Parallel Huffman Coding

Final Presentation

Chen Luo

Patrick (Zhanxiang) Huang

Background

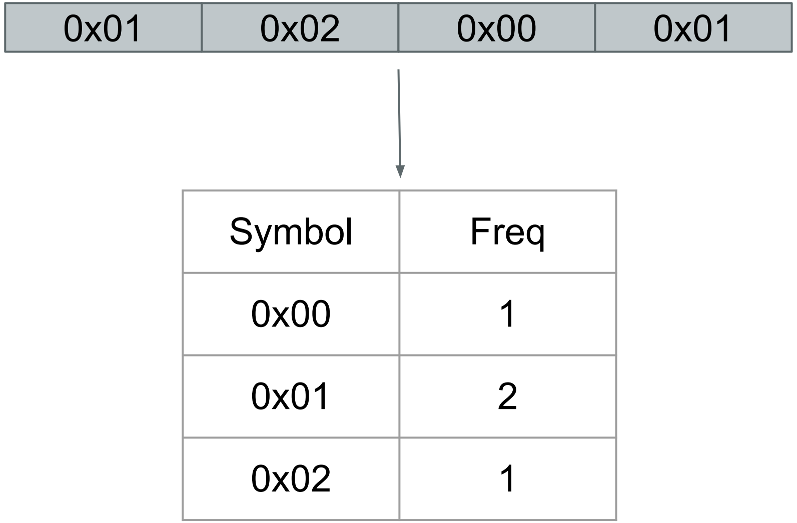
- What is Huffman Coding?
    - Compression algorithm to generate optimal prefix code.

- Why parallelize Huffman Coding is important?
    - Sequential Huffman Coding is used by many compression libraries
    - Huffman Coding takes a significant percentage of compression time

Sequential Compression

- Step 1: Build Symbol Frequencies Histogram

Input File:



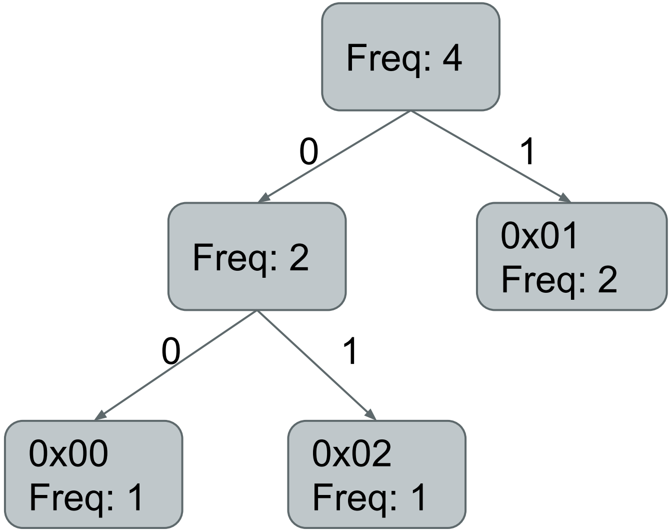
0x01	0x02	0x00	0x01

Symbol	Freq
0x00	1
0x01	2
0x02	1

Sequential Compression

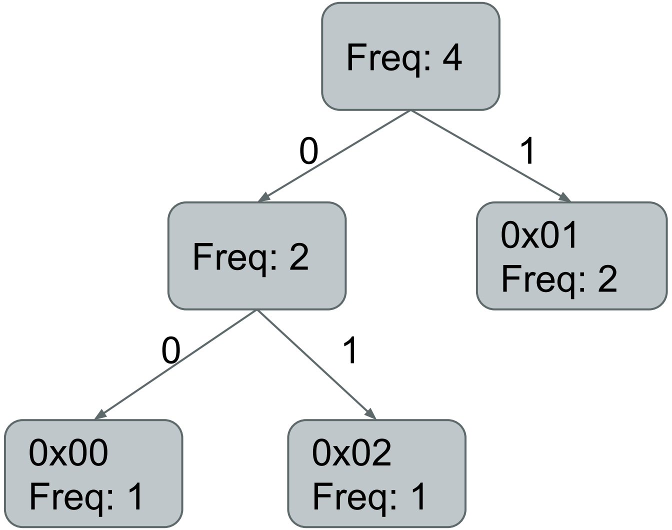
- Step 2: Build Huffman Tree

Symbol	Freq
0x00	1
0x01	2
0x02	1



Sequential Compression

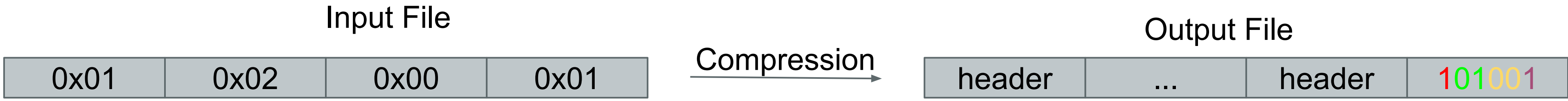
- Step 3: Build Prefix Code Table from Huffman Tree



