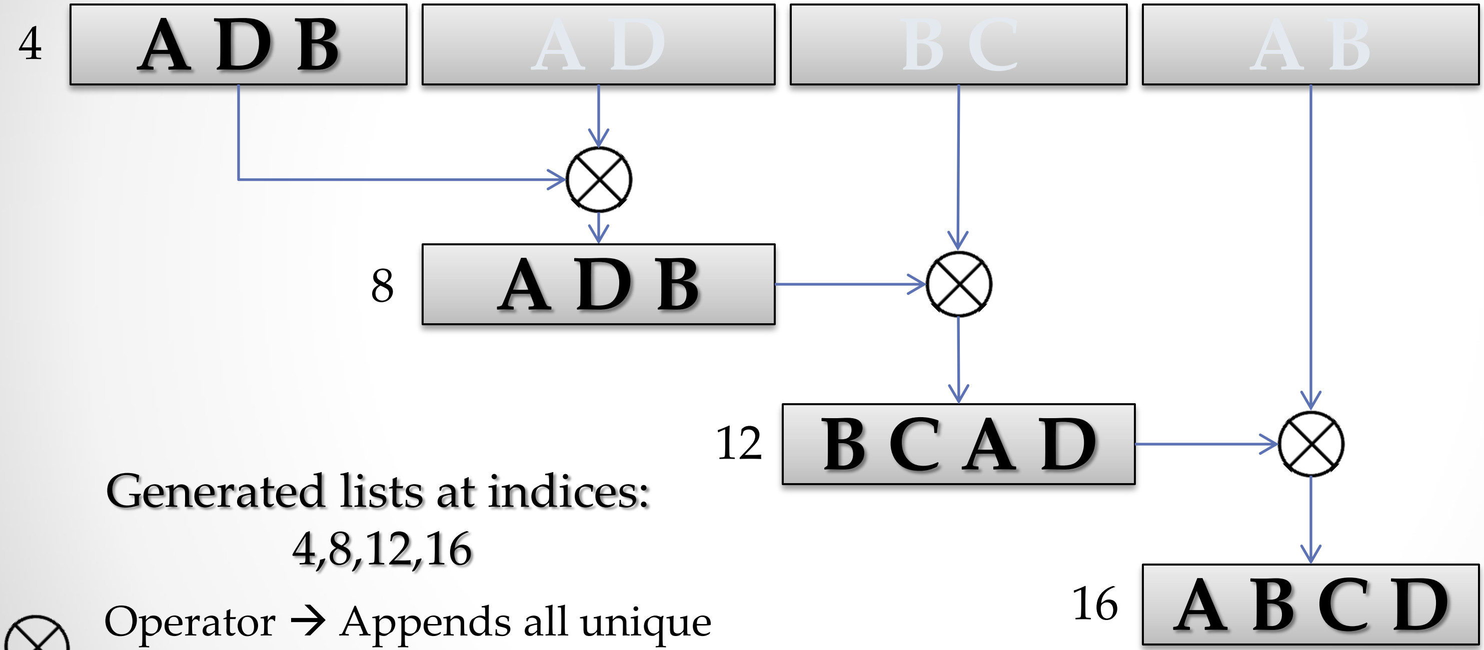
Example Parallel MTF

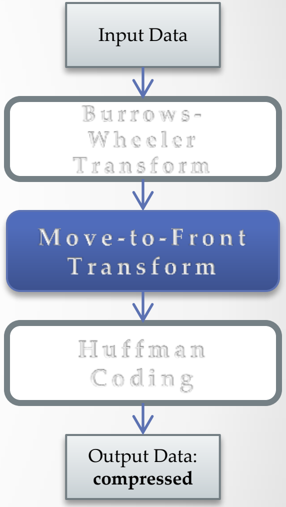


MTF list at index 16

$\otimes$ Operator → Appends all unique symbols to current list

Example Parallel MTF

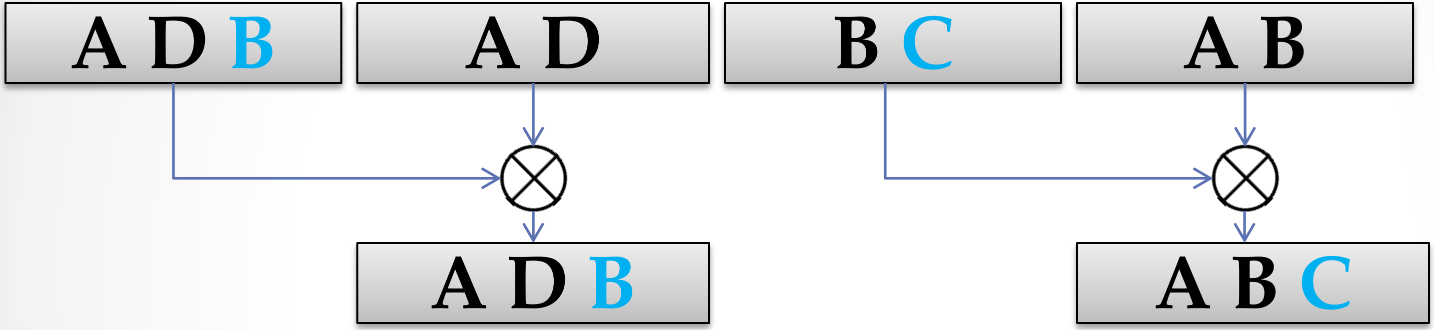


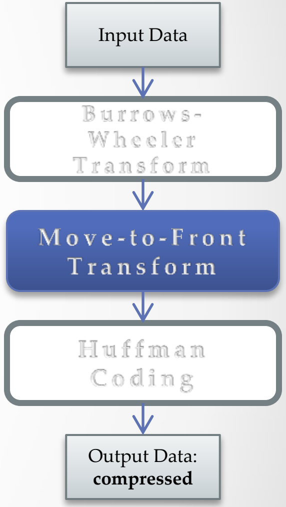


Generated lists at indices:
4,8,12,16

$\otimes$ Operator → Appends all unique symbols to current list

Example Parallel MTF

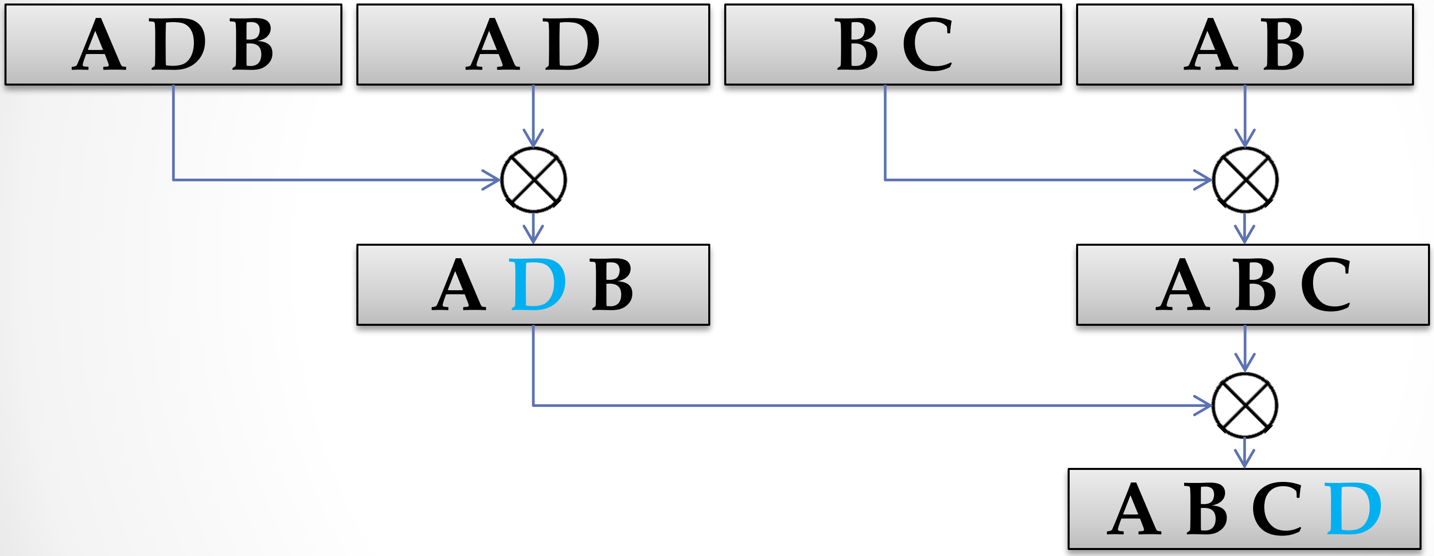


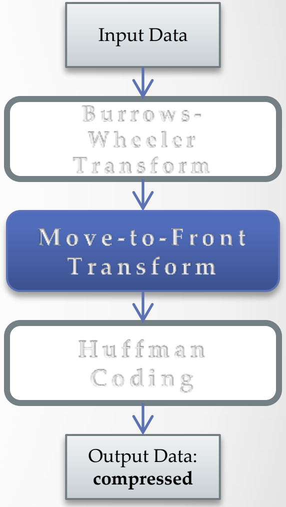


$\otimes$ Operator → Appends all unique symbols to current list

UCDAVIS
