Massively-Parallel Lossless Data Decompression

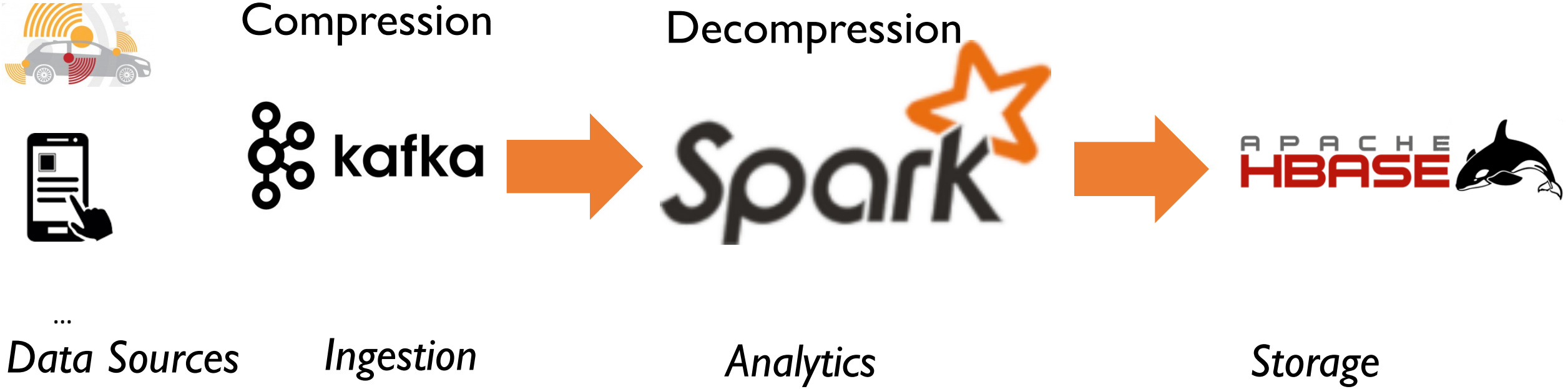
Eva Sitaridi *, Rene Mueller ✤, Tim Kaldewey † , Guy Lohman ✤, Ken Ross *

Columbia University *, IBM Almaden ✤, IBM Watson †

ICPP 2016, Philadelphia

Why **De**compression?

- Big Data workloads
  - Data compressed **once** during *ingestion*
  - Data decompressed **repeatedly** for *analytics & machine learning jobs*



Big data pipeline

Why GPUs?

- Super-computer with accelerators (Source: Top500)

  June 2014: **62** → June 2016: **93**

- **Top** super-computer with accelerators (Source: Green500)
  - June 2015: **32** top positions
  - November 2015: **40** top positions

  } 25% increase

Better "*fuel efficiency*": **If** the problem is parallelizable!

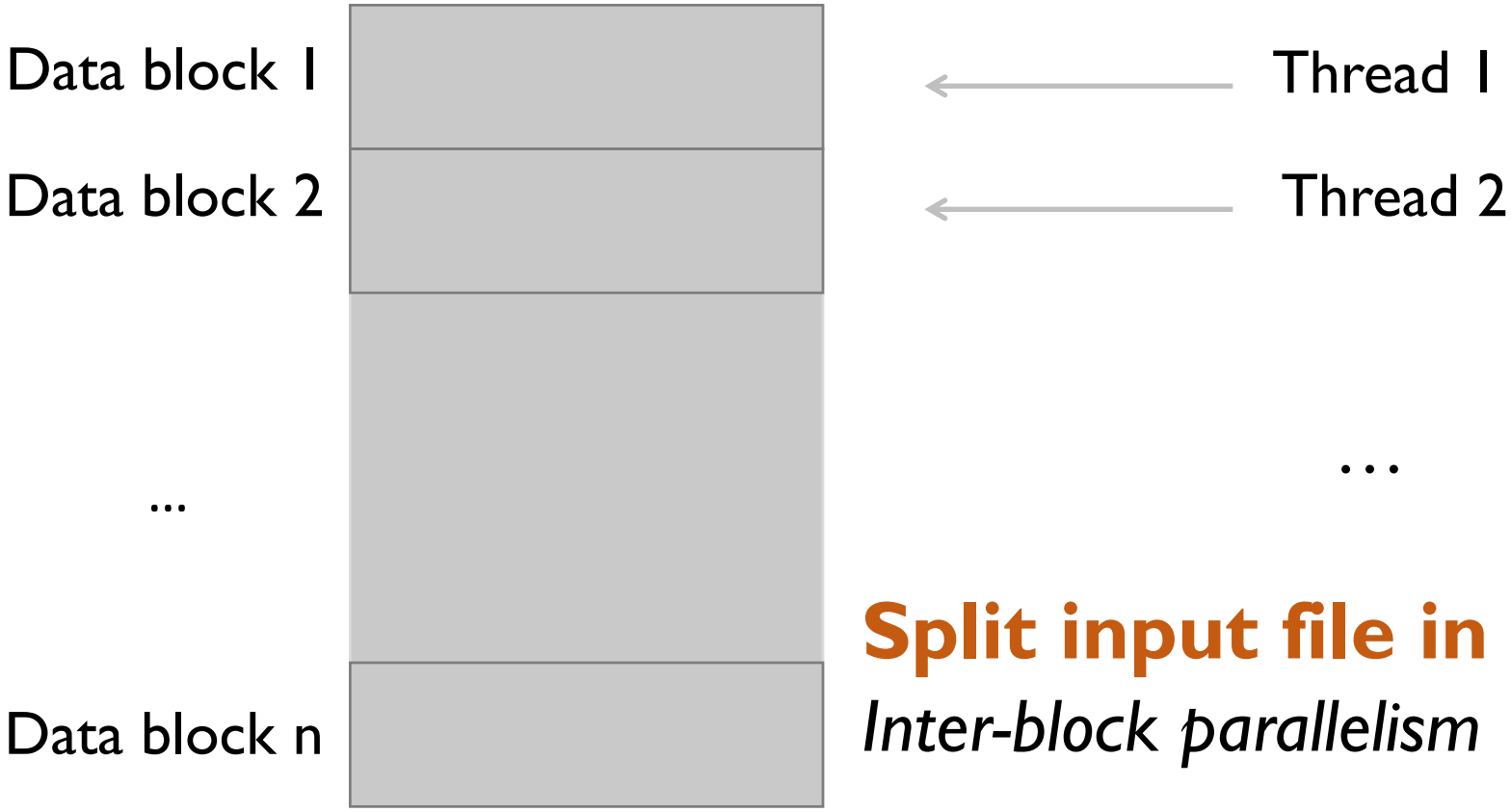
Contribution

Challenge: Data **dependencies** limit parallelism

- Designed alternative methods tackling dependencies for **LZ77**
  a) **MRR** (Multi-Round Resolution)
    - Resolves dependencies by using hardware primitives
  b) **DE** (Dependency Elimination)
    - Eliminates dependencies proactively during compression
    - Achieves 2X faster decompression sacrificing <10% compression ratio
- Implemented **Gompresso**
  - Combines LZ-like compression with Huffman Coding
- Benchmark Gompresso against **multi-core** CPU libraries
  - **2X** faster than multi-core gzip
  - **17%** reduced energy consumption

How to Parallelize Decompression?

>1000 threads available!