Byte	Code
0x00	00
0x01	1
0x02	01

Sequential Compression

- Step 4: Encode file using Prefix Code



Input File

0x01 | 0x02 | 0x00 | 0x01

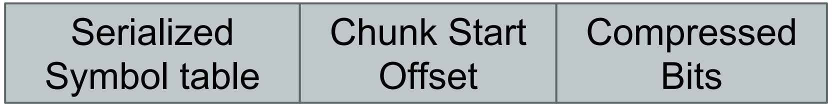
Compression

Output File

header | ... | header | 101001

Byte	Code
0x00	00
0x01	1
0x02	01

Compression Ratio : 6 / 32 = 0.1875



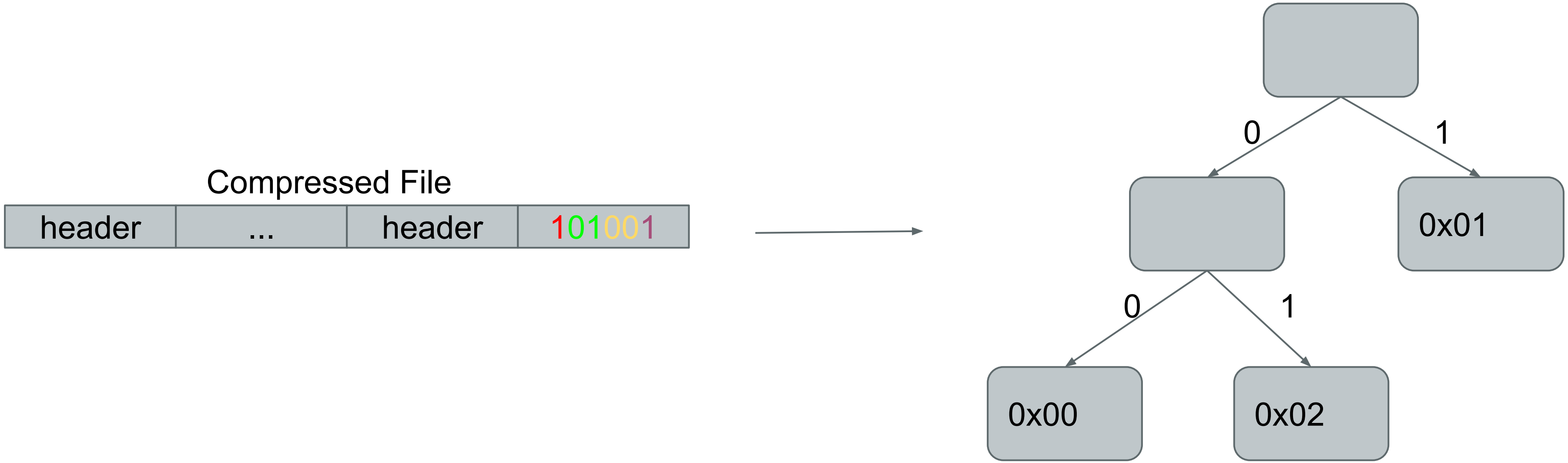
Serialized Symbol table