UNIVERSITY OF CALIFORNIA

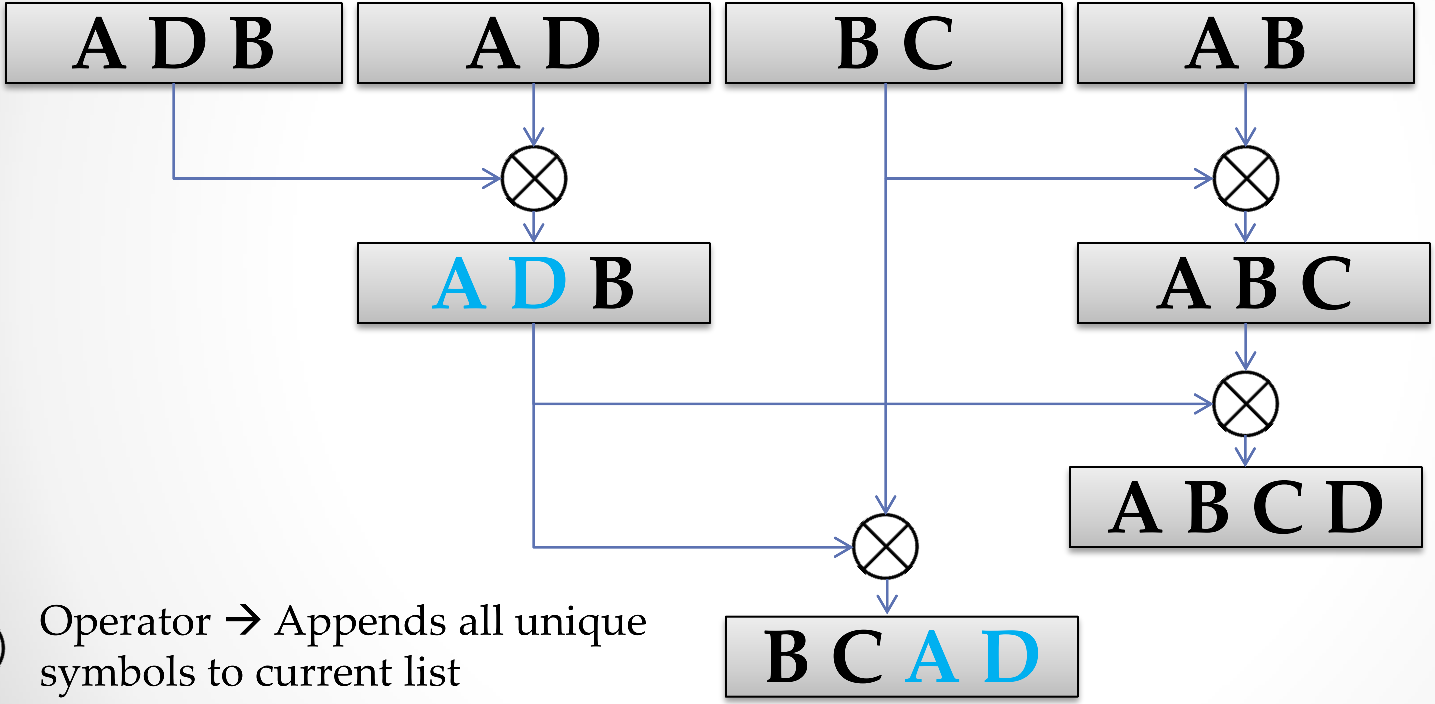
Example Parallel MTF

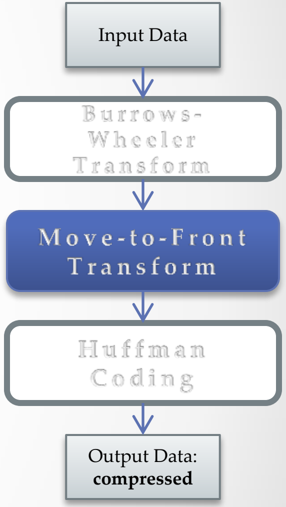




$\otimes$ Operator → Appends all unique symbols to current list

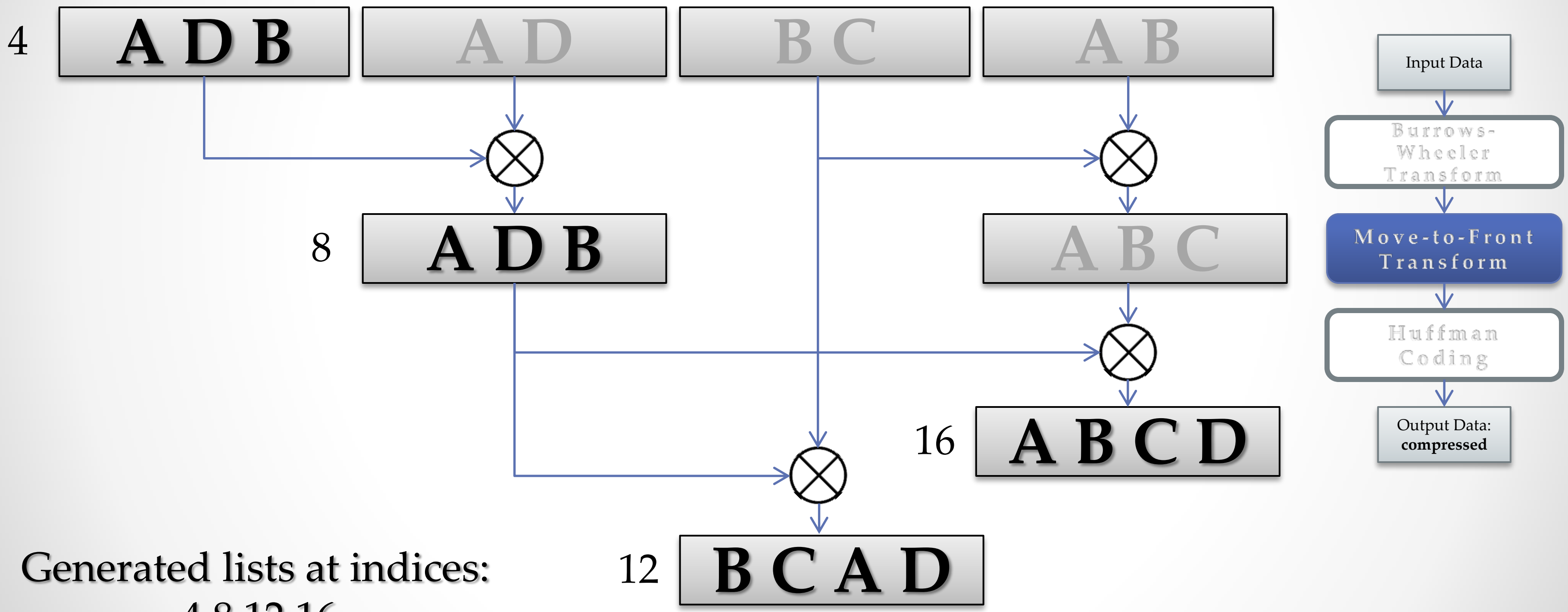
Example Parallel MTF





$\otimes$ Operator → Appends all unique symbols to current list

Example Parallel MTF



Generated lists at indices:
4,8,12,16

UCDAVIS
UNIVERSITY OF CALIFORNIA

Example Parallel MTF

B A D A D D D A C C C B B A A A

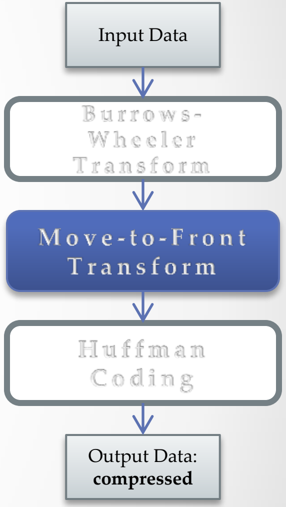
	0	4	8	12
In	B A D A	D D D A	C C C B	B A A A
MTF Lists	A B C…	A D B…	A D B…	B C A D…
