Parallel lossless compression using GPUs

Eva Sitaridi* Rene Mueller Tim Kaldewey

Columbia University **IBM Almaden** **IBM Almaden**

eva@cs.columbia.edu muellerr@us.ibm.com tkaldew@us.ibm.com

*Work done while interning in IBM Almaden, partially funded from NSF Grant IIS-1218222

Agenda

- Introduction
- Overview of compression algorithms
- GPU implementation
  - LZSS compression
  - Huffman coding
- Experimental results
- Conclusions

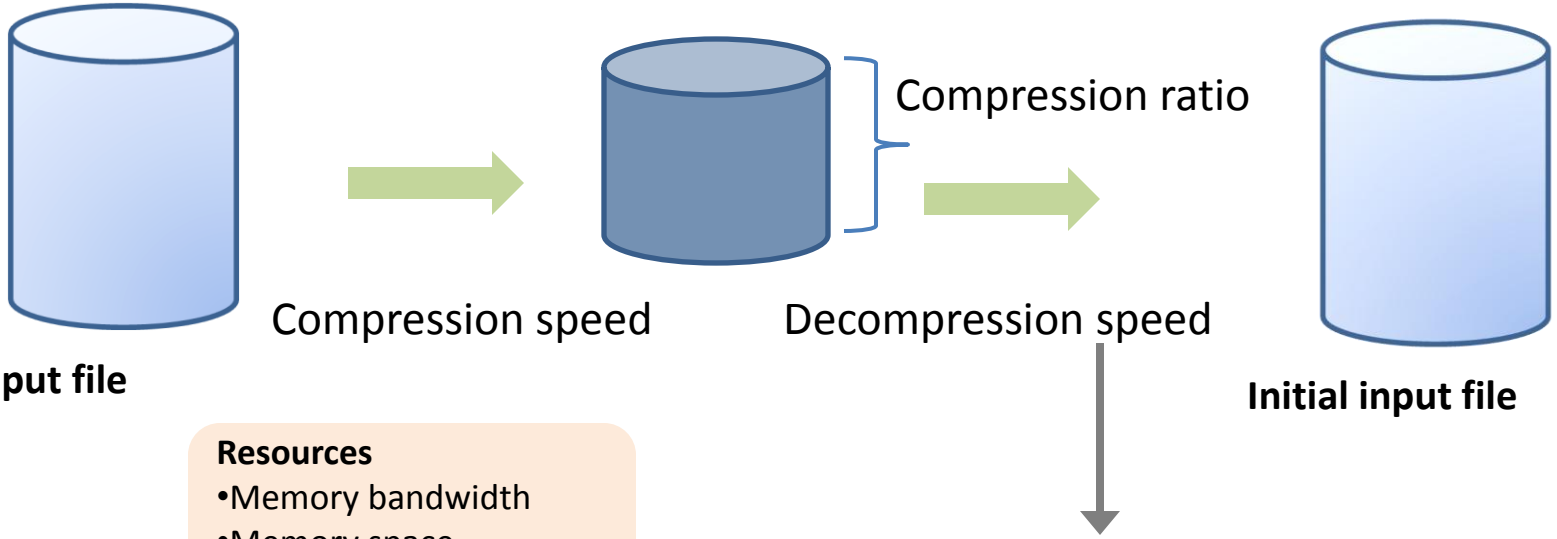
Why compression?

- Data volume doubles every *2 years**
  - Data retained for longer periods
  - Data retained for business analytics
- Make better utilization of available storage resources
  - Increase storage capacity
  - Improve backup performance
  - Reduce bandwidth utilization

- ***Compression** should be seamless*
- ***Decompression** important for Big Data workloads*

*Sybase Adaptive Server Enterprise Data Compression, Business white paper 2012

Compression trade-offs



Resources
- Memory bandwidth
- Memory space
- CPU utilization

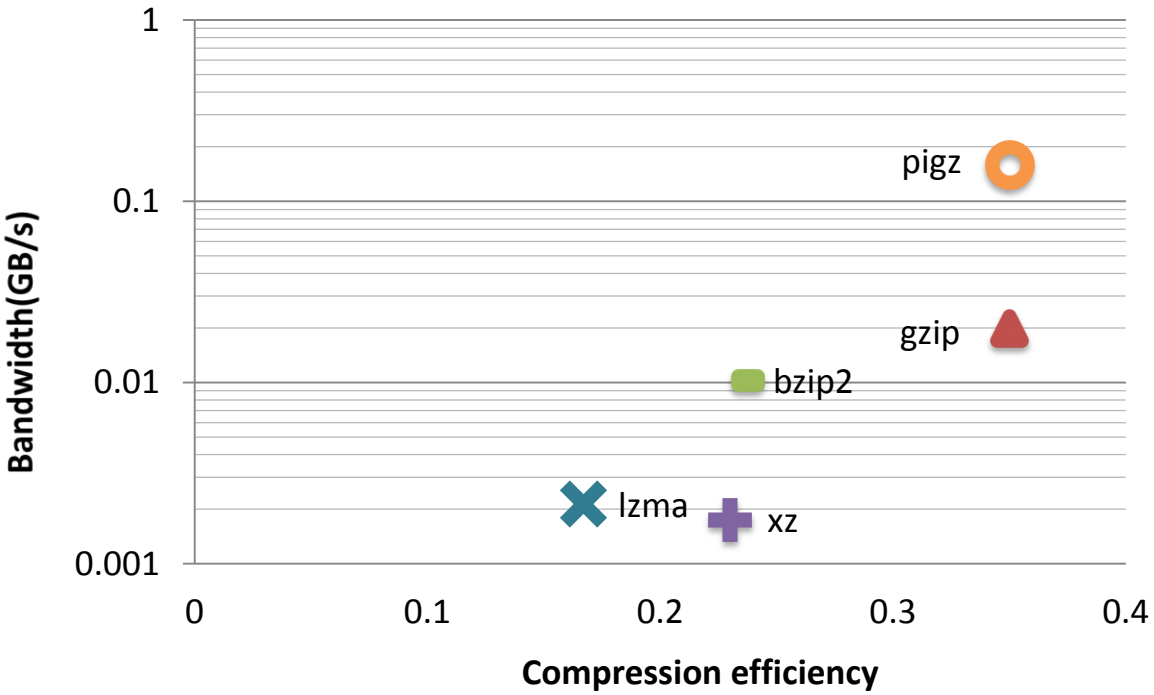
More important in some cases!

Compression speed **vs** Compression efficiency

Decompression speed **vs** Compression efficiency

Compression speed **vs** Decompression speed

Compression resource intensive



Dataset: English Wikipedia pages
1GB XML text dump

*Compression efficiency=**0.5** →*
***Compressed file** is **half** the original*

•Default compression level used - Performance on Intel i7-3930K (6 cores, 3.2 GHz)