Chunk Start Offset

Compressed Bits

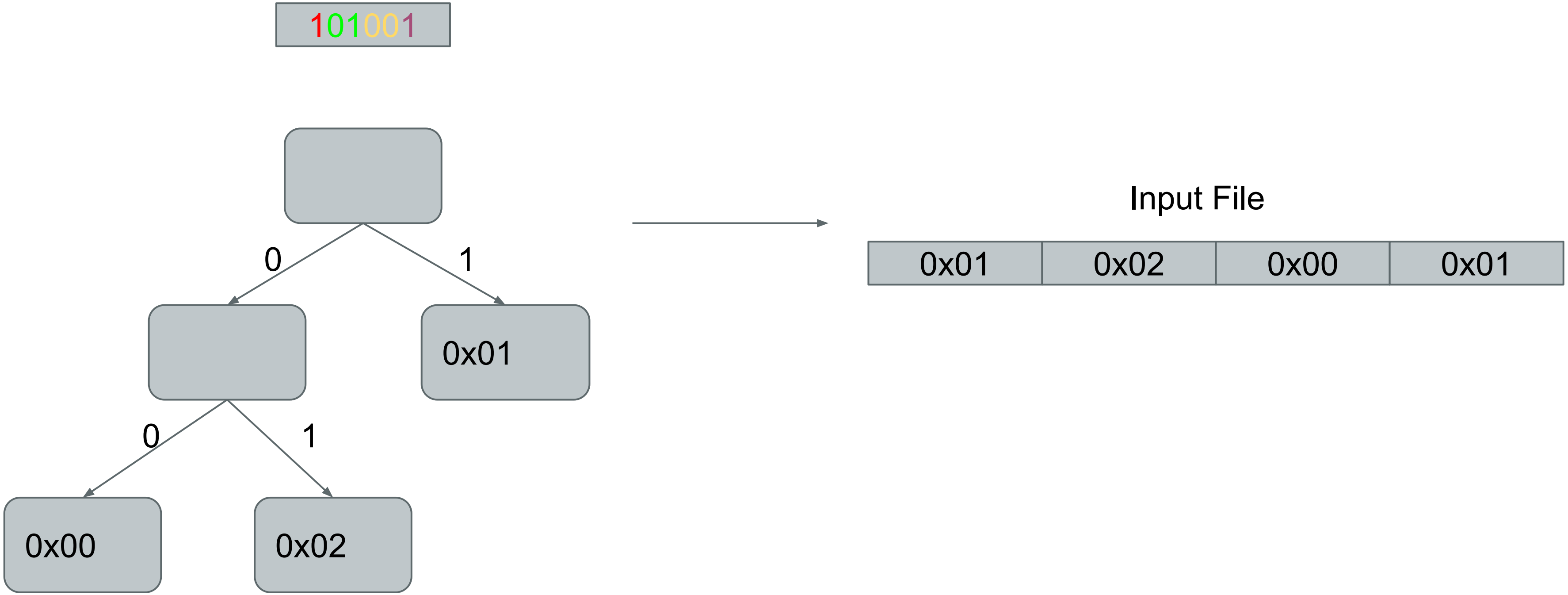
Sequential Decompression

- Rebuild Huffman Tree



Sequential Decompression

- Decompress file using Huffman Tree

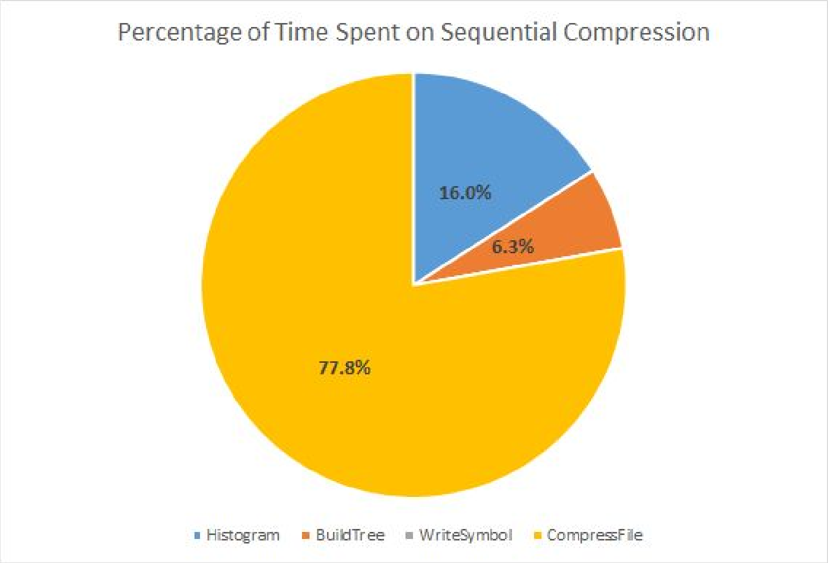


Input File

0x01	0x02	0x00	0x01

Sequential Compression Bottlenecks

- 78% of the Time Spends on the Second Pass of the Data to Do Compression