	(Initial MTF List)			

Input Data → Burrows-Wheeler Transform → Move-to-Front Transform → Huffman Coding → Output Data: **compressed**

Example Parallel MTF

B A D A D D D A C C C B B A A A

	0	4	8	12
In	B A D A	D D D A	C C C B	B A A A
MTF Lists	A B C… (Initial MTF List)	A D B…	A D B…	B C A D…
MTF Out	[1,1,3,1]	[1,0,0,1]	[3,0,0,3]	[0,2,0,0]



So far…

`ababacabac` — Input

↓

Burrows-Wheeler Transform

↓

`ccbbbaaaaa` — Grouped characters

↓

Move-to-Front Transform

↓

`99,0,99,0,0,99,0,0,0,0` — Lots of 0s

↓

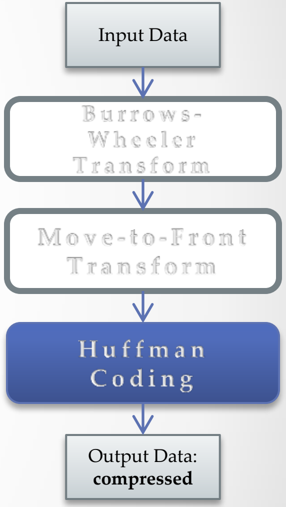
Huffman Coding

↓