Compression libraries

Deflate format

- LZ77 compression
- Huffman coding
- Single threaded

snappy

Parallel gzip

XZ

All use LZ-variants

LZSS compression

Input characters

0 1 2 3 …

ATTACTAGAATGT TACTAATCTGAT

CGGGCCGGGCCTG

Output tokens

*ATTACTAGAATGT***(2,5)…**

Literals

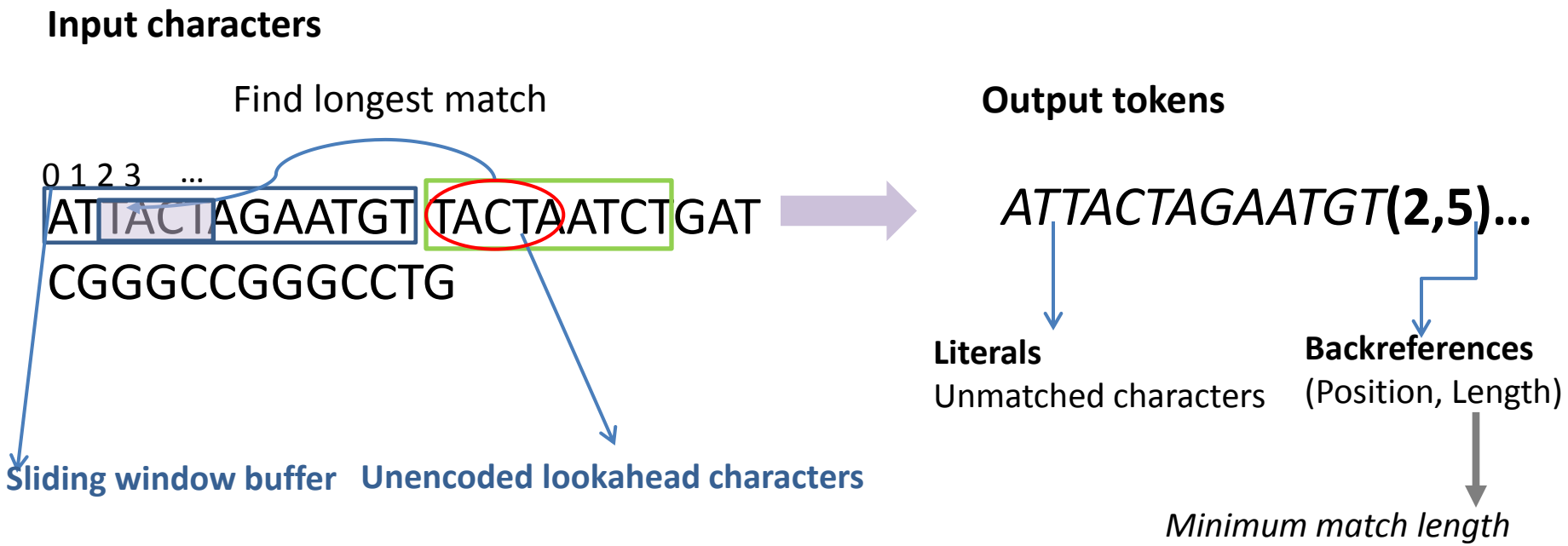
Unmatched characters

Backreferences

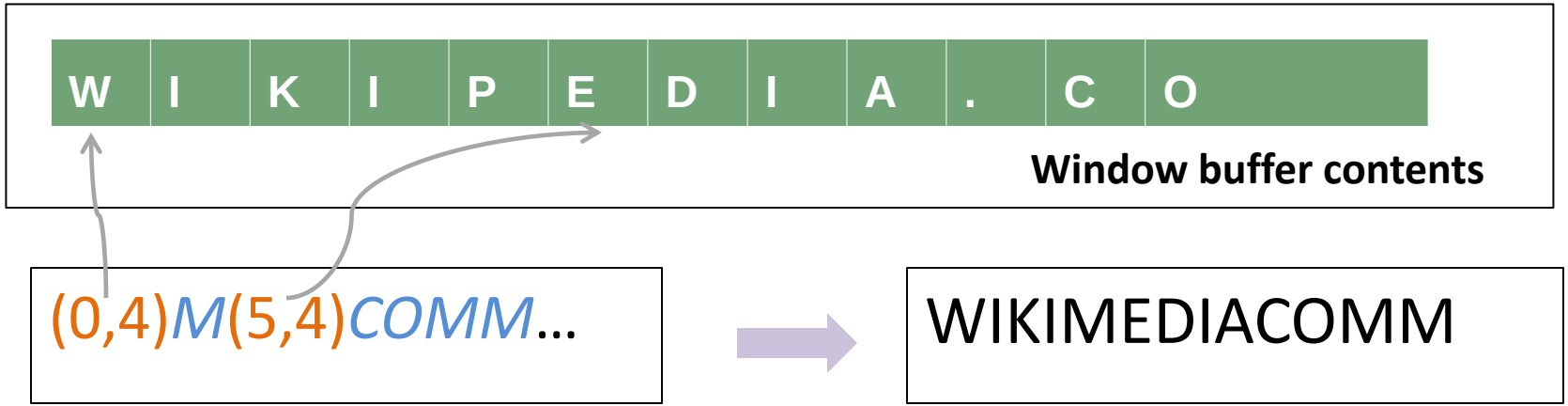
(Position, Length)

Minimum match length

LZSS compression

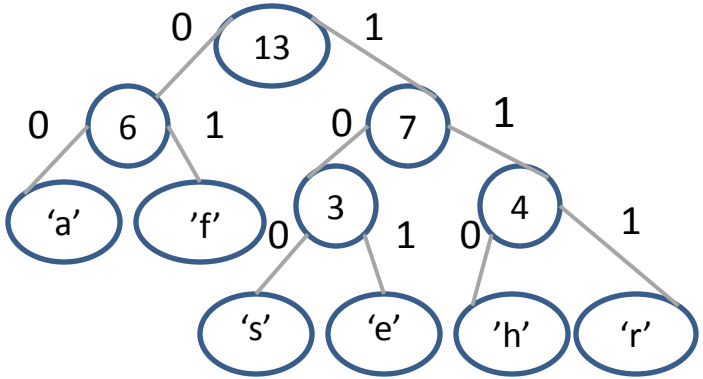


LZSS decompression



Huffman algorithm

- Huffman tree
  - Leaves: encoded symbols
  - Unique prefix for each character



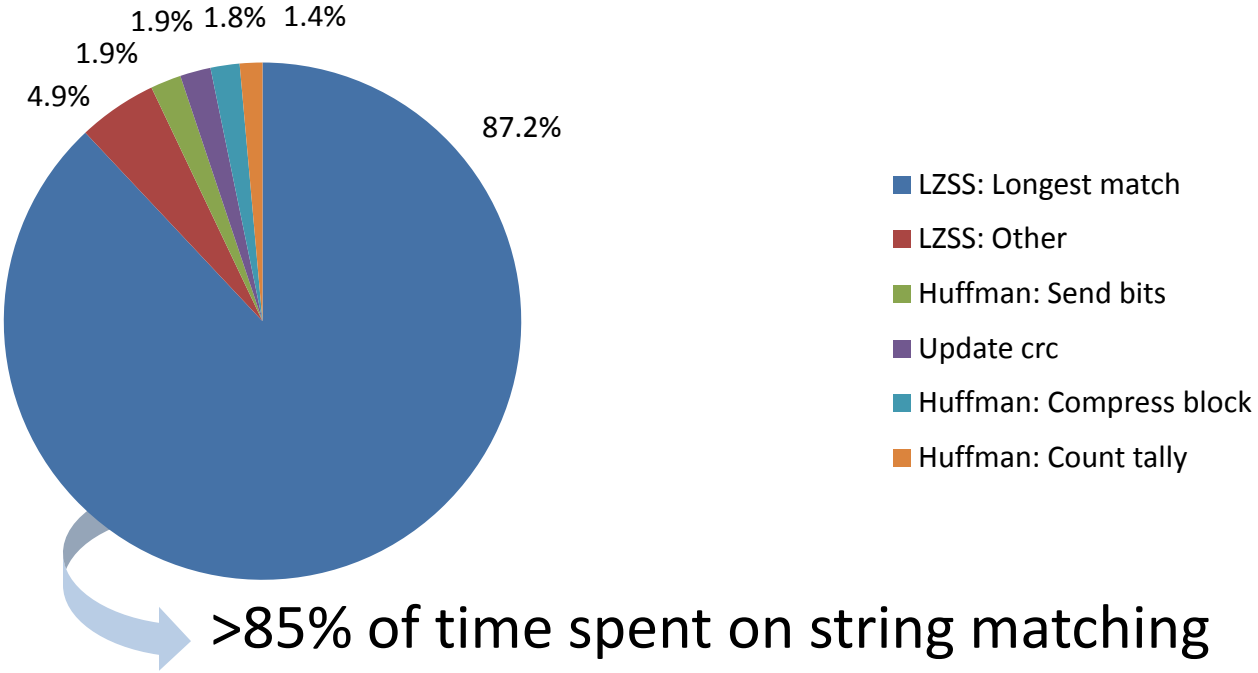
- Huffman coding
  - Short codes for frequent characters

- Huffman decoding
  A) Traverse tree to decode
  B) Use look-up tables for faster decoding

What to accelerate?

Profile of gzip on Intel i7-3930K

Input: Compressible database column



>85% of time spent on string matching

Accelerate LZSS first

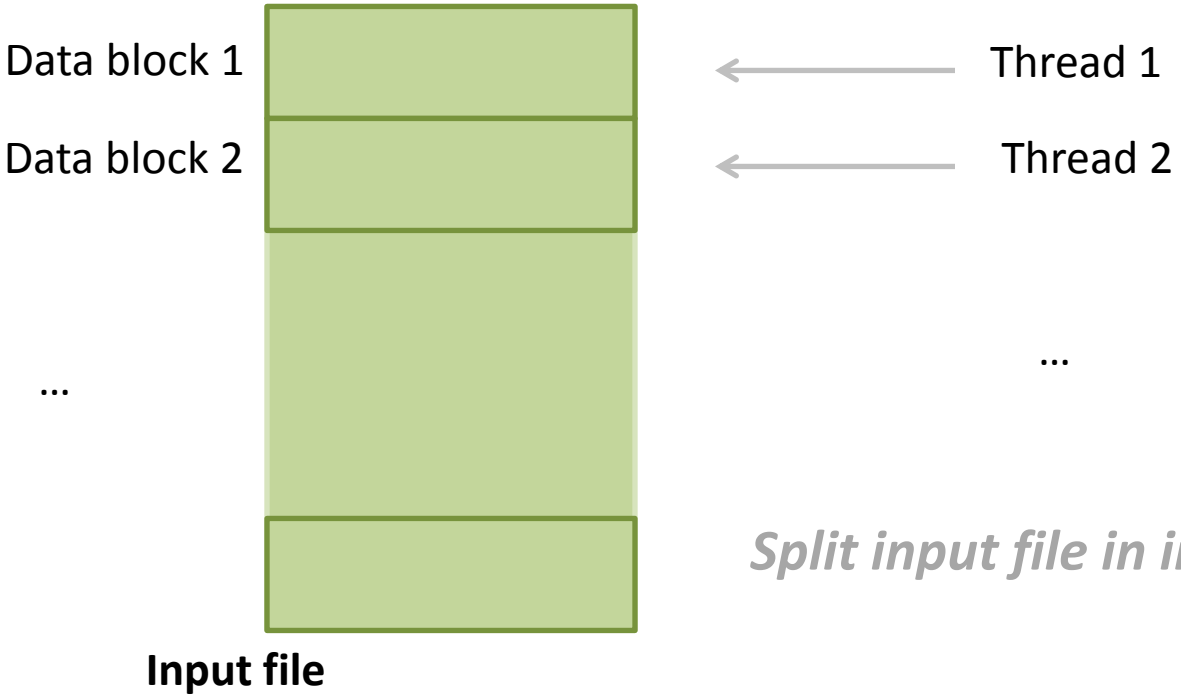
Why GPUs?

- LZSS string matching is memory bandwidth intensive
  - Leverage GPU bandwidth

	Intel i7-3930K	Tesla K20x
Memory Bandwidth (Spec)	51.2 GB/s	250 GB/s
Memory Bandwidth (Measured)	40.4 GB/s	197GB/s
#Cores	6	2688

How to parallelize compression/decompression?

>1000 cores available!