Split input file in independent blocks

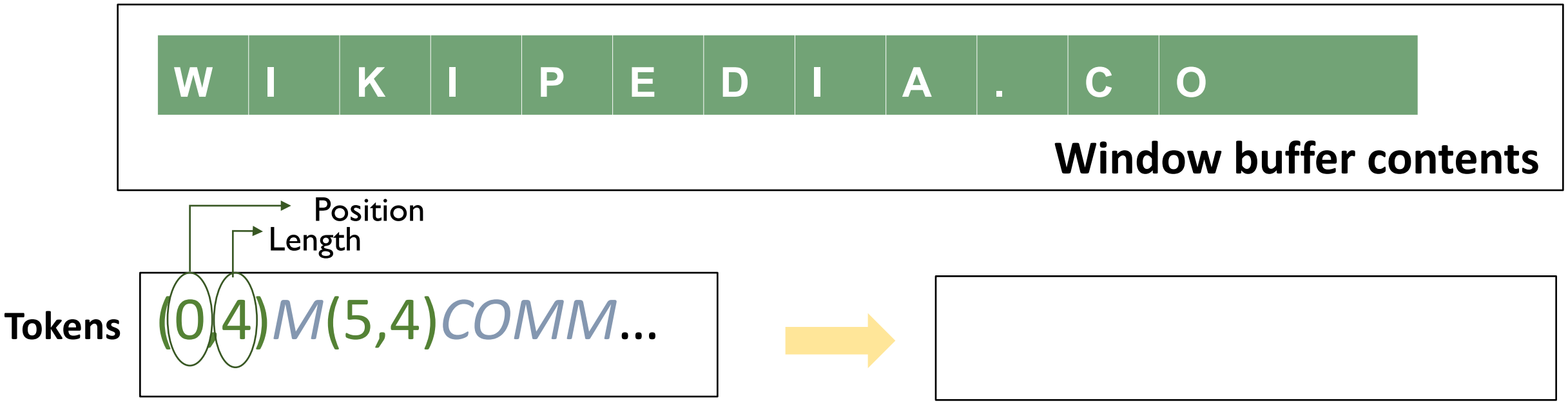
Inter-block parallelism

Naïve approach performance 200 MB/s << 250 GB/s (K20x)

Challenges

- Intra-block parallelism harder
  - Group of threads processing the same data block
  - Data dependencies between threads decompressing a data block
- Balance compression ratio & decompression speed
  - How to trade-off compression ratio for higher parallelism degree?
- Reduce per character cost
  - Writing a single or few characters wastes memory bandwidth

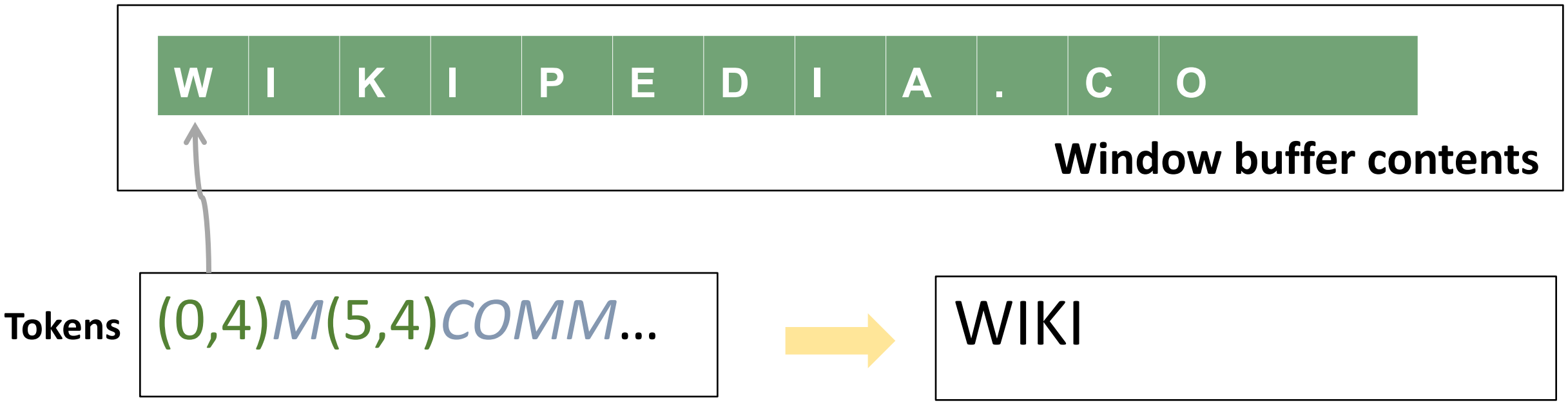
Background: LZ77 Decompression



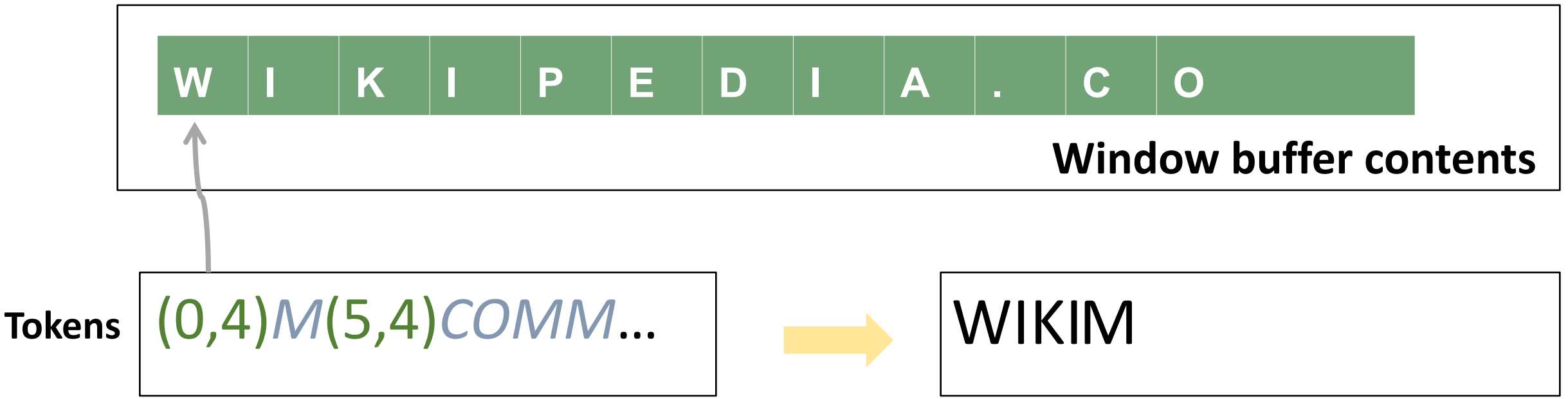
LZ77 Compression: String matching intensive