Output Data: **compressed**

Huffman Coding

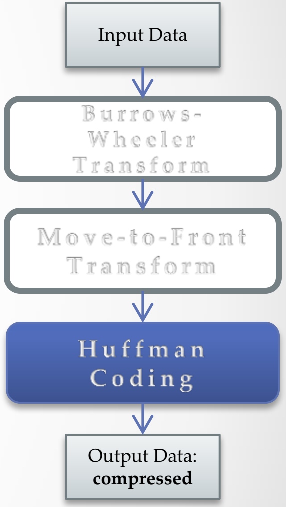
- Final stage that performs actual compression
- Replace each character with a bit code
- Characters that occur more often get shorter codes
- Huffman Coding is more effective with repetitive data (after MTF)



Huffman Coding

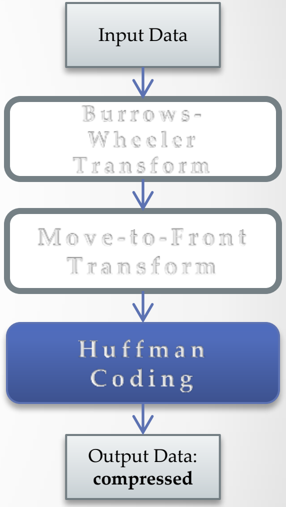
"HUFF" → 00 01 1 1

32 bits → 6 bits



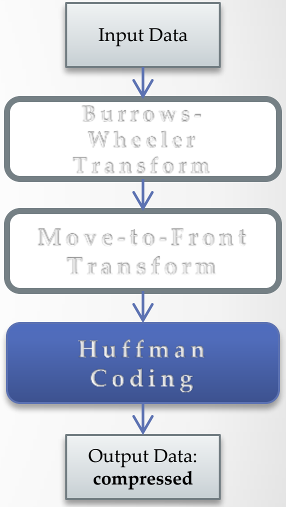
Huffman Coding

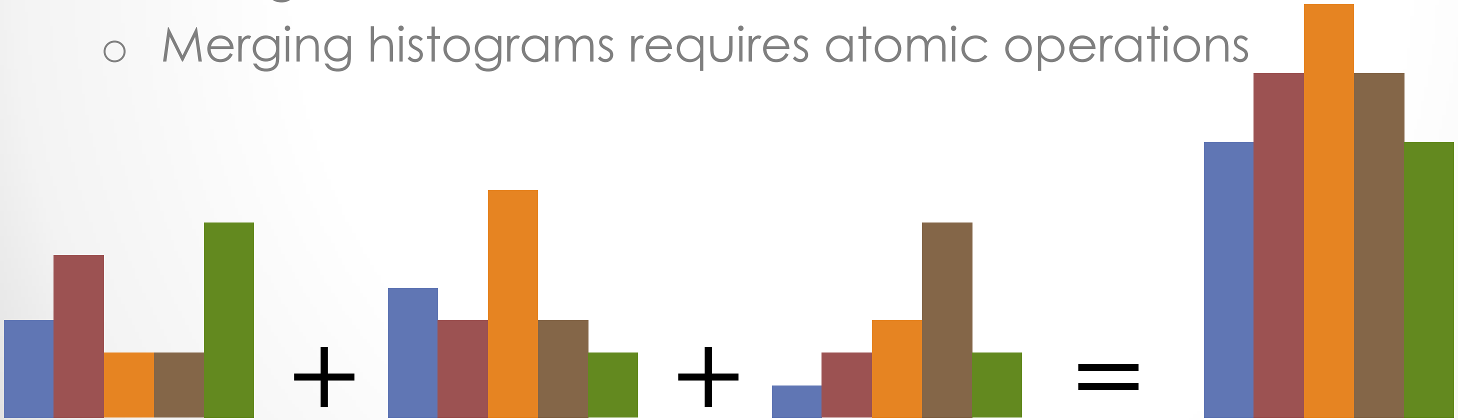
- Huffman Stages:
  1. Generate 256-bin histogram
  2. Build Huffman tree
  3. Replace characters with codes