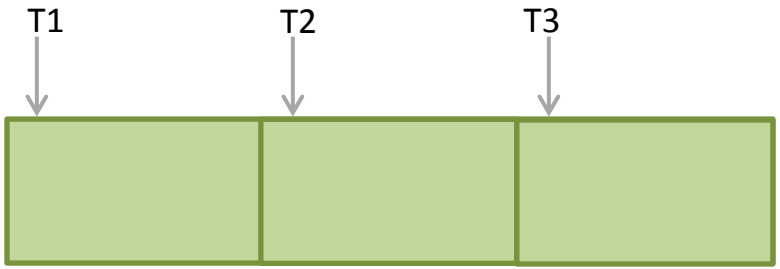


Split input file in independent blocks

Naïve approach: *Threads process independent data/file blocks*

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

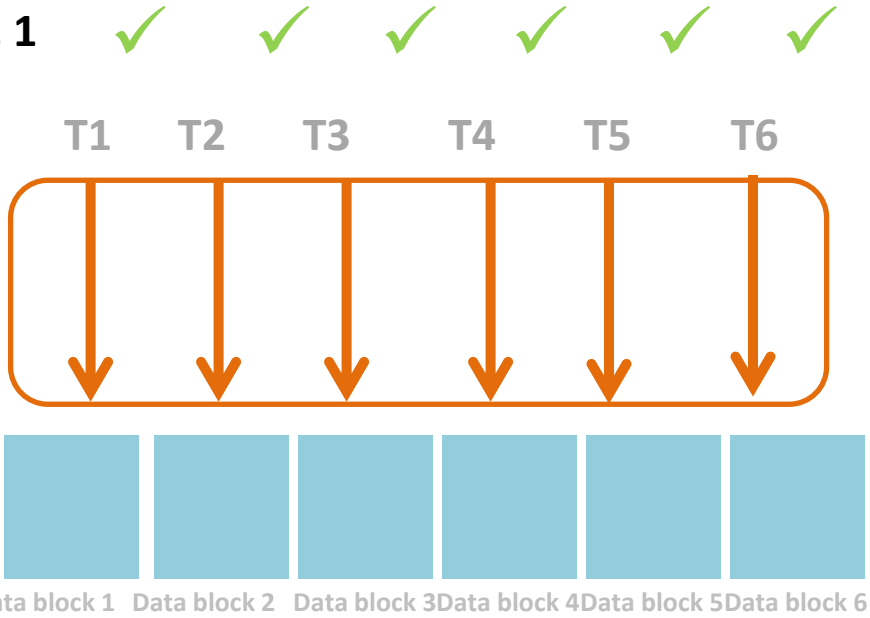
Optimal GPU memory access pattern

Data Block 1 Data Block 2 Data Block 3

Thread memory accesses in the same cache line

Thread utilization

Iter. 1



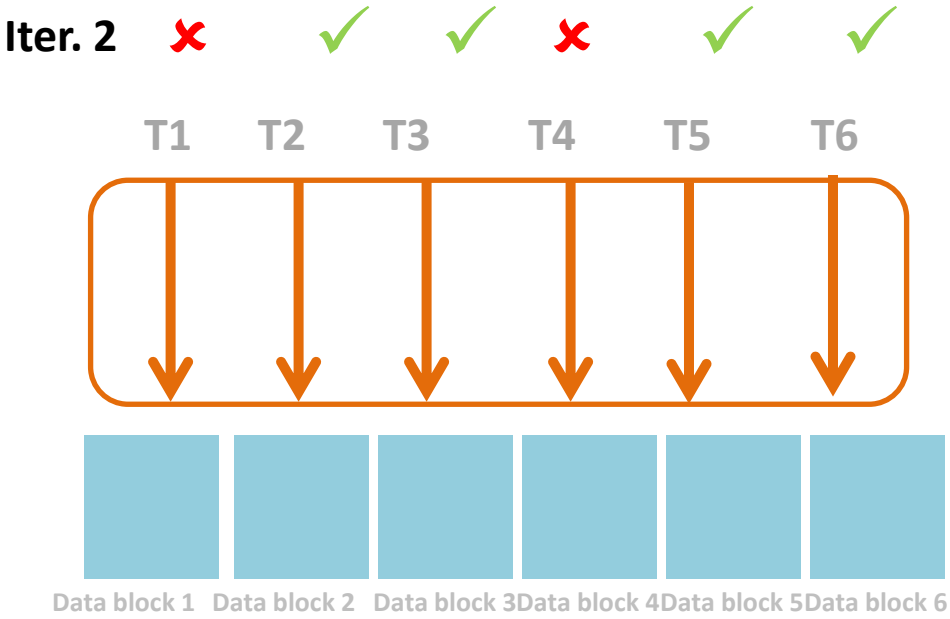
SIMT Architecture: Group execution

6 active threads

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

Different #iterations for each thread

Thread utilization



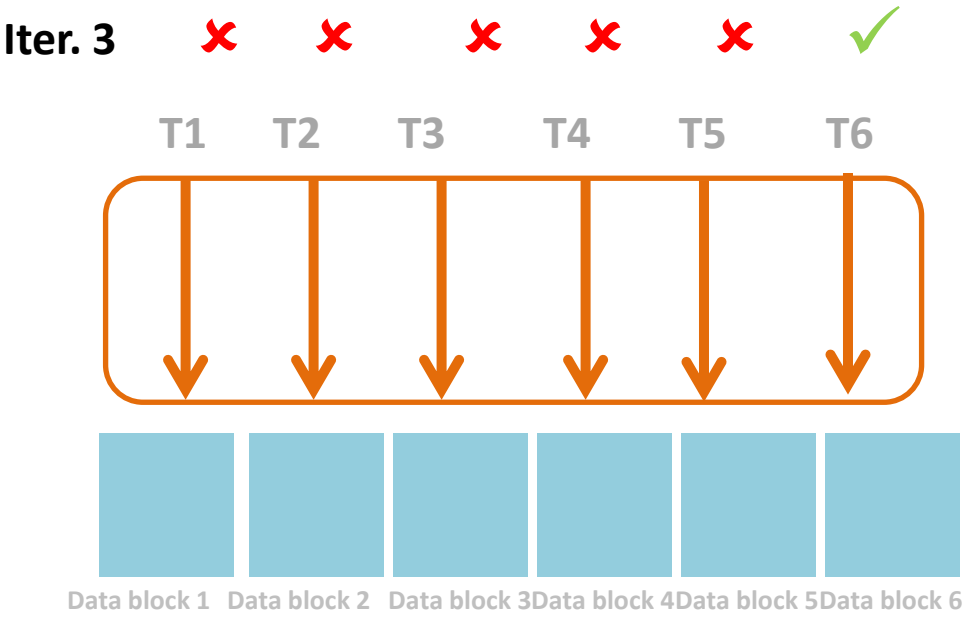
SIMT Architecture: Group execution

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

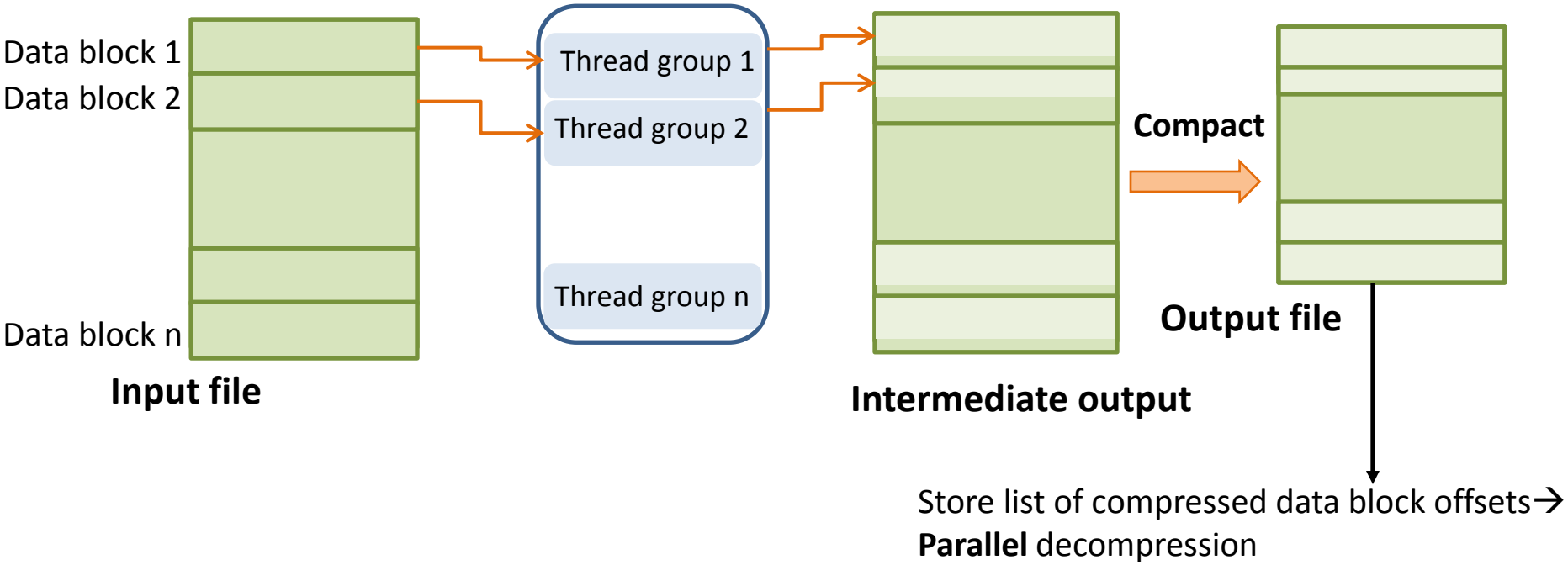
1 active thread