- Search for the longest match in the recent input

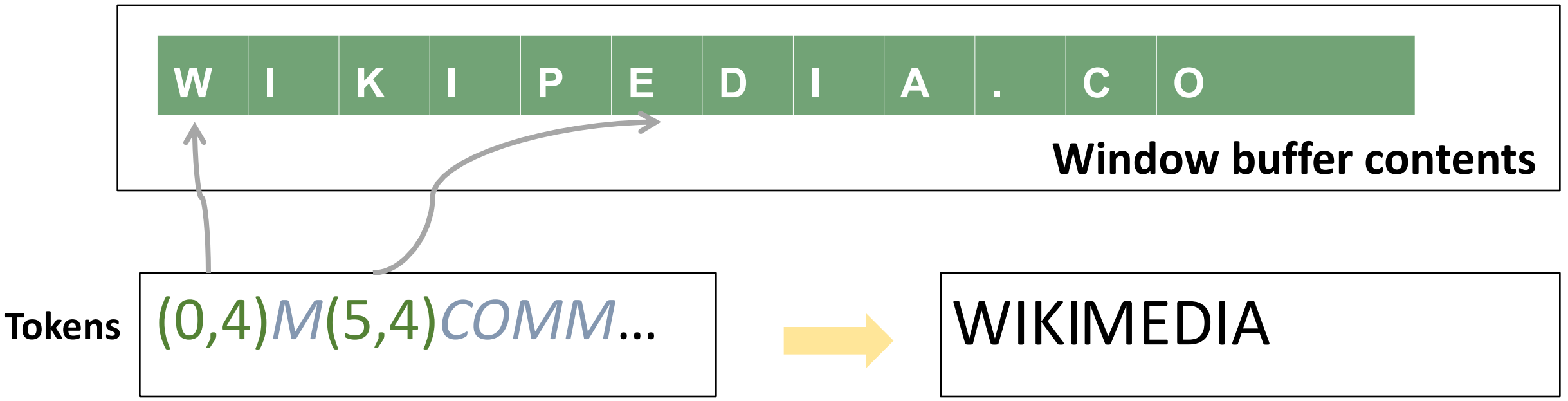
Background: LZ77 Decompression



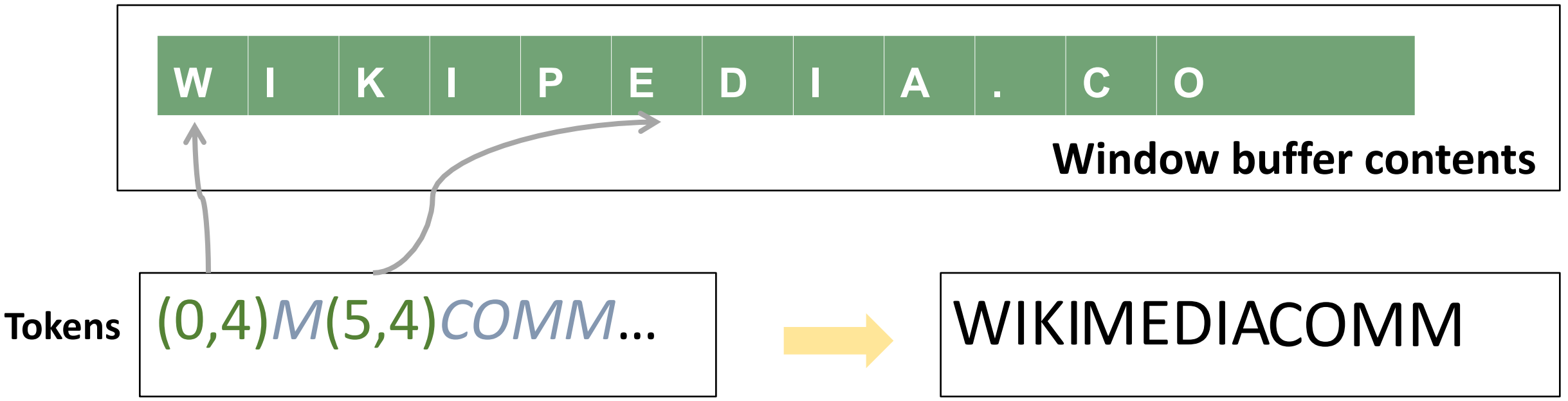
Background: LZ77 Decompression



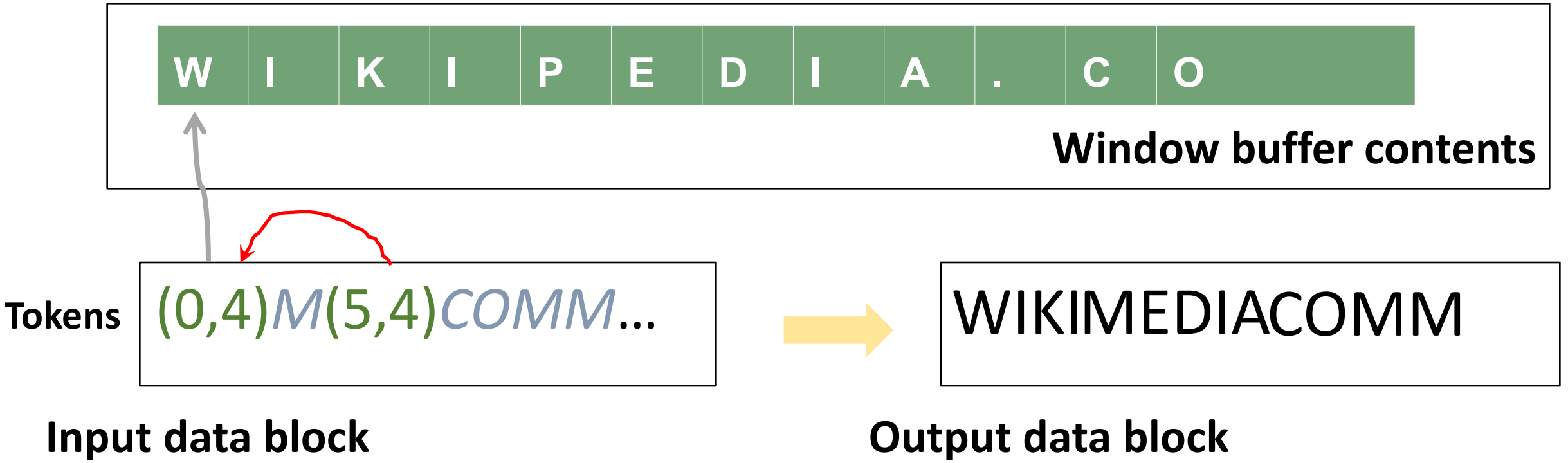
Background: LZ77 Decompression



Background: LZ77 Decompression



Background: LZ77 Decompression

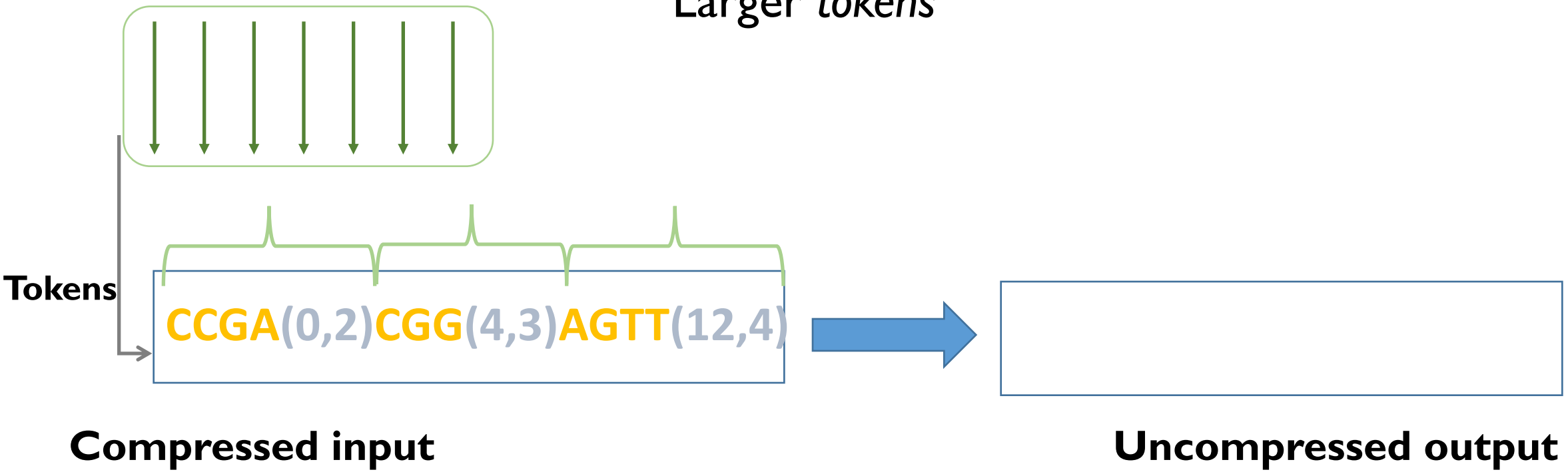


How should threads handle possible dependencies?