- 16% of the Time Spends on Generating Symbol Histogram
- 3% of the Time Spends on Building Huffman Tree



Xeon Phi (KNL) Co-processor (68 Cores, 256 Threads), 5.5GB Wiki dataset

Sequential Decompression Bottlenecks

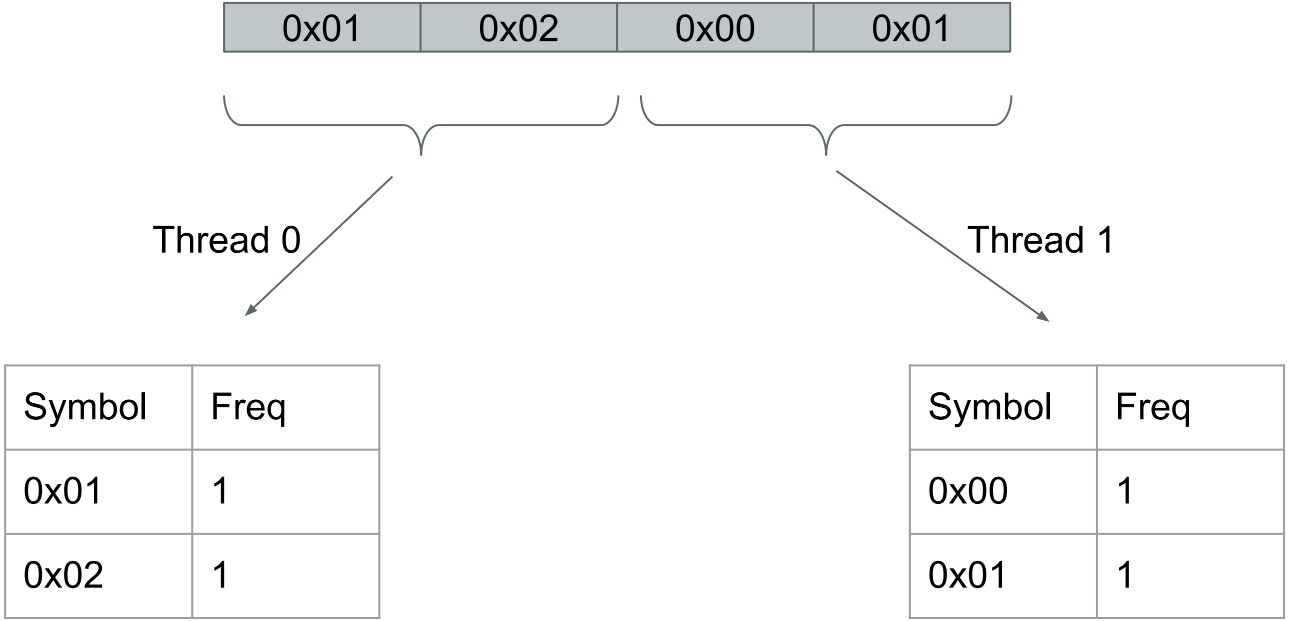
- 0.2% of the time is spent on building Huffman Tree
- 99.8% of the time is spent on decoding files