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

Different #iterations for each thread

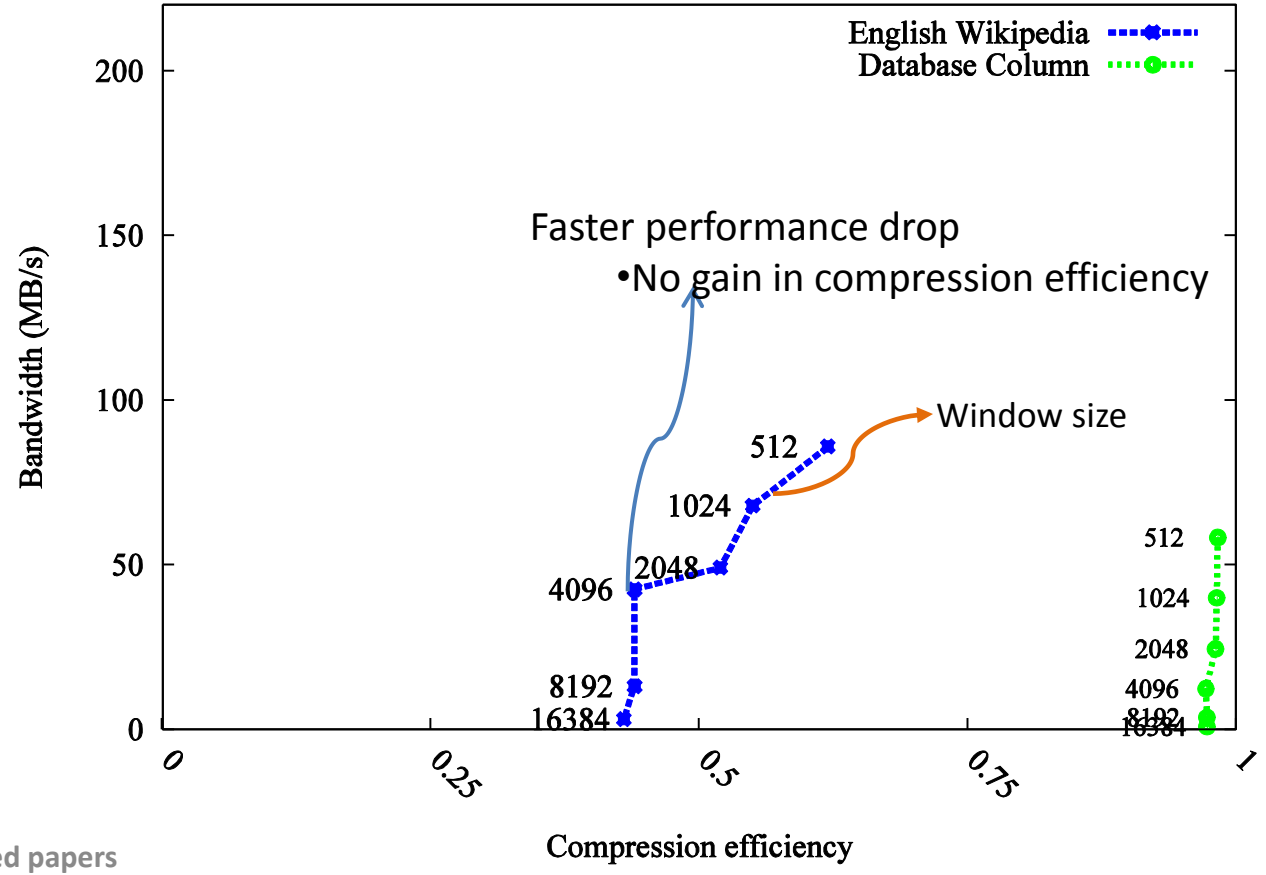
(6+4+1)/(3*6) = 11/18 = 61% thread utilization

GPU LZSS General compression



Better approach: *Each data block is processed by a thread group*

Compression efficiency vs Compression performance



GPU LZSS*
Lookahead: 66 chars
Block size: 64K chars

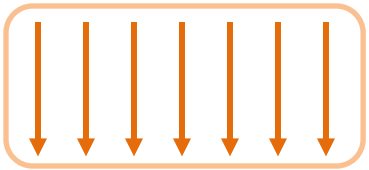
* Related papers

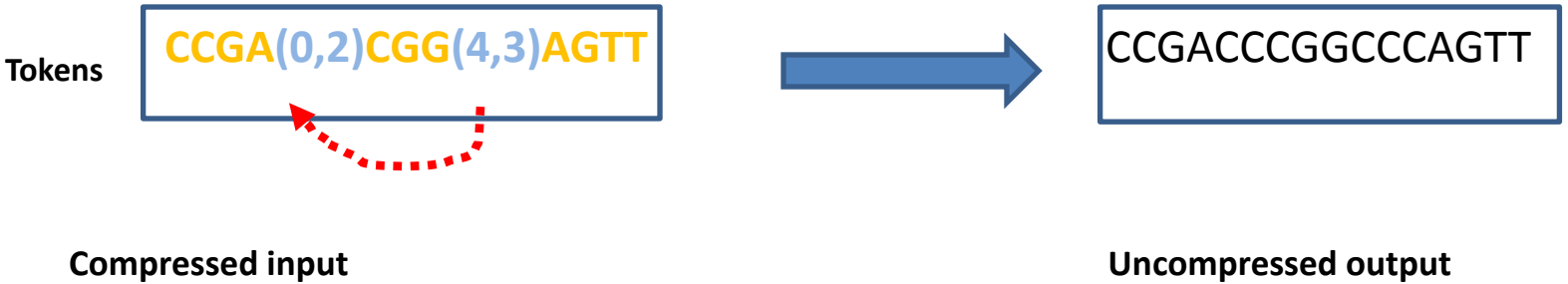
A. Ozsoy and M. Swany, "CULZSS: LZSS Lossless Data Compression on CUDA"

A. Balevic, "Parallel Variable-Length Encoding on GPGPUs"

GPU LZSS decompression

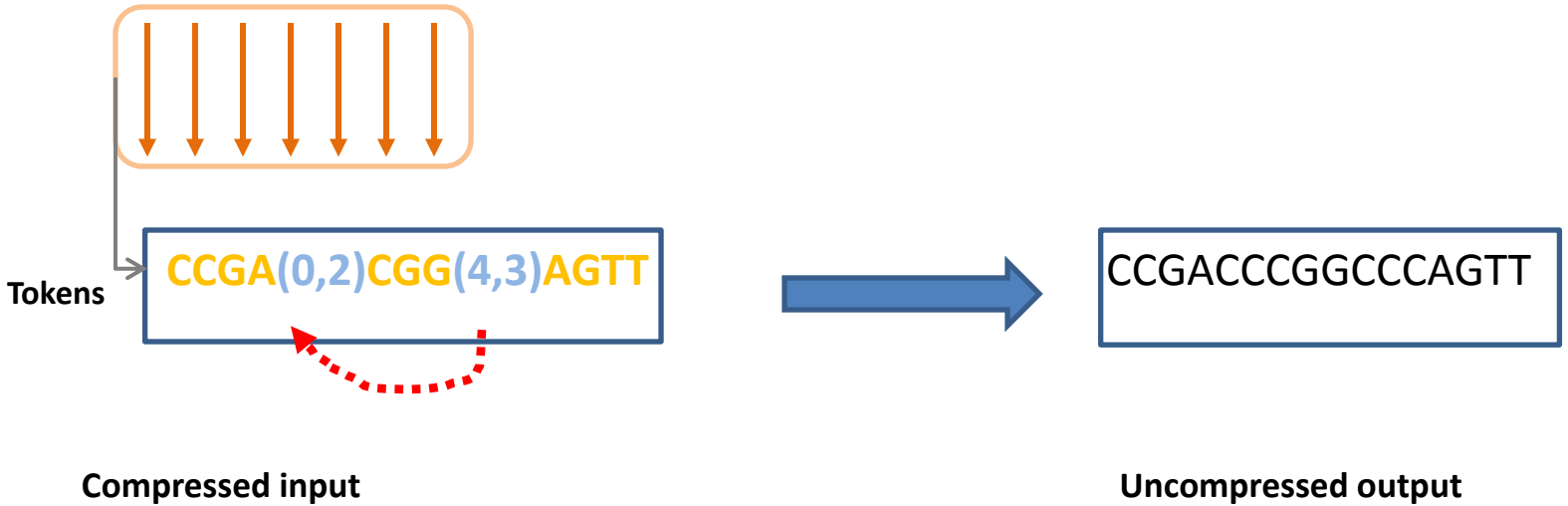
1) Compute total size of tokens (serialized)



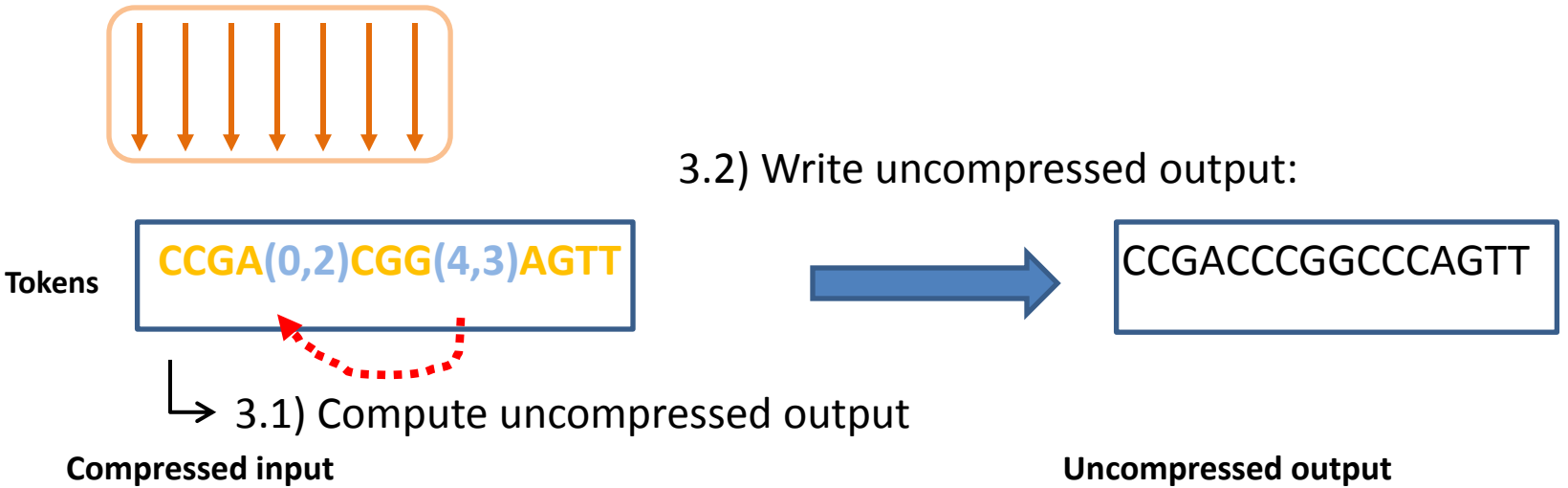


GPU LZSS decompression

2) Read tokens (parallel)