MRR Decompression

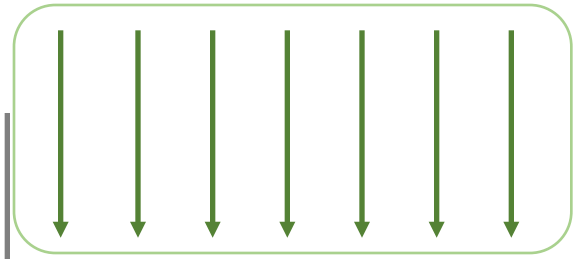
Improve utilization: Group strings of literals with the following back-reference

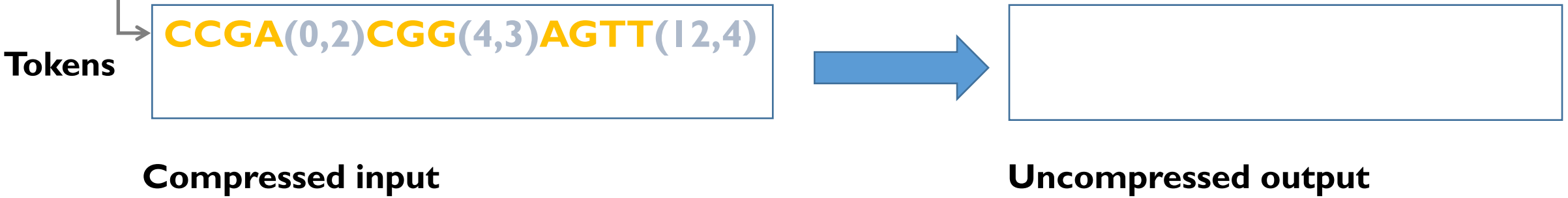
Larger *tokens*



MRR Decompression – Step 1

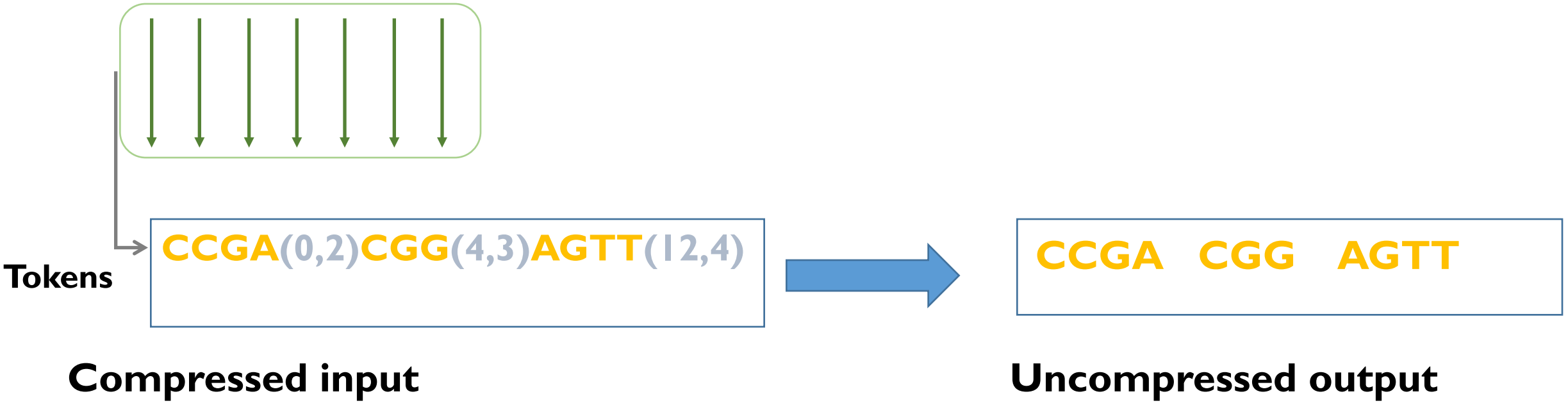
1) ***Read*** tokens (parallel)



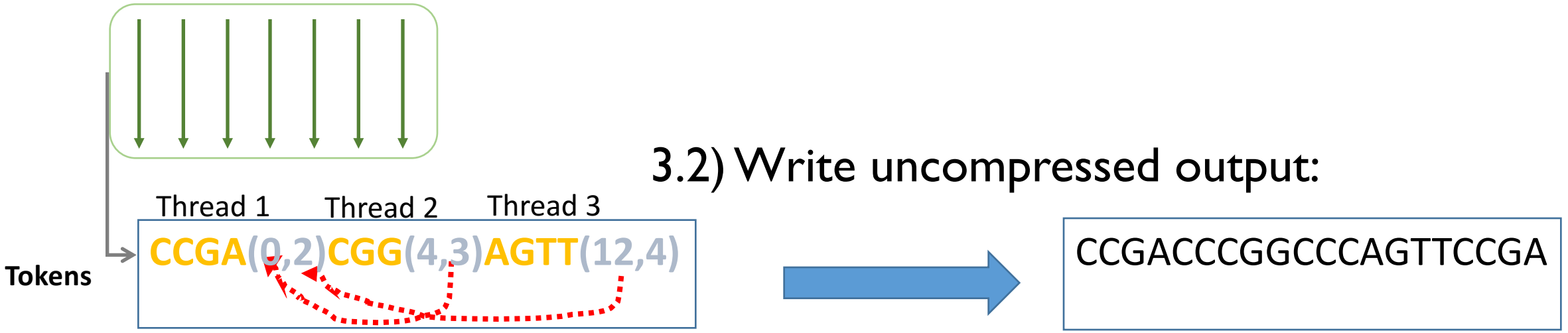


MRR Decompression – Step 2

2) ***Write*** literals (parallel prefix sum)



MRR Decompression – Step 3



Problem: Back-references processed in parallel might be dependent! →

Use voting function __***ballot*** to detect dependencies

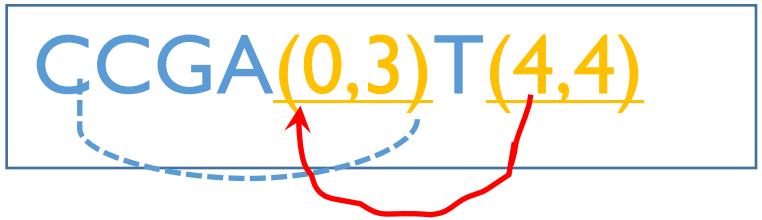
2 rounds of serialization in this example