Huffman Histogram

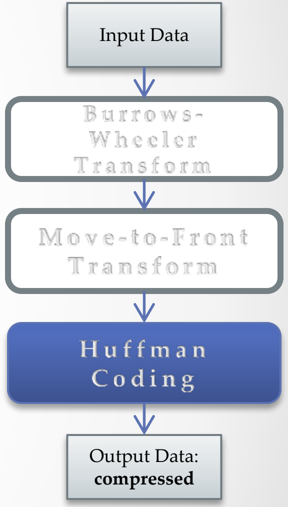
- Histogram
  - Build a 256-entry histogram to count characters
  - To do this on the GPU, divide the input data among threads with each thread maintaining its own histogram
  - Merging histograms requires atomic operations





Huffman Tree

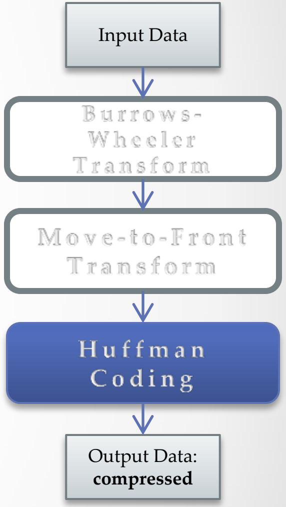
- Huffman Tree Algorithm
  - Remove two lowest counts, combine to form composite node
  - Composite node “inserted” into histogram with combined counts
  - Each step is dependent on the previous step
  - Parallelize finding the two lowest counts (maximum 256-way parallelism)



Huffman Tree Algorithm

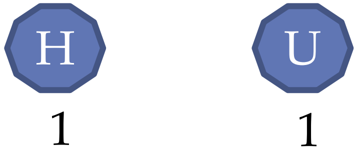
<u>Histogram</u>
H: 1
U: 1
F: 2

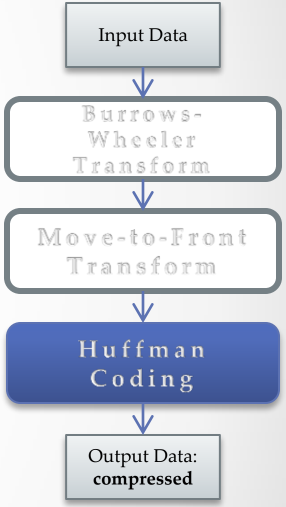
“HUFF”