GPU LZSS decompression



Problem: Backreferences processed in parallel might be dependent! →
Use voting function __*ballot* to detect conflicts

Writing LZSS tokens to output

Case A: All literals

Tokens: CCGAGATTGAGTT

1) Write literals (parallel)

Case B: Literals & non-conflicting backreferences

Tokens: CCGA(0,2)CGG(0,3)AGTT

1) Write literals (parallel)
2) Write backreferences (parallel)

Case C: Literals & conflicting backreferences

Tokens: CCGA(0,2)CGG(4,3)AGTT

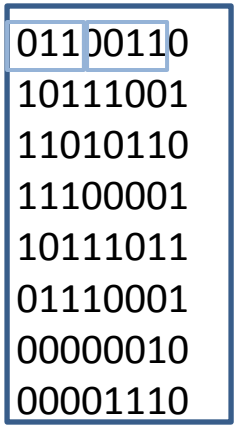
1) Write literals (parallel)
2) Write non-conflicting backreferences (parallel)
3) Write remaining backreferences (serial)

Huffman entropy coding

- Inherently sequential
- Coding challenge
  - Compute destination of encoded data
- Decoding challenge
  - Determine **codeword boundaries**

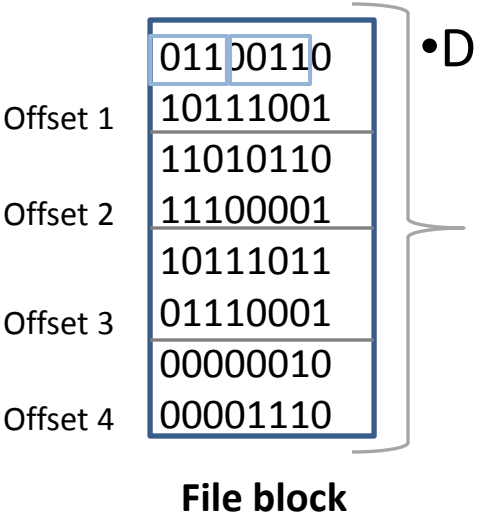
Focus on decoding for end-to-end decompression

Parallel Huffman decoding



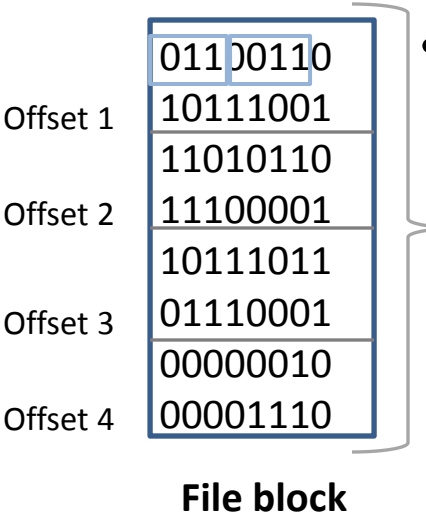
File block

Parallel Huffman decoding



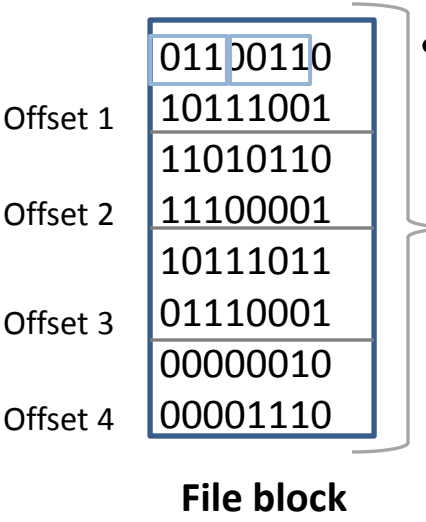
- During **coding**
  - Split data blocks in sub-blocks
  - Store sub-block offsets → Parallel sub-block decoding

Parallel Huffman decoding



- During **coding**
  - Split data blocks in sub-blocks
  - Store sub-block offsets → Parallel sub-block decoding
- During **decoding**
  - Use look-up tables for decoding rather than Huffman trees
  - Fit look-up table in shared memory
    - Reduce number of codes for length and distance

Parallel Huffman decoding



- During **coding**
  - Split data blocks in sub-blocks
  - Store sub-block offsets → Parallel sub-block decoding
- During **decoding**
  - Use look-up tables for decoding rather than Huffman trees
  - Fit look-up table in shared memory
    - Reduce number of codes for length and distance

Trade compression efficiency for decompression speed

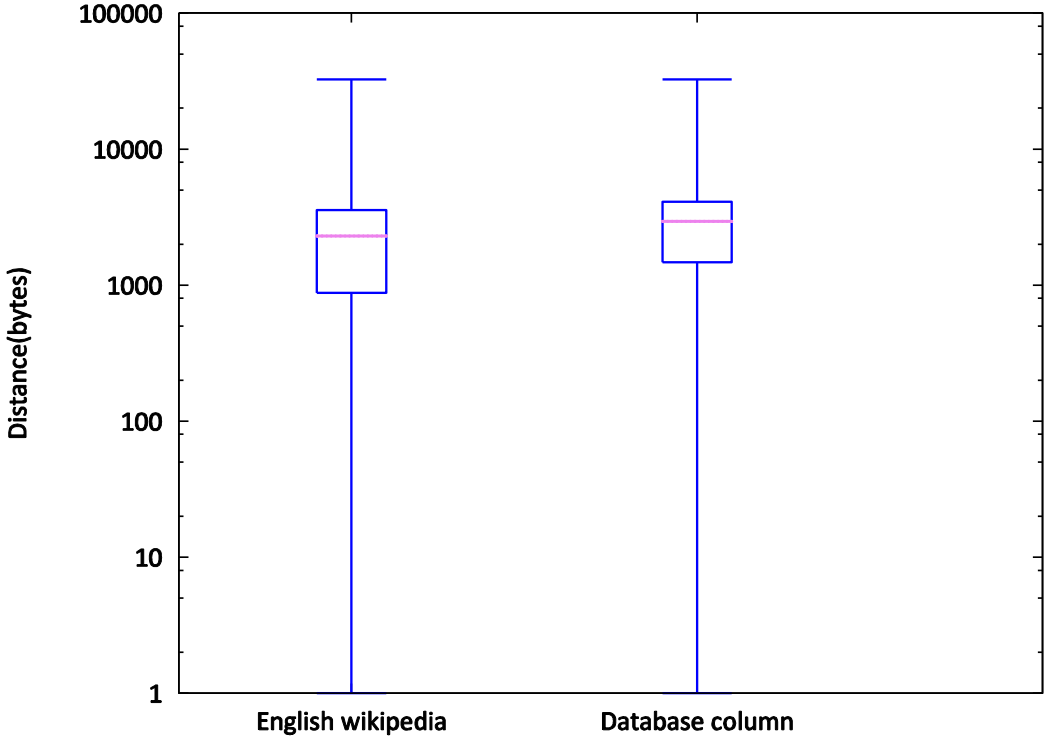
Experimental system

Linux, kernel 3.0.74

	Intel i7-3930K	Tesla K20x
Memory bandwidth (Spec)	51.2 GB/s	250 GB/s
Memory bandwidth (Measured)	40.4 GB/s	197 GB/s
Memory capacity	64 GB	6 GB
#Cores	6 (12 threads)	2688
Clock frequency	3.2 GHz	0.732 GHz