DE: Dependency Elimination during Compression

…CCGACCGTCCGT…

Uncompressed input

MRR



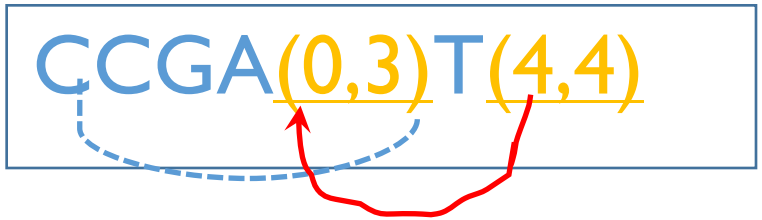
Longest match

DE: Dependency Elimination during Compression

…CCGACCGTCCGT…

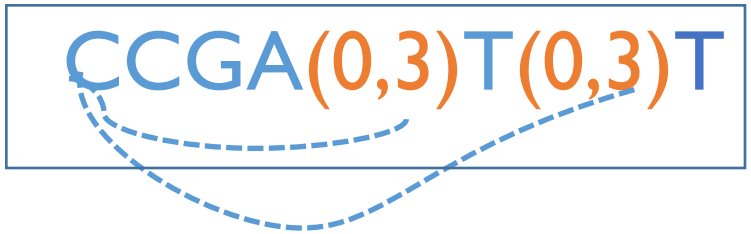
Uncompressed input

MRR



Longest match

DE



*Longest match **not** causing dependencies*

A) Compression

Only search for matches *w/o dependencies*

B) Decompression

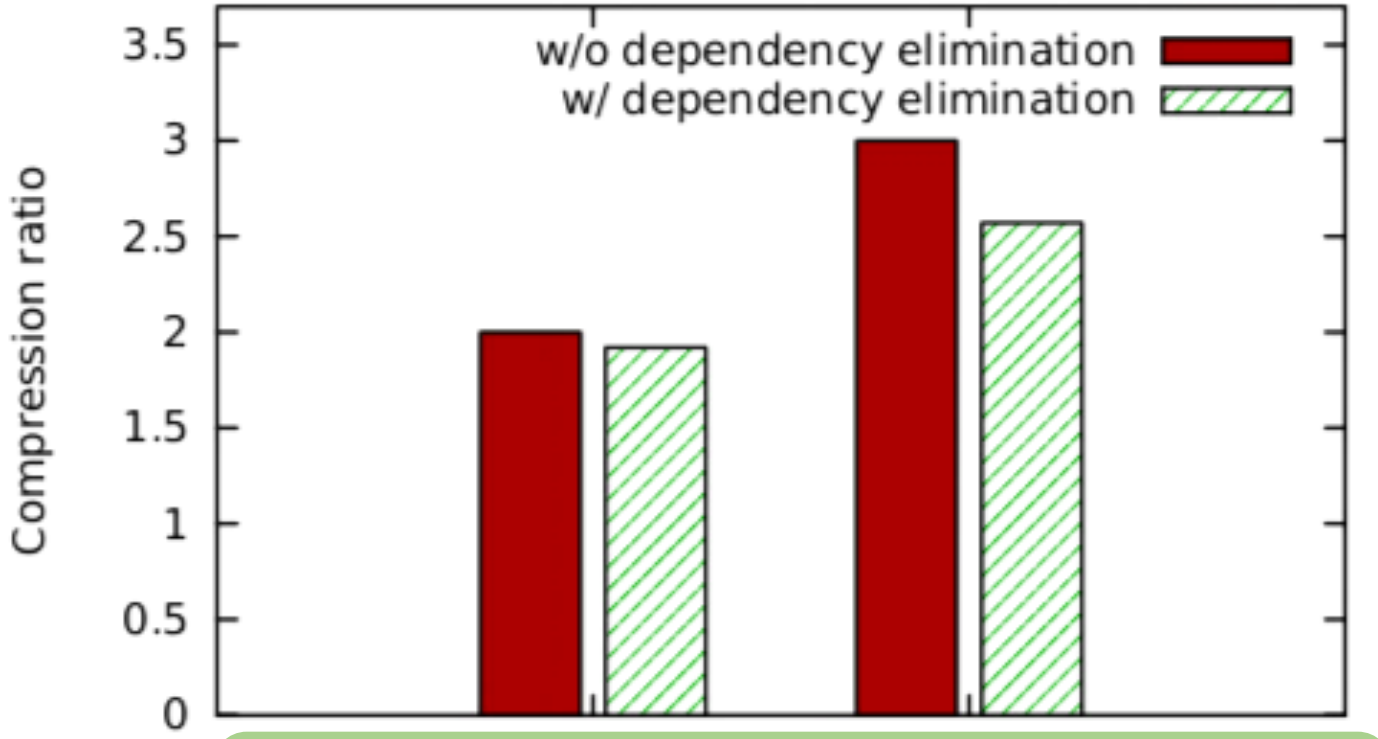
Copy back-references (*fully parallel*)

DE Overhead

Datasets

Wikipedia: 1 GB XML dump of English Wikipedia

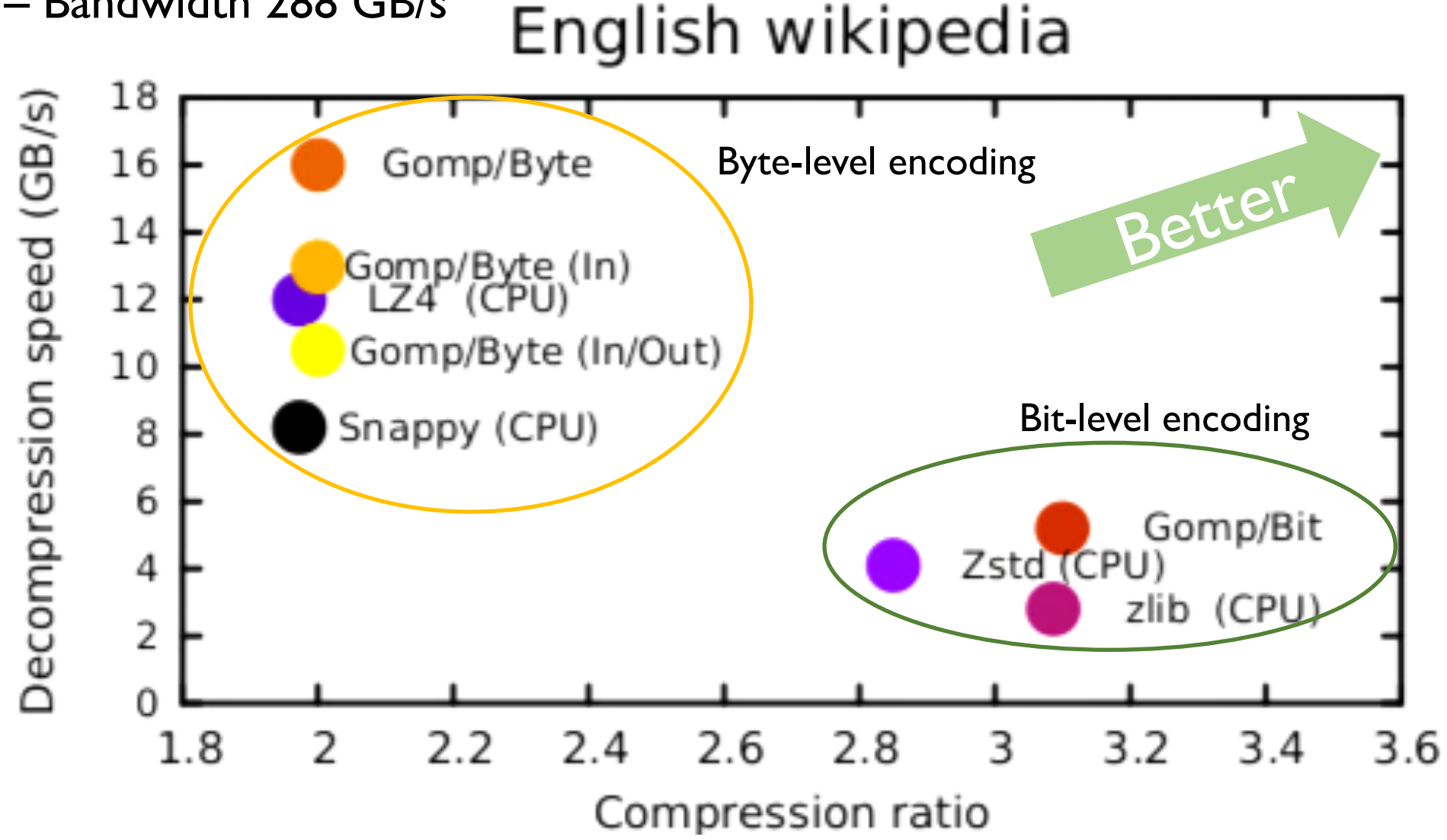
Matrix: 0.77GB Sparse Matrix in Market File format