Huffman Tree Algorithm

Histogram
H: 1
U: 1
F: 2

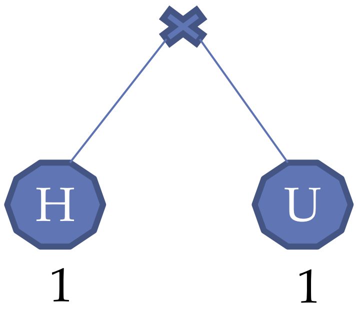


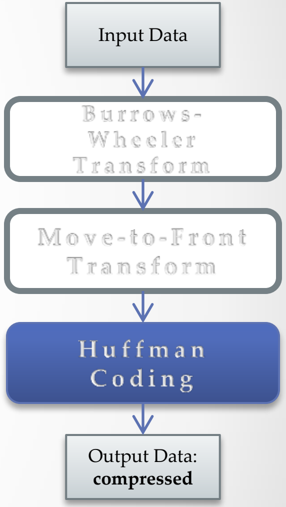


Huffman Tree Algorithm

<u>Histogram</u>

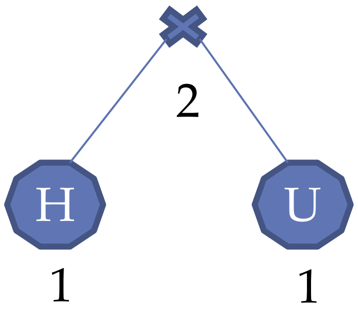
F: 2

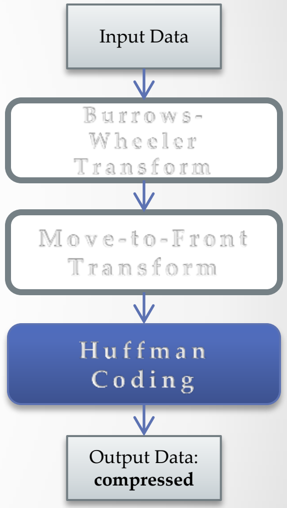




Huffman Tree Algorithm

Histogram
H+U: 2
F: 2



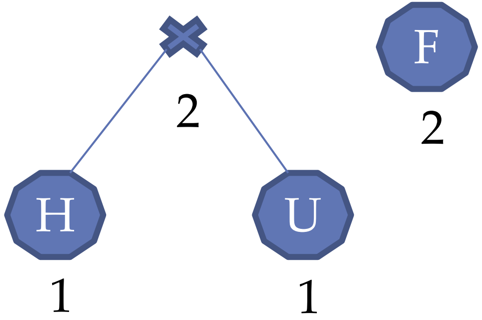


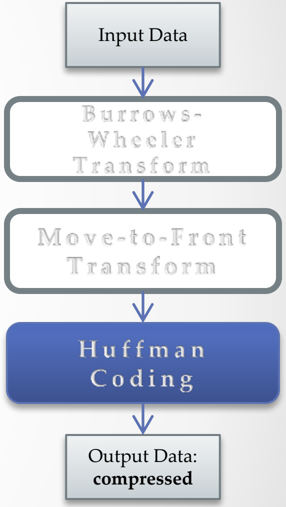
Huffman Tree Algorithm

<u>Histogram</u>

H+U: 2

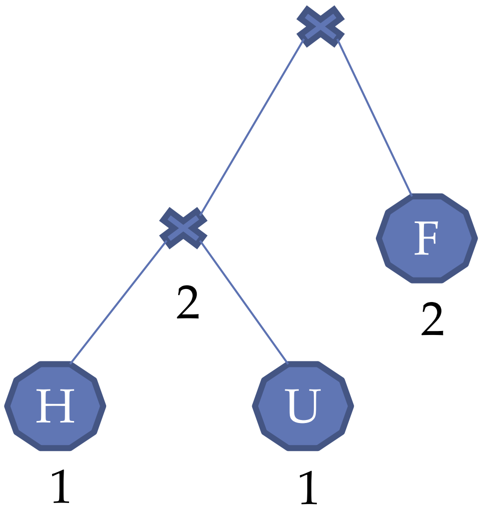
F: 2

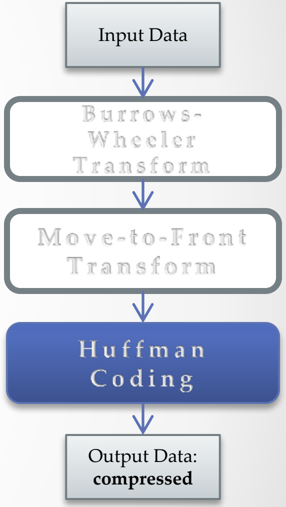




Huffman Tree Algorithm

Histogram
(empty)