Datasets

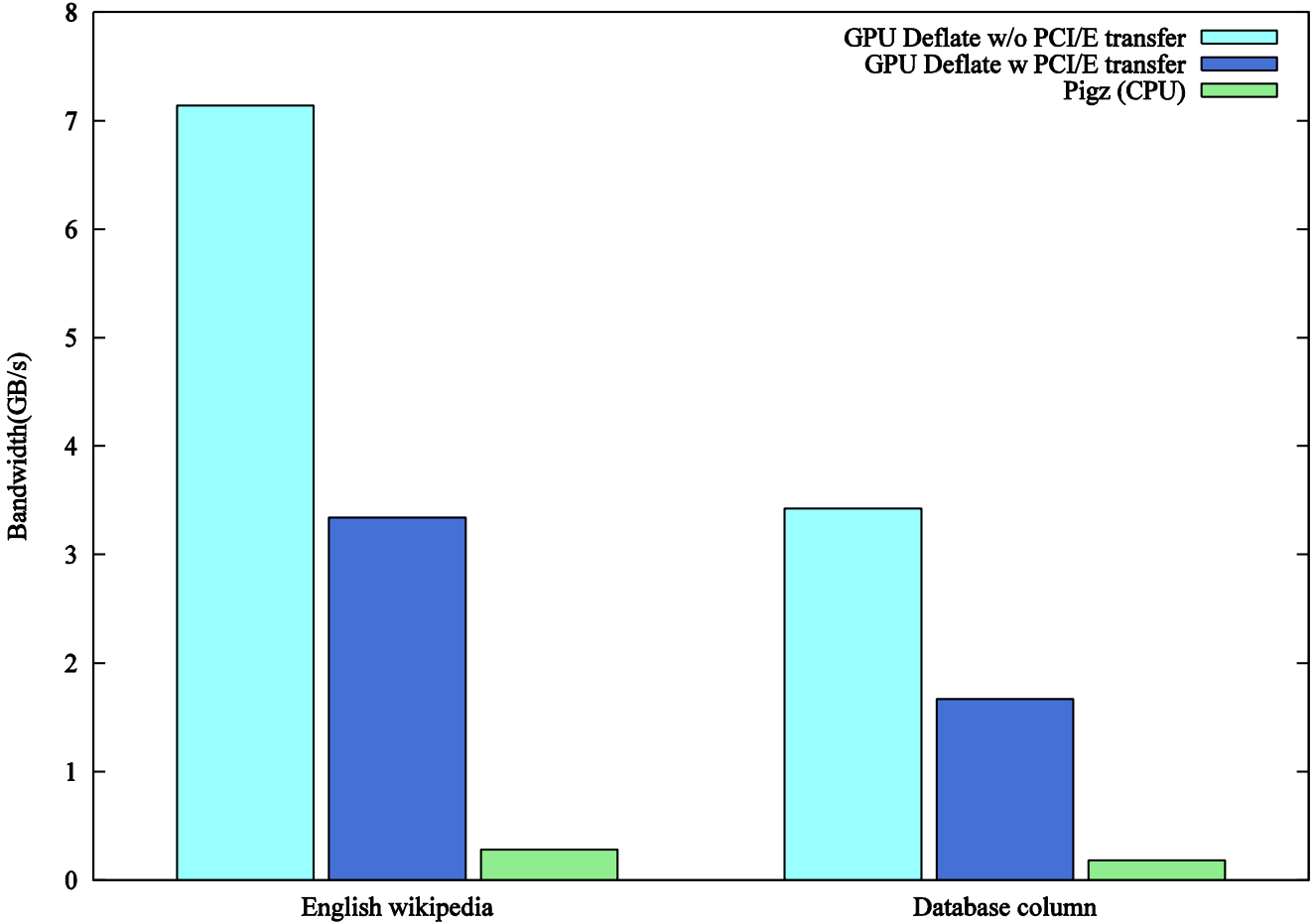
Dataset	Size	Comp. efficiency*
English wikipedia	1GB	0.35
Database column	245MB	0.98



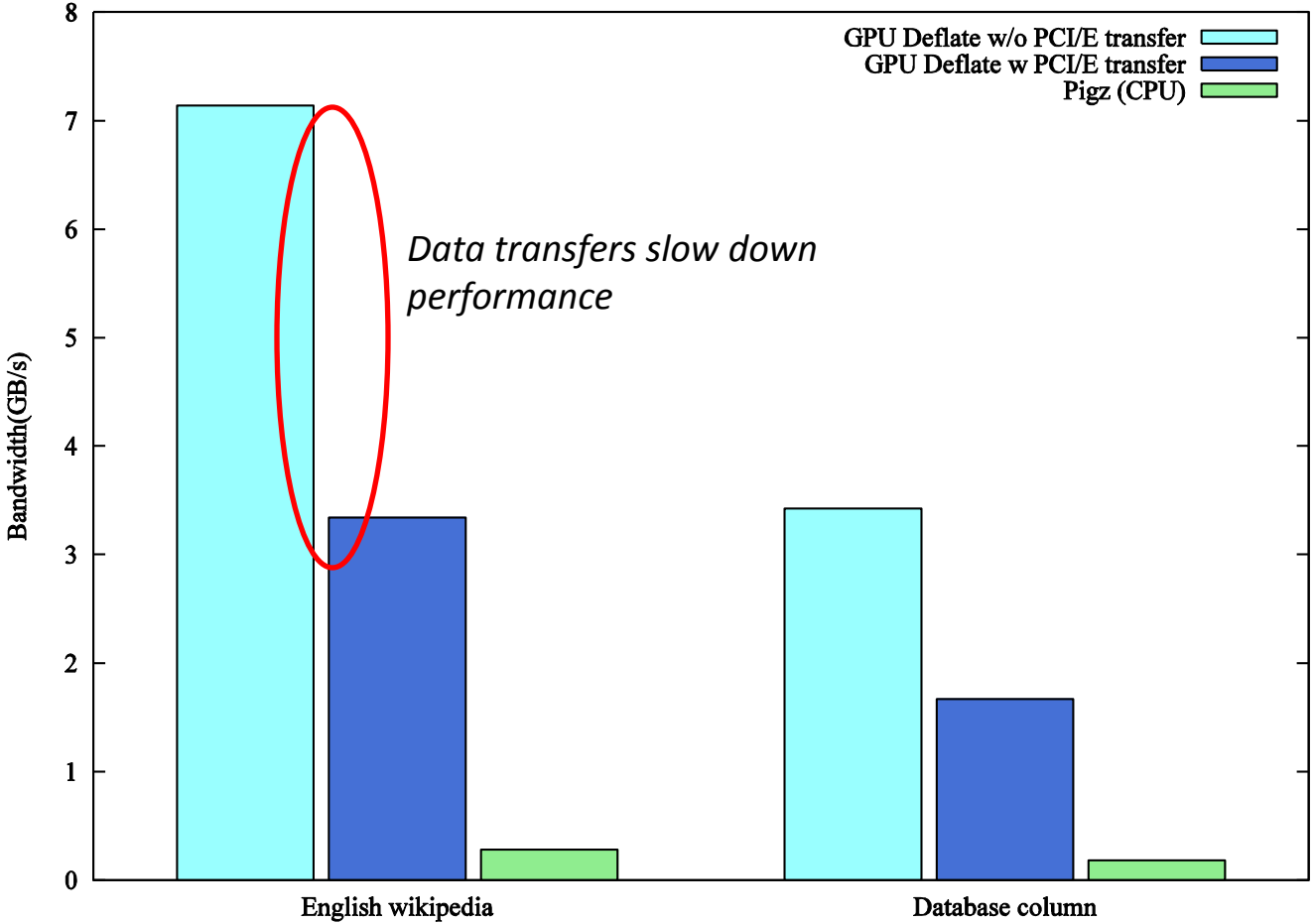
- Datasets already loaded in memory
  - No disk I/O