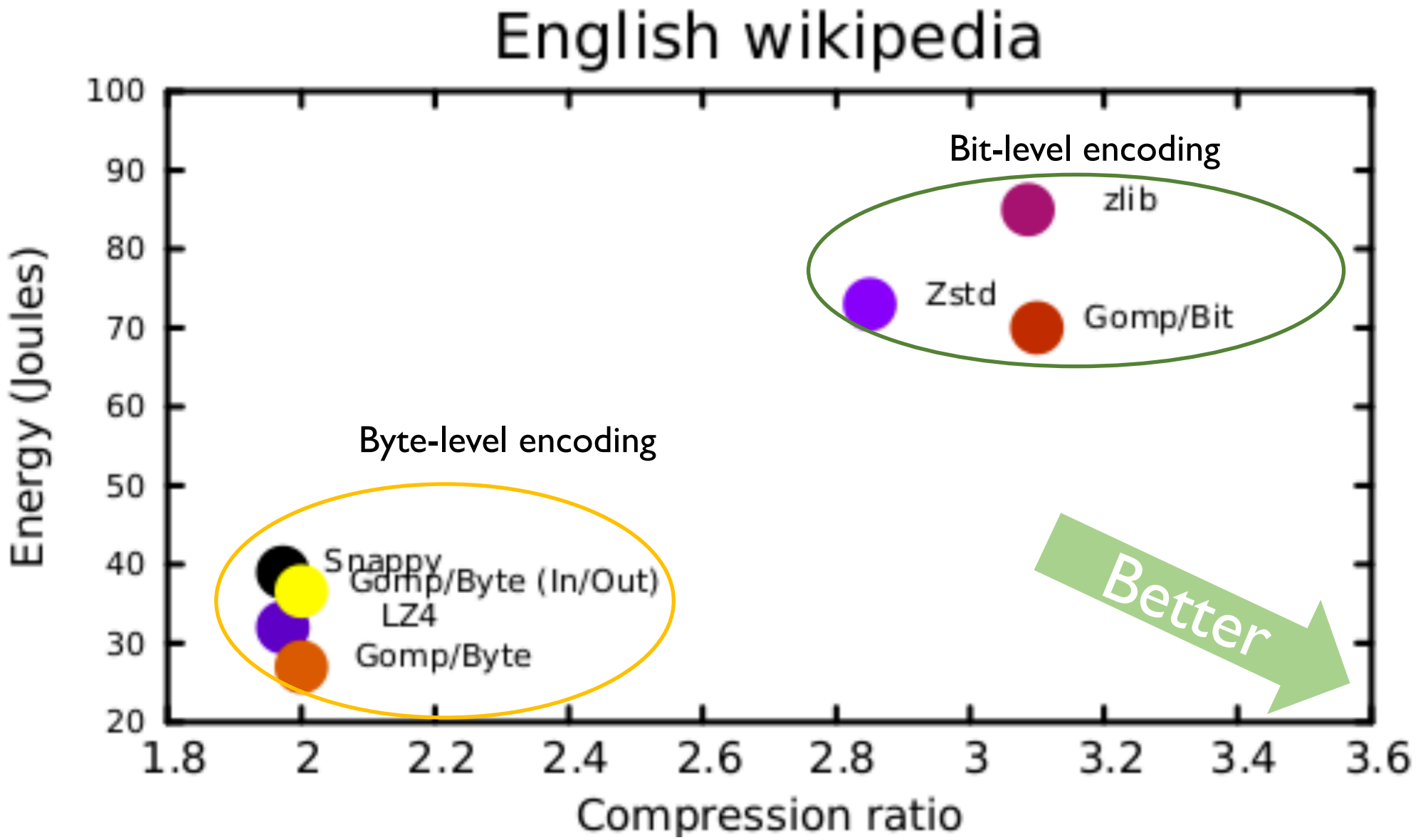
- <19% in compression ratio
- <13% in compression speed

Decompression Speed vs. Compression Ratio

CPU: Dual-socket E5-2620 – Bandwidth 102.4 GB/s
GPU: Tesla K40 – Bandwidth 288 GB/s



Consumed Energy vs. Compression Ratio



Summary & Future Work

- Designed & integrated parallel methods in Gompresso
  - Parallel LZ77 decompression
    - MRR: Iteratively resolve back-reference dependencies
    - DE: Proactively eliminate dependencies during compression
  - Parallel Huffman decoding
- Gompresso vs. multi-core CPU libraries
  - 2X faster decompression
  - <10% penalty in compression ratio
  - 17% reduced energy
- Future work: Investigate alternative coding algorithms

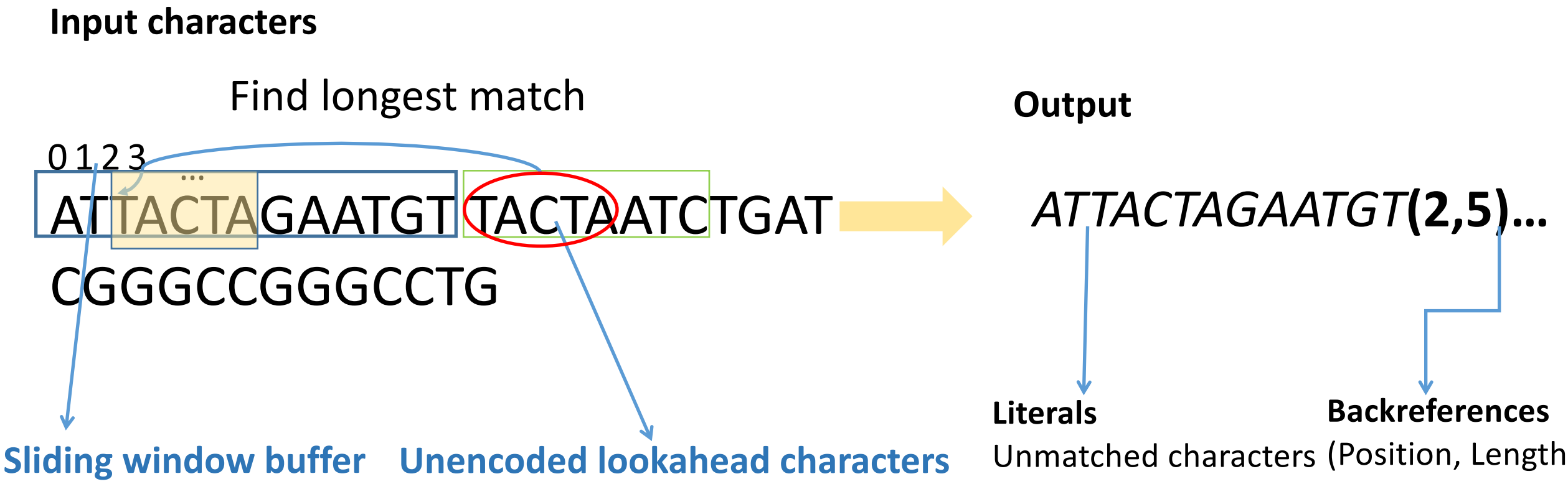
Summary & Future Work

Thank you!

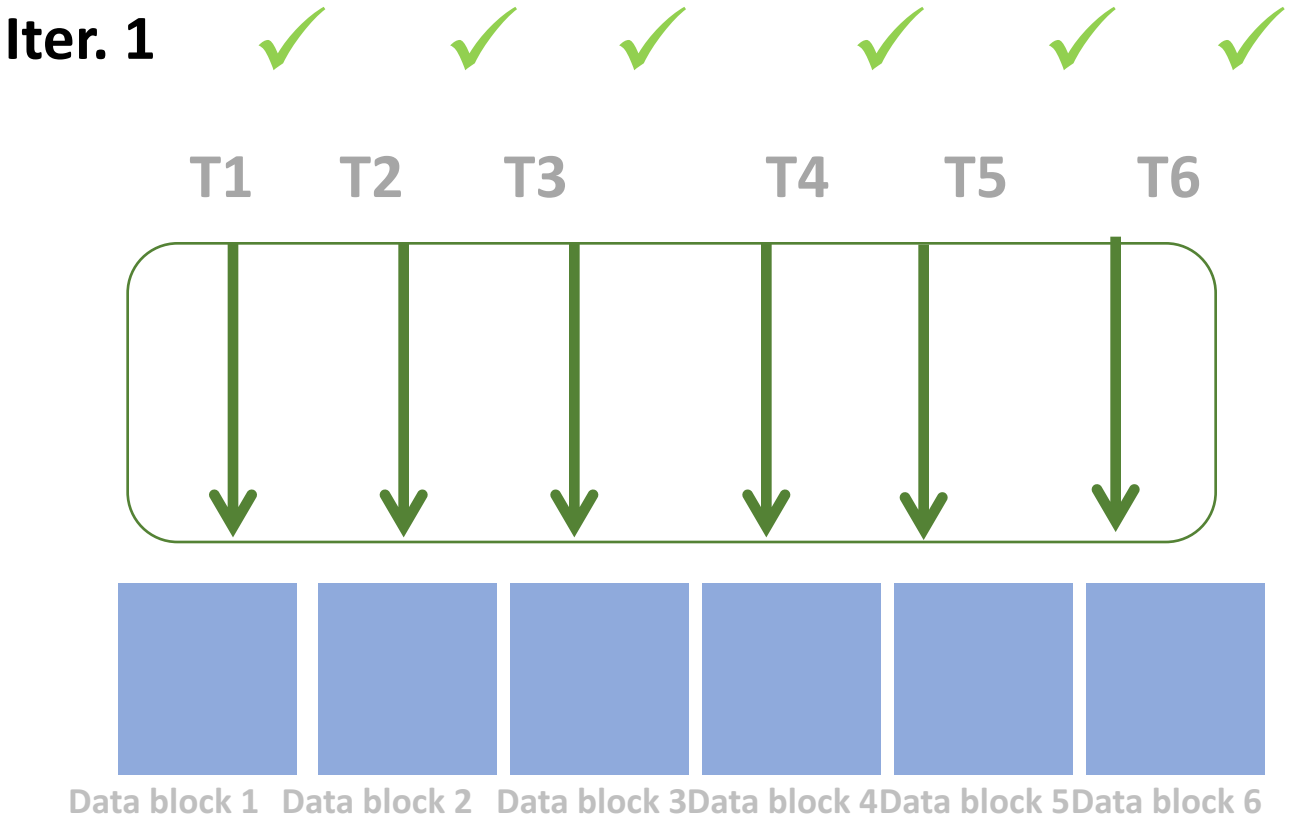
- Designed & integrated parallel methods in Gompresso
  - Parallel LZ77 decompression
    - MRR: Iteratively resolve back-reference dependencies
    - DE: Proactively eliminate dependencies during compression
  - Parallel Huffman decoding
- Gompresso vs. multi-core CPU libraries
  - 2X faster decompression
  - <10% penalty in compression ratio
  - 17% reduced energy
- Future work: Investigate alternative coding algorithms