Huffman Tree Algorithm

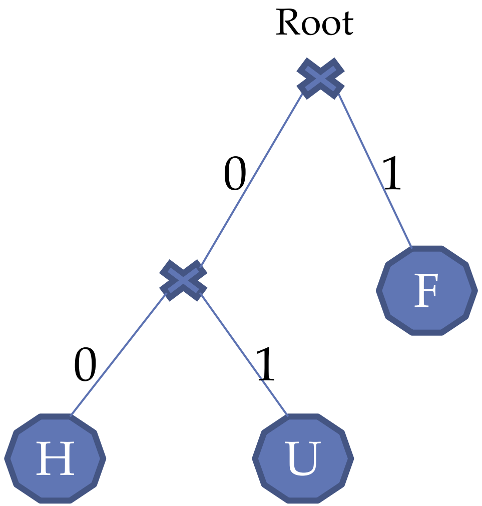
Left → 0 Right → 1

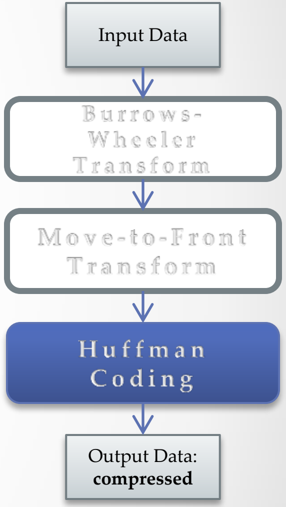
<u>Huffman Codes</u>

H: 00

U: 01

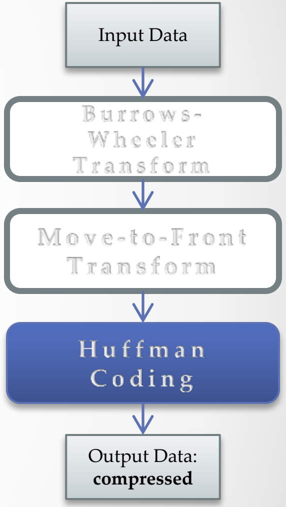
F: 1





Huffman Coding

- Huffman coding assigns **prefix codes**
- No codes share the same prefix



<u>Huffman Codes</u>

H: 00

U: 01

F: 1

UCDAVIS
UNIVERSITY OF CALIFORNIA

Huffman Coding

<u>Huffman Codes</u>

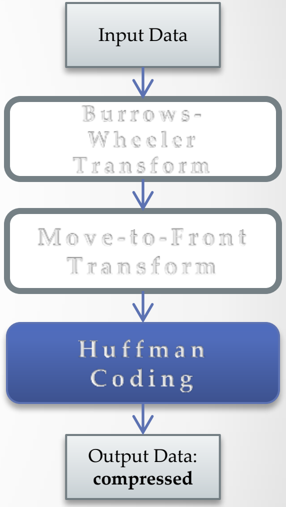
H: 00

U: 01

F: 1

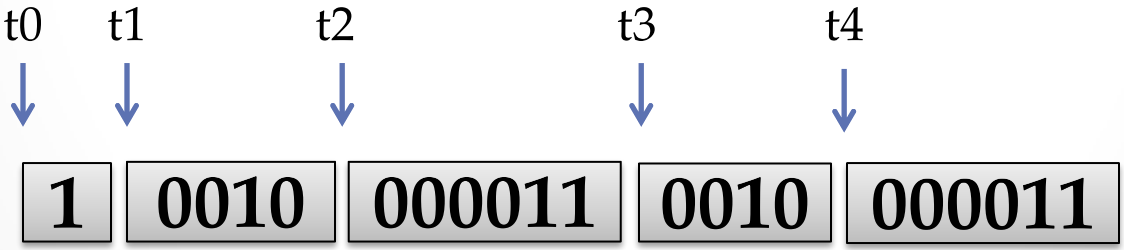
“HUFF” → 000111

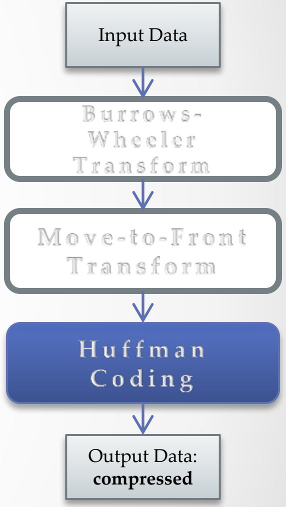
32 bits → 6 bits



Huffman Coding

- Replace characters with codes
  - To do in parallel, divide input among threads
  - Hard to do on the GPU because codes are variable-length
  - Each thread must calculate the correct offset to begin writing its code





Results

- GPU vs. Bzip2
  - Overall
    - GPU **2.78x slower**
  - BWT
    - GPU 2.9x slower (91% of runtime)
  - MTF + Huffman
    - GPU 1.34x slower

Results

Benchmark results

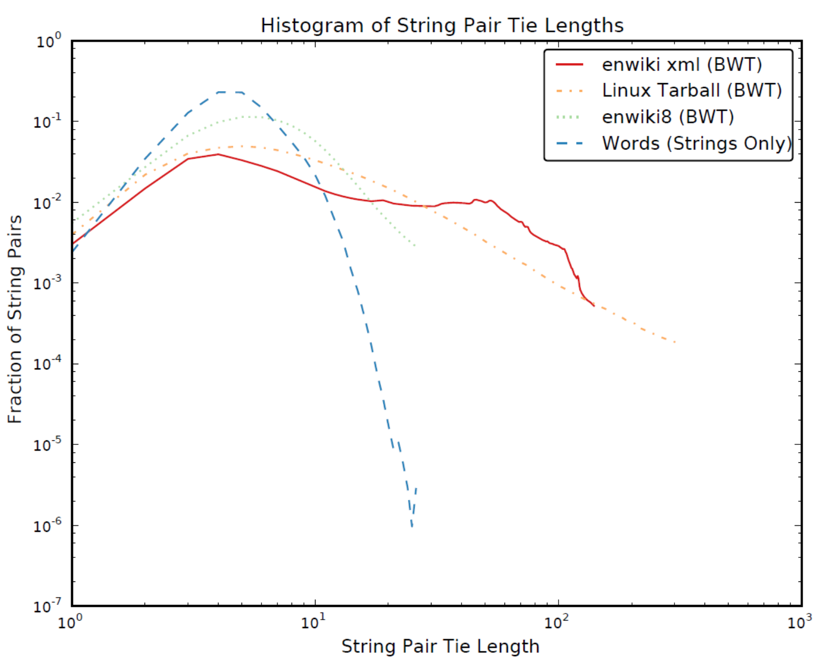
File (Size)	Compress Rate (MB/s)	BWT Sort Rate (Mstrings/s)	MTF+Huffman Rate (MB/s)	Compress Ratio $\frac{\textit{Compressed Size}}{\textit{Uncompressed Size}}$
Text (97 MB)	GPU: 7.37 bzip2: 10.26	GPU: 9.84 bzip2: 14.2	GPU: 29.4 bzip2: 33.1	GPU: 0.33 bzip2: 0.29
Source Code (203 MB)	GPU: 4.25 bzip2: 9.8	GPU: 4.71 bzip2: 12.2	GPU: 44.3 bzip2: 48.8	GPU: 0.24 bzip2: 0.18
XML (151 MB)	GPU: 1.42 bzip2: 5.3	GPU: 1.49 bzip2: 5.4	GPU: 32.6 bzip2: 69.2	GPU: 0.19 bzip2: 0.10