*For default parameter of gzip

Decompression performance

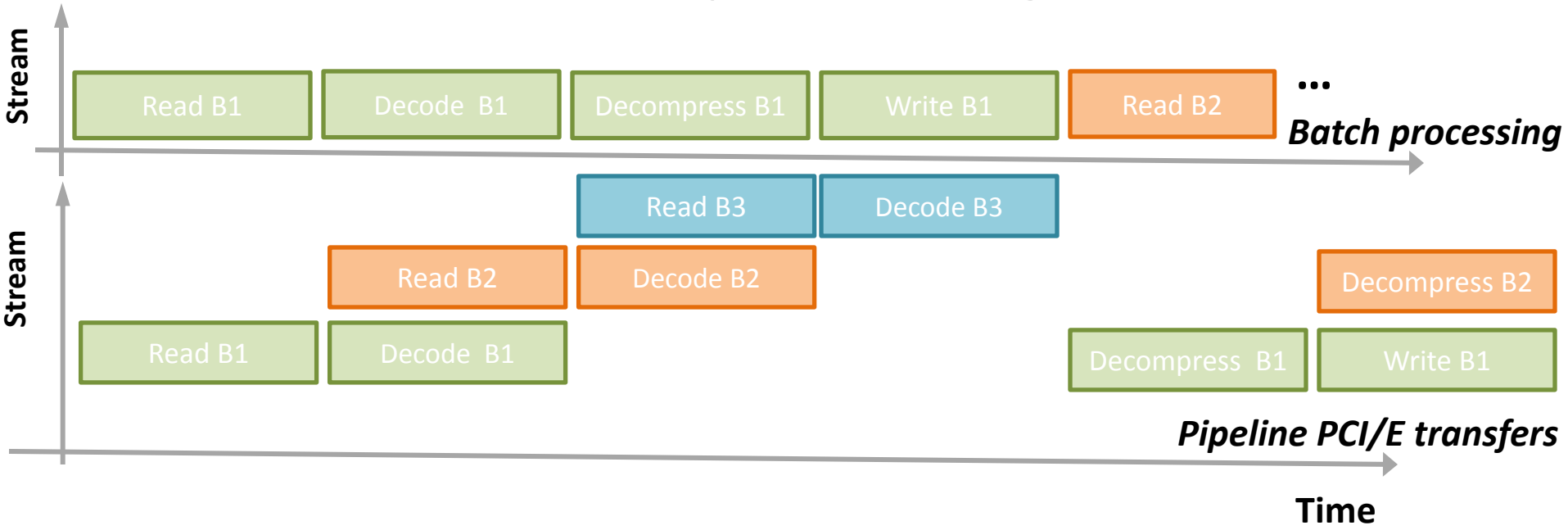


Decompression performance

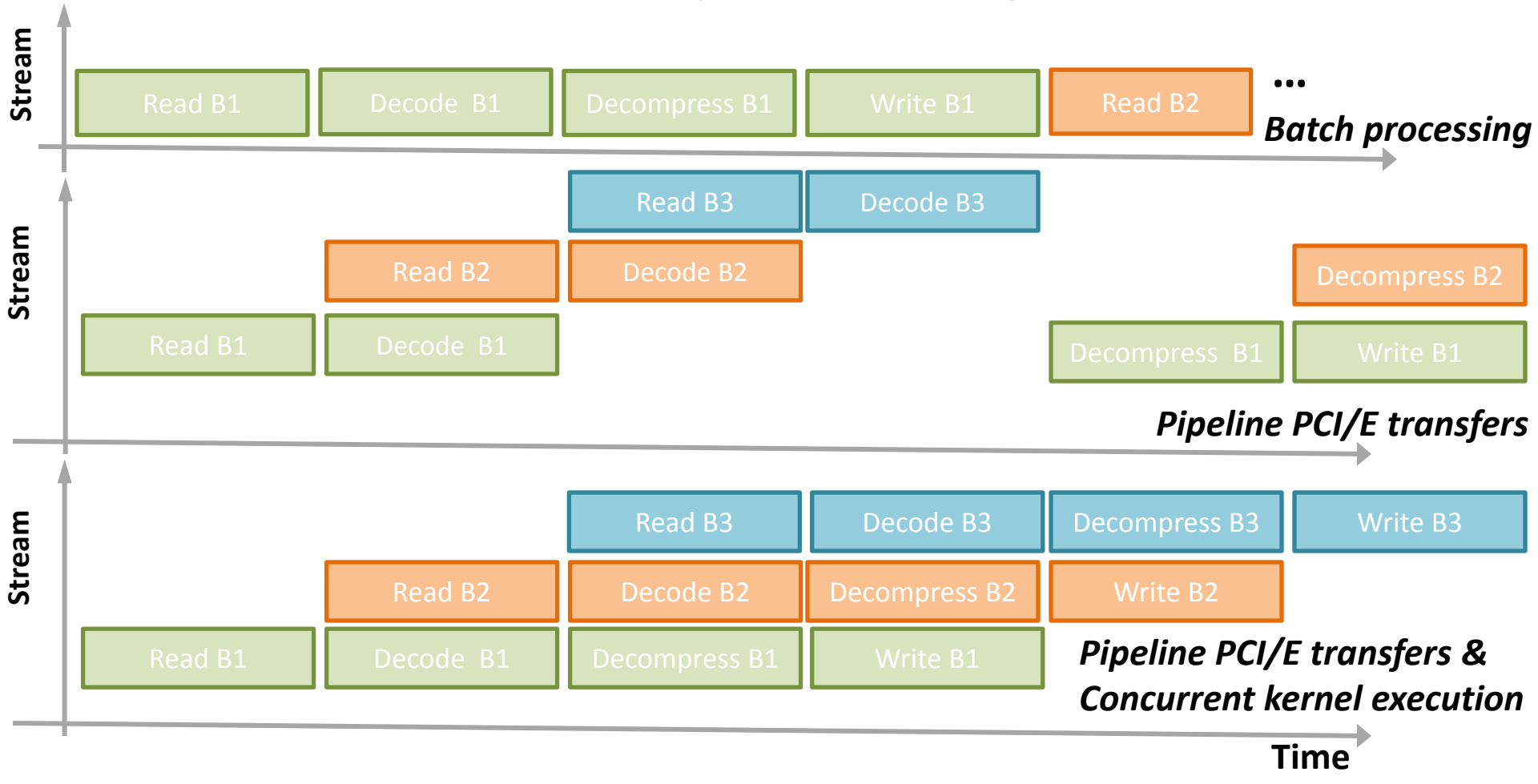


Hide GPU to CPU transfer I/O using CUDA Streams

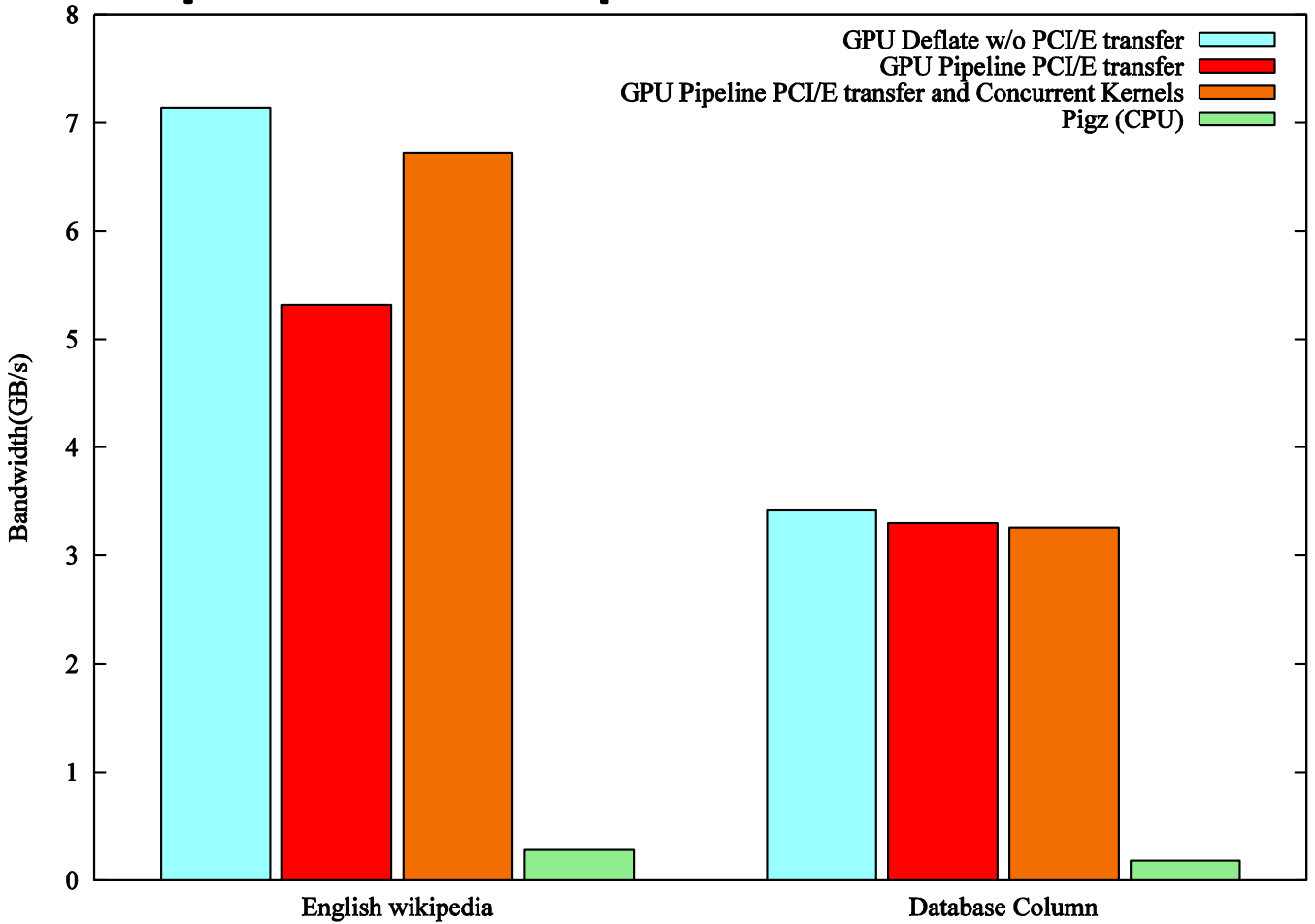
Hide GPU to CPU transfer I/O using CUDA Streams



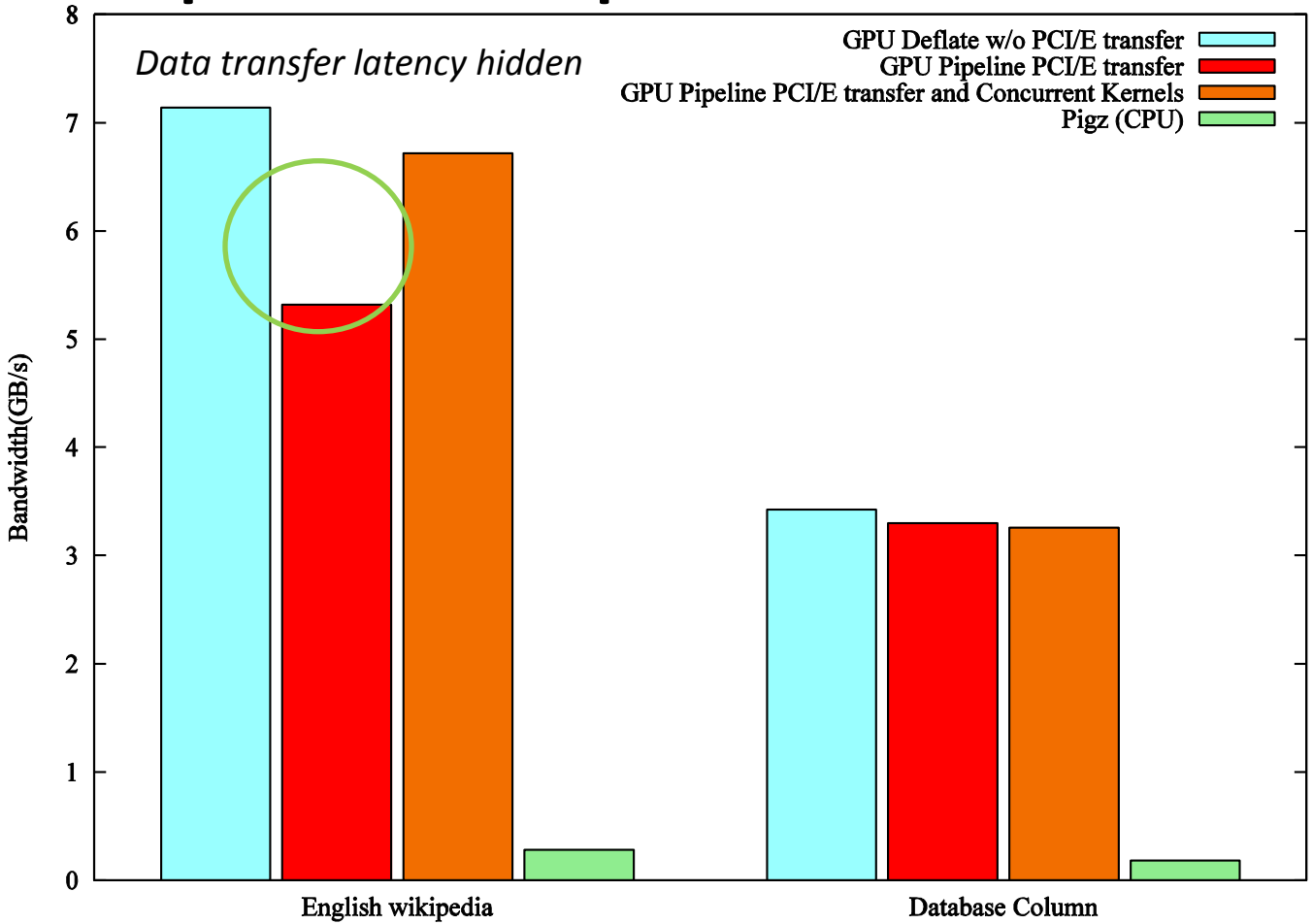
Hide GPU to CPU transfer I/O using CUDA Streams