Back-up Slides

Background: LZ77 Compression



Thread Utilization



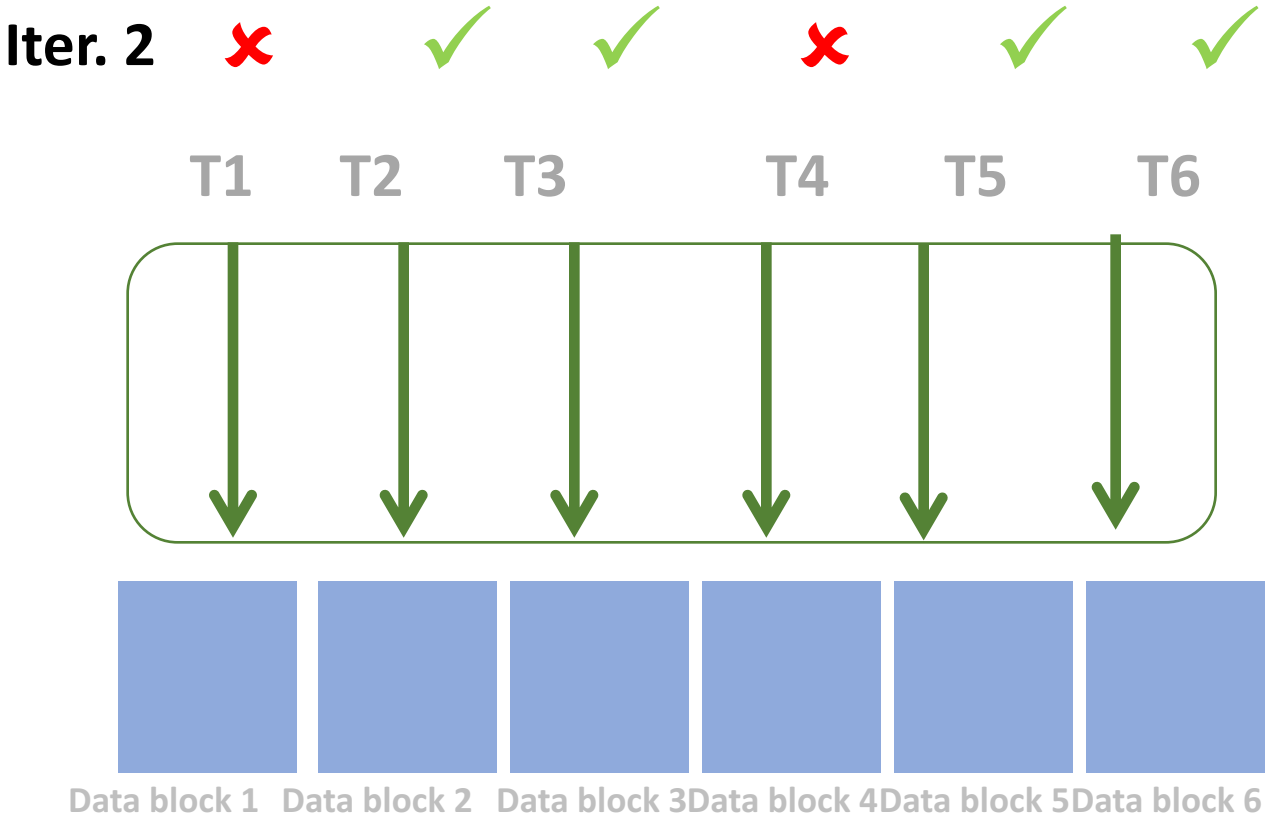
SIMT Architecture: Group execution

6 active threads

```
i=thread id
j=0
…
 while(window[i]==lookahead[j]) {
          j++;
        ….
}
```

Different #iterations for each thread

Thread Utilization



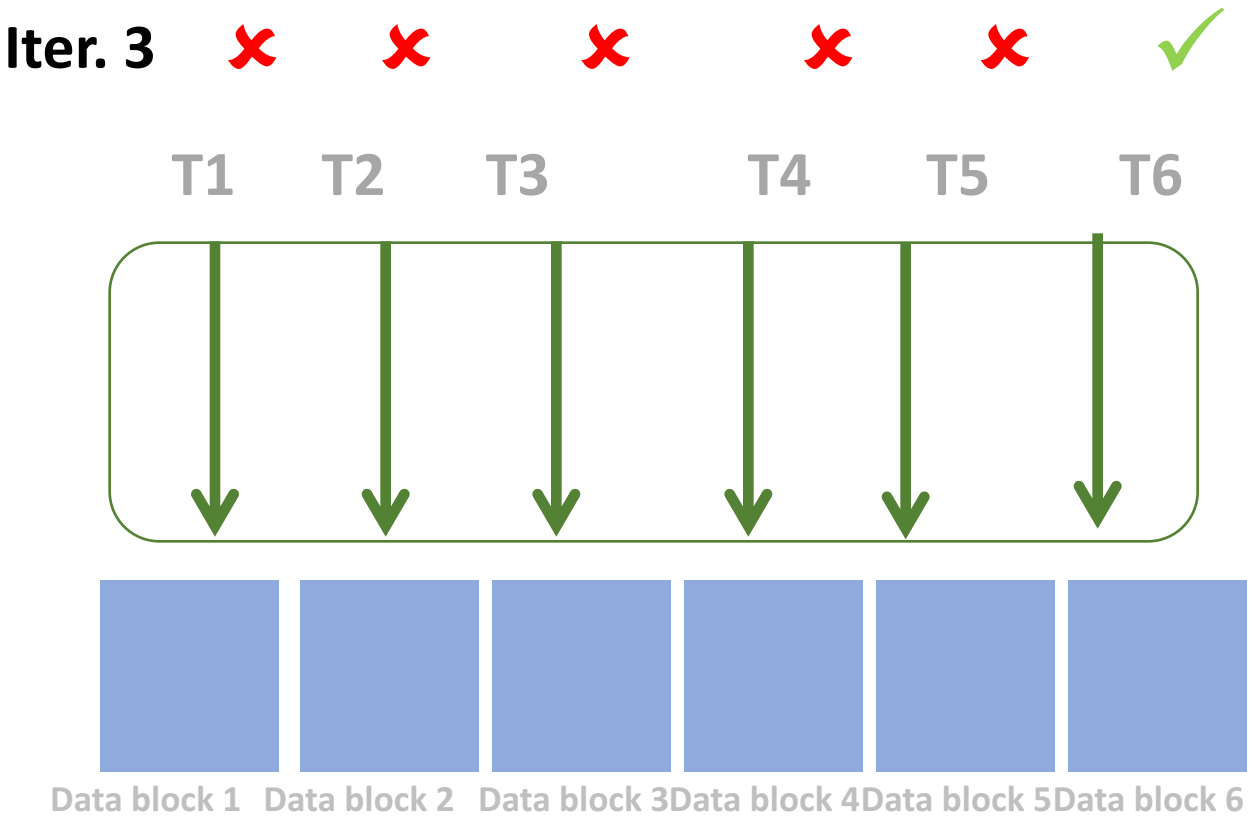
SIMT Architecture: Group execution

4 active threads

```
i=thread id
j=0
…
 while(window[i]==lookahead[j]) {
            j++;
          ….
}
```