Worst performance during string sort and overall

Best amount of compression

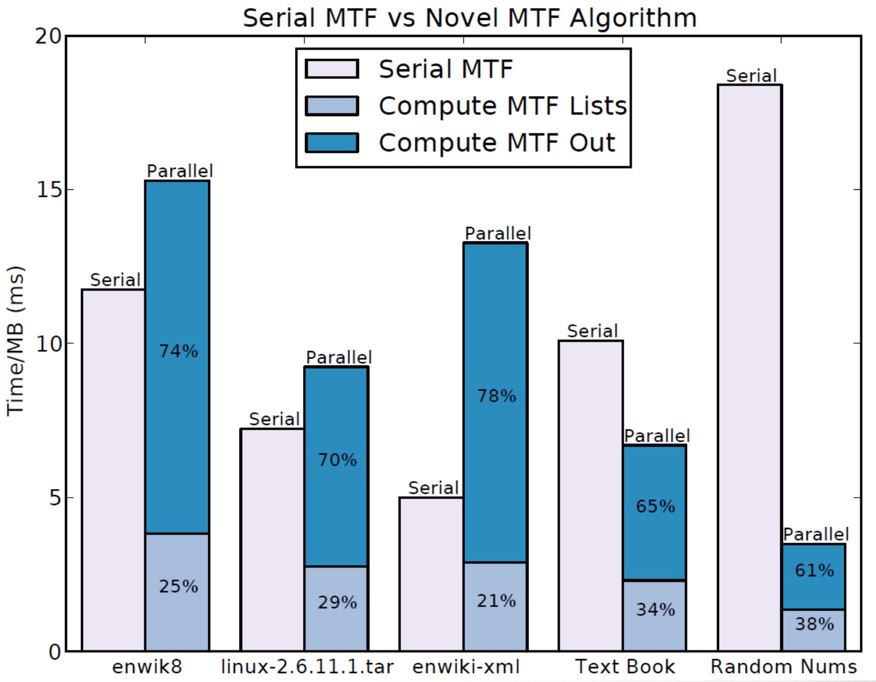
Results

- BWT Performance
  - GPU – 91% of runtime
  - Bzip2 – 81% of runtime
  - Tie Analysis:
    - Amount of compression is data dependent
    - Better compression leads to longer ties and poor performance



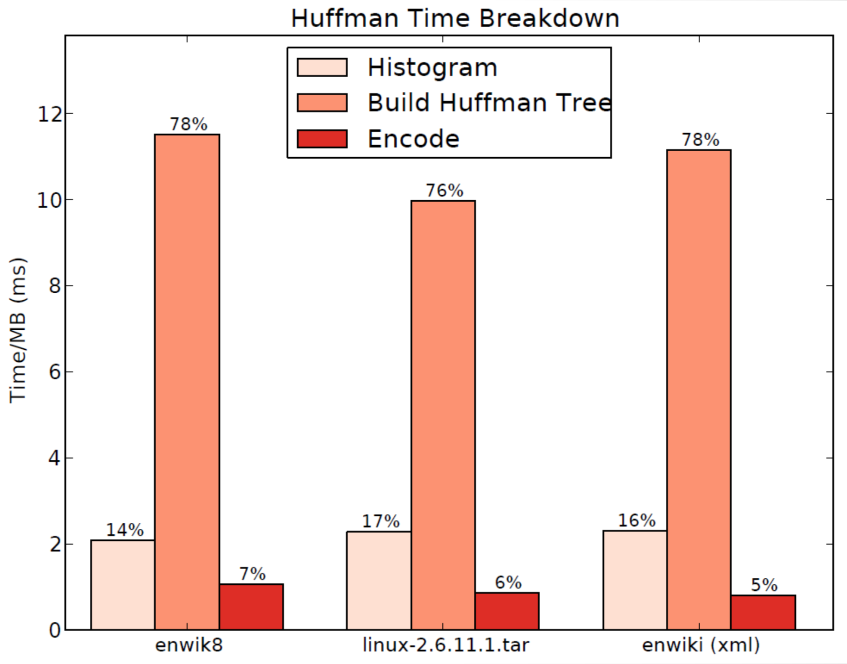
Results

- MTF Performance
  - Majority of runtime during character replacement
  - Moving element to front, shifting all other elements
  - Thread divergence



Results

- Huffman Performance
  - Majority of runtime during Huffman tree building
  - 256-way parallelism is inadequate for the GPU



UCDAVIS
UNIVERSITY OF CALIFORNIA

Decompression

- Reverse Burrows-Wheeler transform
  - Much faster than forward BWT
  - Requires only 1 character-sort
  - Similar to linked-list traversal
  - Preliminary implementation is an extension of Hillis and Steele’s parallel linked-list traversal algorithm

Decompression

- Reverse Burrows-Wheeler transform
  - Much faster than forward BWT
  - Requires only 1 character-sort
  - Similar to linked-list traversal
  - Preliminary implementation is an extension of Hillis and Steele's parallel linked-list traversal algorithm
- Reverse Move-to-Front transform
  - Parallel approach uses similar algorithm as forward-MTF
    - Generate partial MTF lists
    - Combine adjacent lists

Decompression

- Reverse Burrows-Wheeler transform
  - Much faster than forward BWT
  - Requires only 1 character-sort
  - Similar to linked-list traversal
  - Preliminary implementation is an extension of Hillis and Steele's parallel linked-list traversal algorithm
- Reverse Move-to-Front transform
  - Parallel approach uses similar algorithm as forward-MTF
    - Generate partial MTF lists
    - Combine adjacent lists
- **Huffman decoding**
  - Decode each encoded block in parallel

UCDAVIS
UNIVERSITY OF CALIFORNIA

Future Work

- Explore other GPU-based string sorts that can better handle long strings with many ties
- Develop a Huffman tree building algorithm that has more than 256-way parallelism
- Overlap GPU compression and PCI-Express data transfer

Conclusion

- Implemented parallel lossless data compression on the GPU
- Parallelized BWT, MTF, and Huffman Coding
  - Developed a novel algorithm for MTF
  - BWT string sort contributes to the majority of runtime
- Implementation is slow but may be better suited for future GPU architectures and other parallel environments

Acknowledgements

- NSF grants OCI-1032859 and CCF-1017399
- HP Labs Innovation Research Program
- Discussion and feedback
    - Shubho Sengupta (Intel)
    - Adam Herr (Intel)
    - Anjul Patney (UC Davis)
    - Stanley Tzeng (UC Davis)