Our Solution

- Compression
    - Build Symbol Frequencies Histogram (Parallel)
    - Build Huffman Tree and Build Prefix Code Table (Sequential)
    - Compress File using Prefix Code Table (Parallel)
- Decompression
    - Build Huffman Tree (Sequential)
    - Decompress File by traversing Huffman Tree (Parallel)

Parallel Compression

- Step 1: Build Symbol Frequencies Histogram



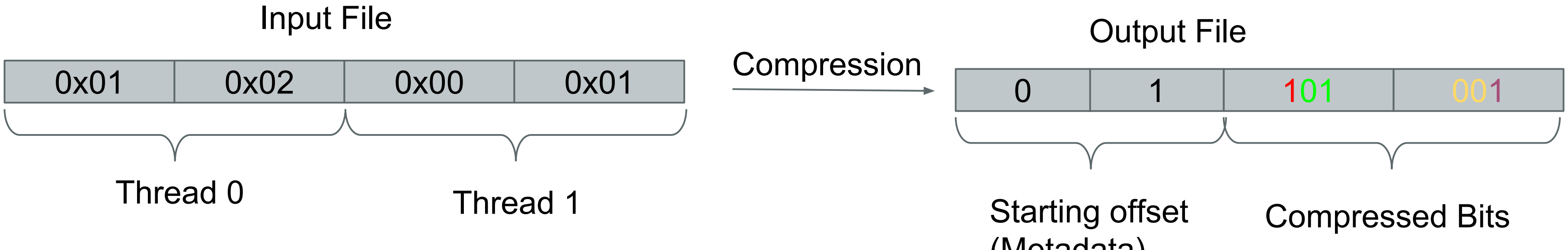
0x01	0x02	0x00	0x01

Symbol	Freq
0x01	1
0x02	1

Symbol	Freq
0x00	1
0x01	1

Parallel Compression

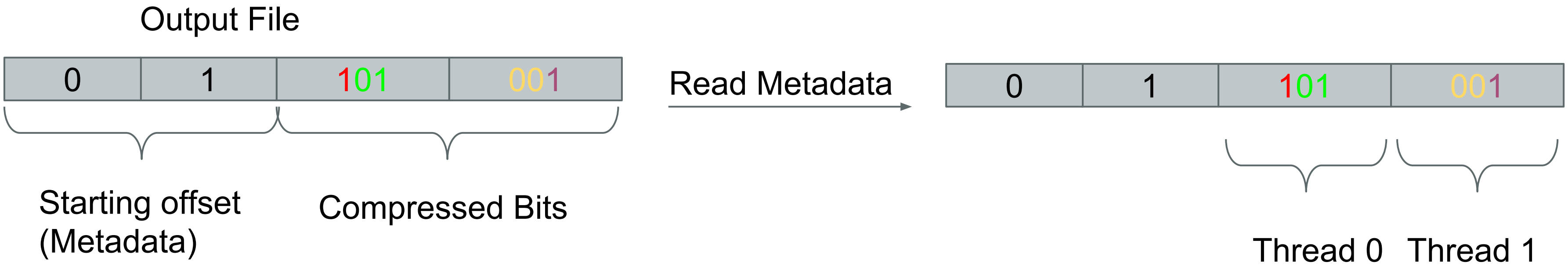
- Step 4: Encode file using Prefix Code



Byte	Code
0x00	00
0x01	1
0x02	01

Parallel Decompression

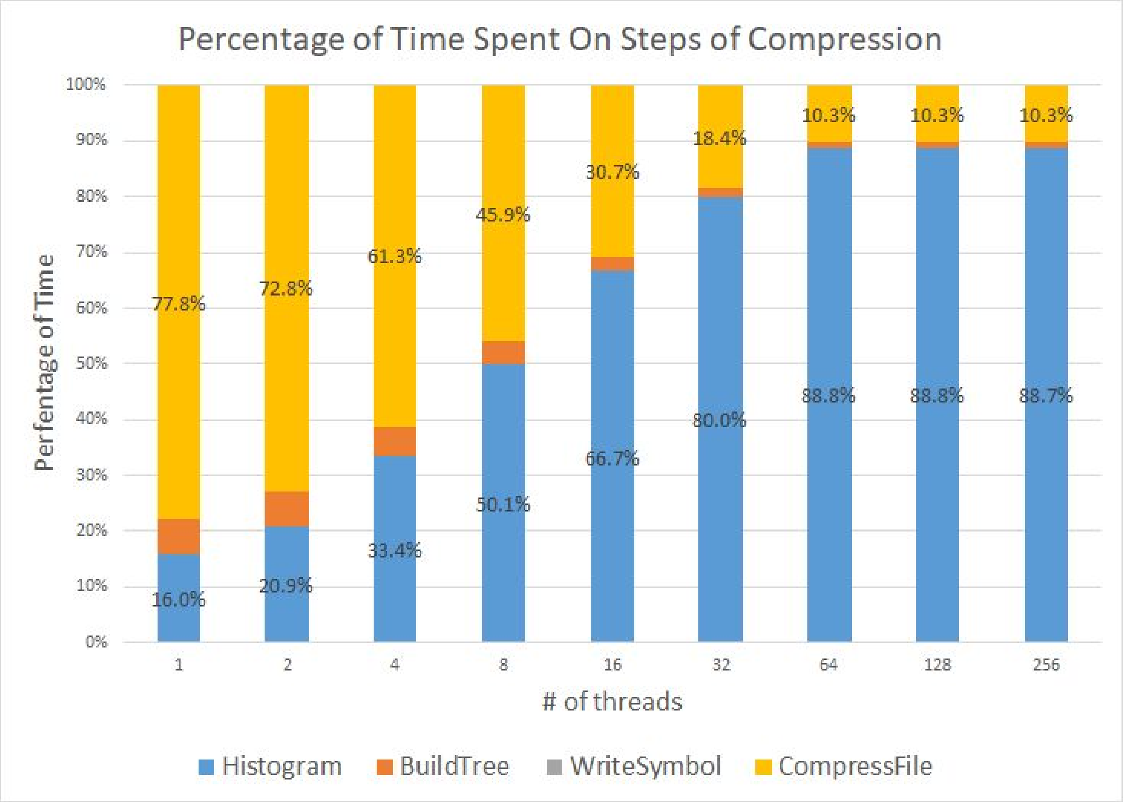
- Step 2: Decompress File using Huffman Tree