Different #iterations for each thread

Thread Utilization



SIMT Architecture: Group execution

1 active thread

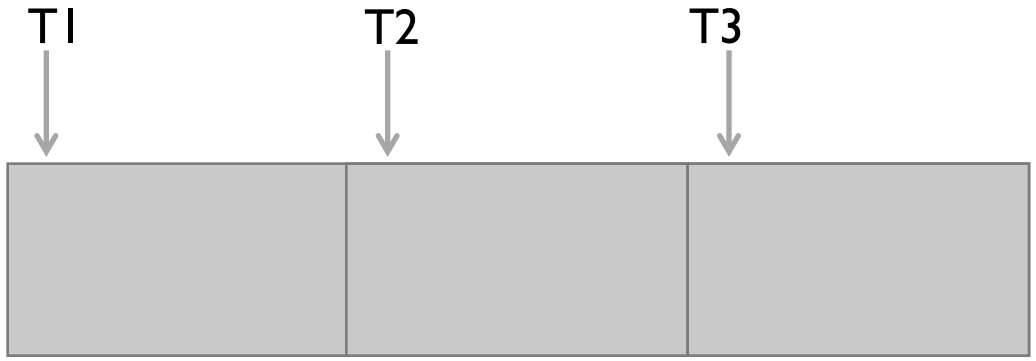
```
i=thread id
j=0
…
 while(window[i]==lookahead[j]) {
            j++;
          ….
}
```

Different #iterations for each thread

(6+4+1)/(3*6) = 11/18 = 61% thread utilization

Memory Access Pattern

Actual memory access pattern

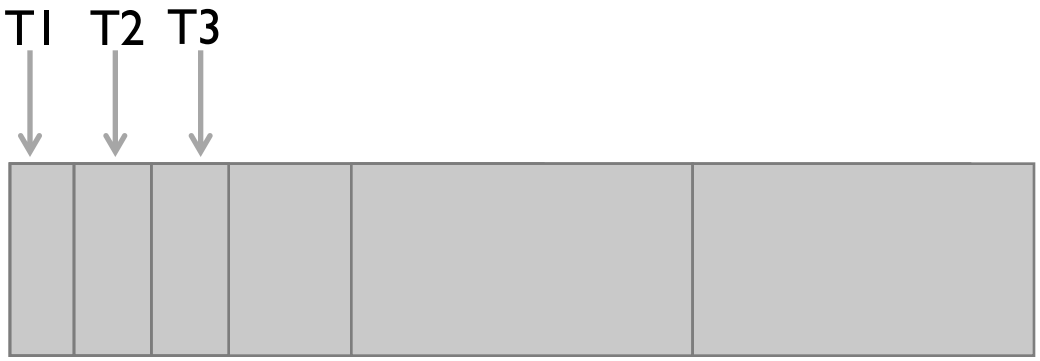


Data block 1 Data Block 2 Data Block 3

Data block size>32K →Many cache lines loaded

- Low memory bandwidth

Optimal GPU memory access pattern



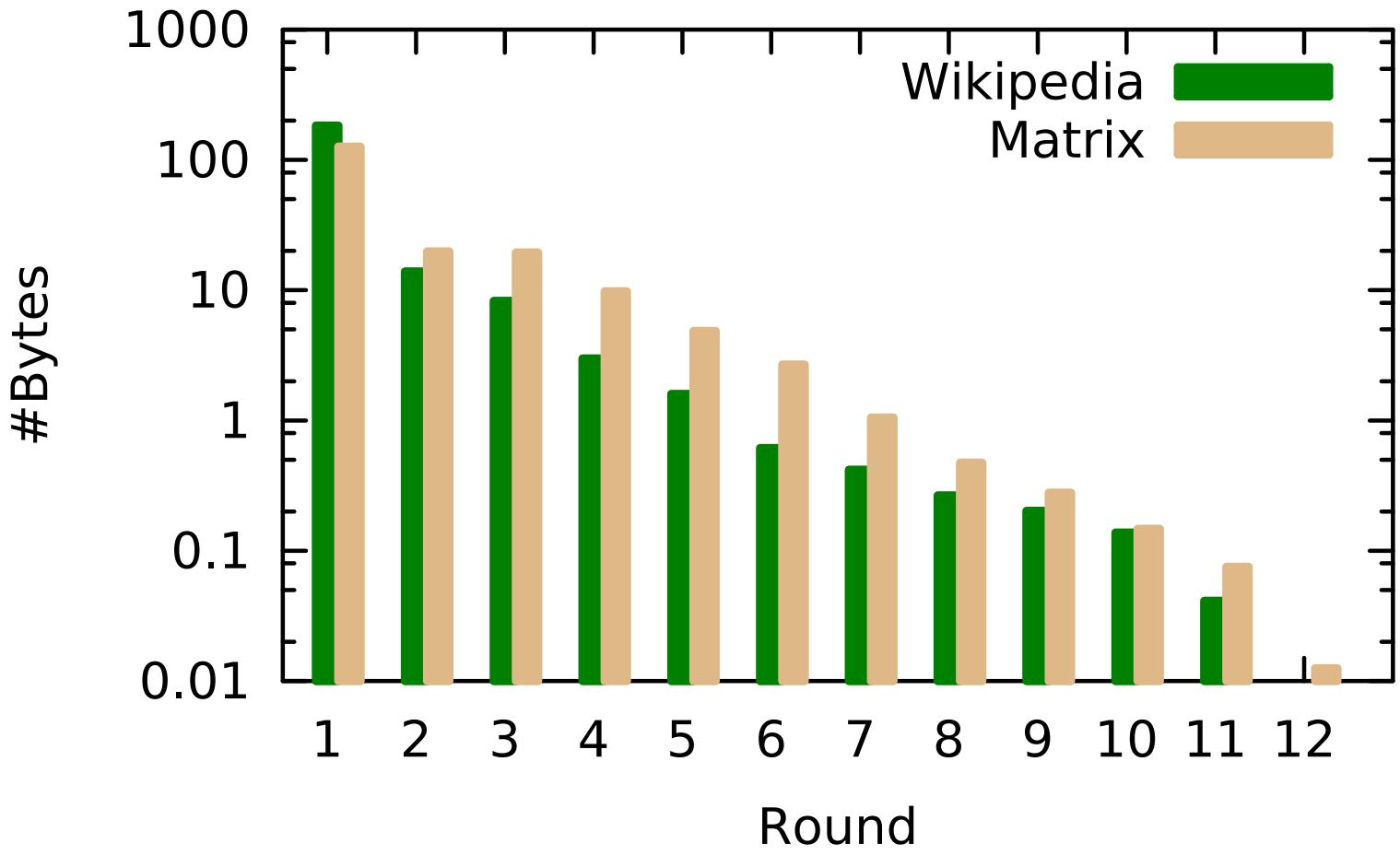
Data Block 1 Data Block 2 Data Block 3

Thread memory accesses in the same cache line

Why Deflate?

- Deflate: Combination of **LZ77** & **Huffman coding**
- LZ77 component of many Big Data compression solutions
  - Snappy
  - LZO
  - gzip
  - LZ4

Bytes Per Round



Level of Nestedness