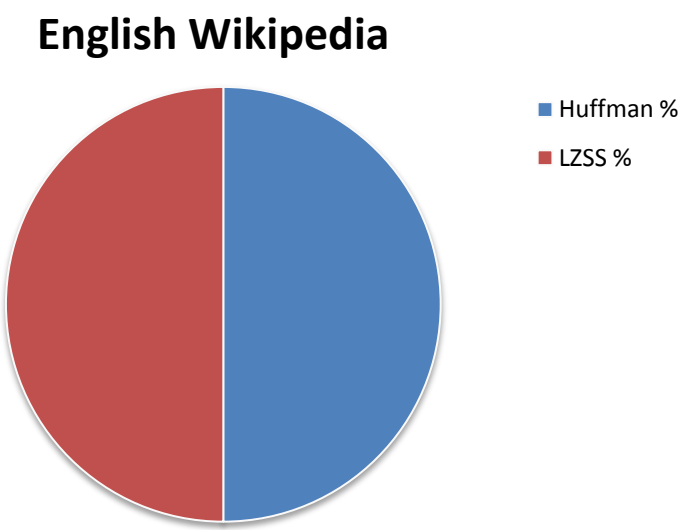
Decompression performance

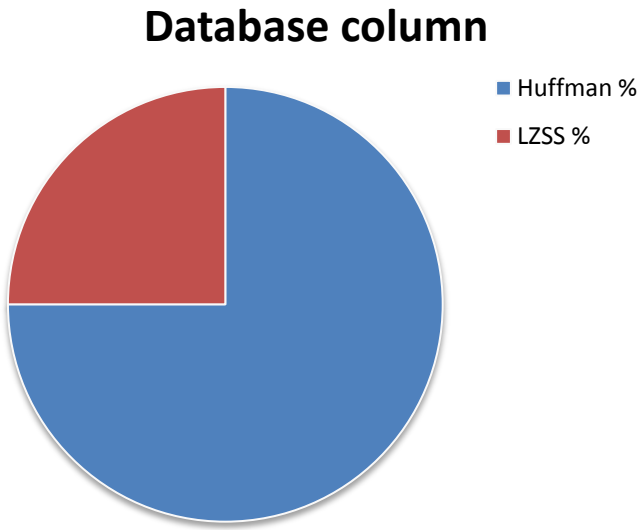


Decompression performance

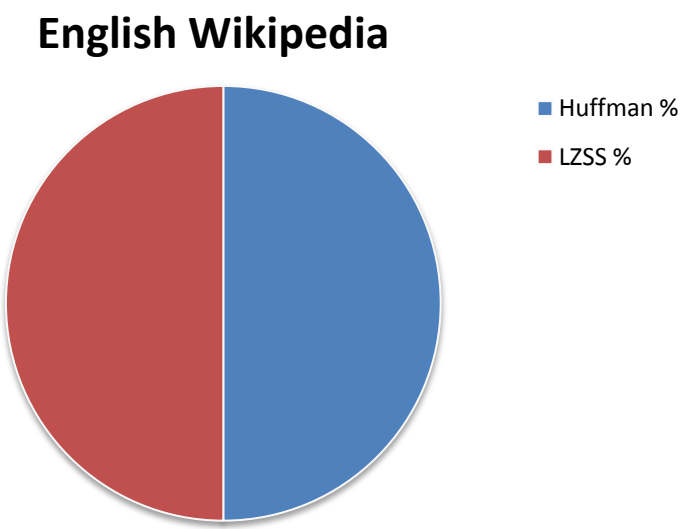


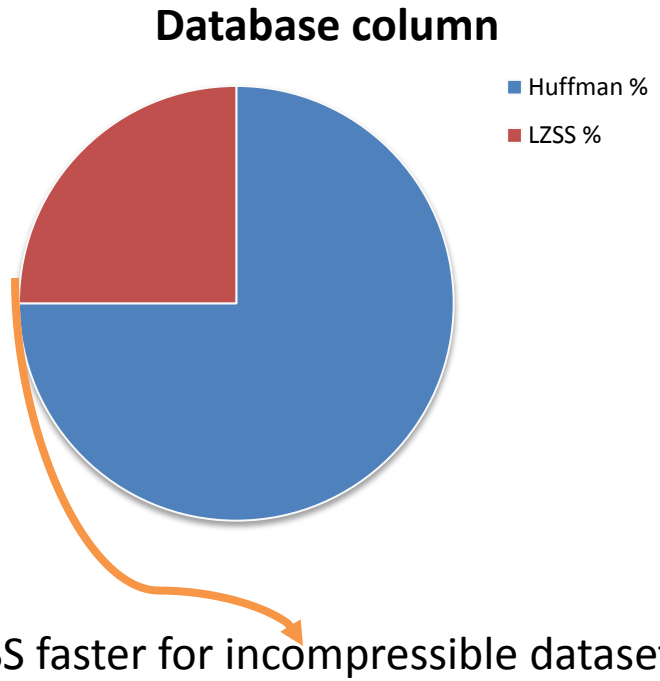
Decompression time breakdown





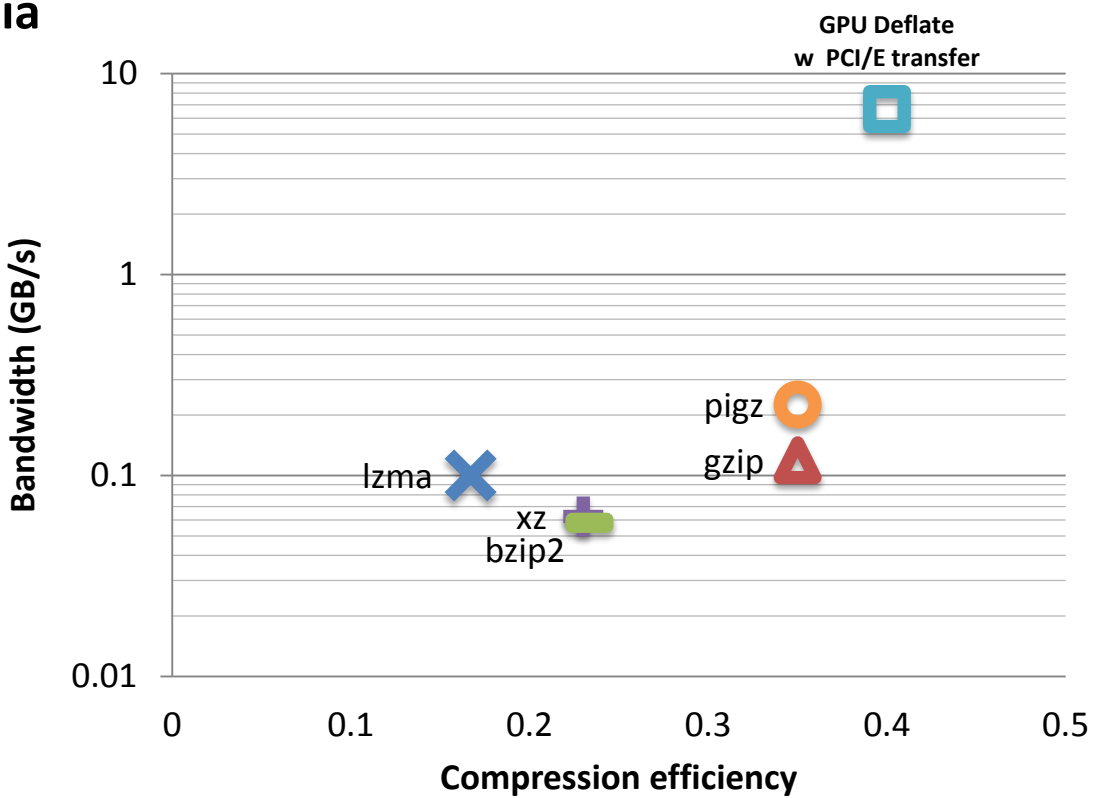
Decompression time breakdown





Decompression performance vs Compression efficiency

English Wikipedia



Conclusions

- Decompression
  - Hide GPU-CPU latency using 4-stage pipelining
  - LZSS faster for incompressible files
- Compression
  - Reduce search time (using hash tables ?)

Conclusions

Questions?

- Decompression
  - Hide GPU-CPU latency using 4-stage pipelining
  - LZSS faster for incompressible files
- Compression
  - Reduce search time (using hash tables ?)