Evaluation Setup

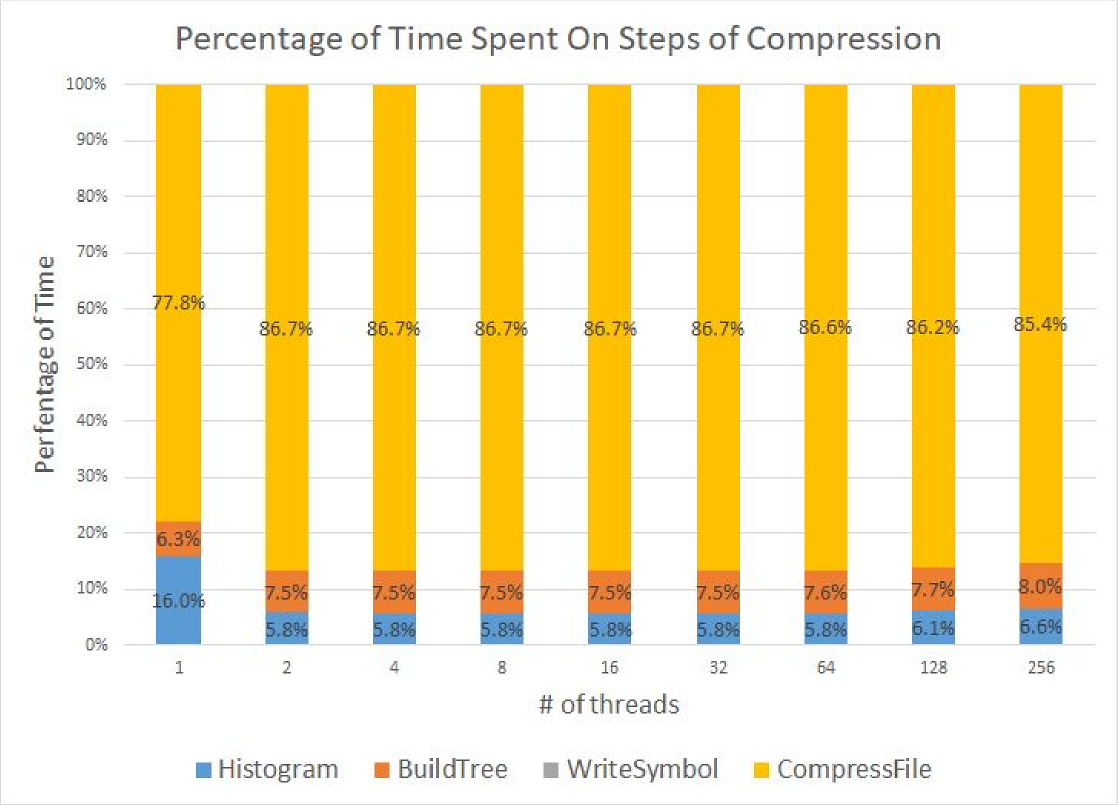
- Xeon Phi (KNL) Co-processor
  - 68 Cores, 256 threads
- Xeon E5-2699 v4 @2.20GHz
  - 88 Cores, NUMA Architecture (22 Cores per socket)

Parallel Encoding Only



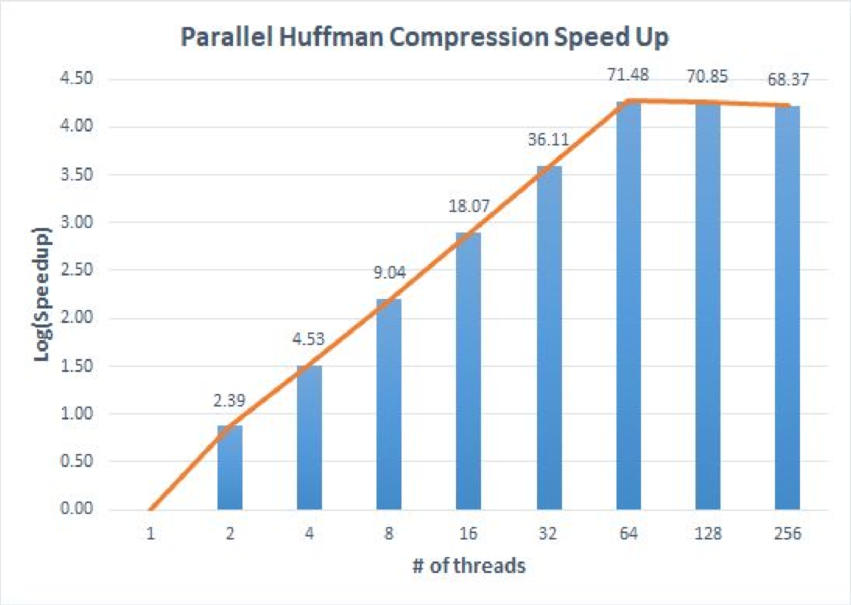
Xeon Phi (KNL) Co-processor (68 Cores, 256 Threads), 5.5GB Wiki dataset

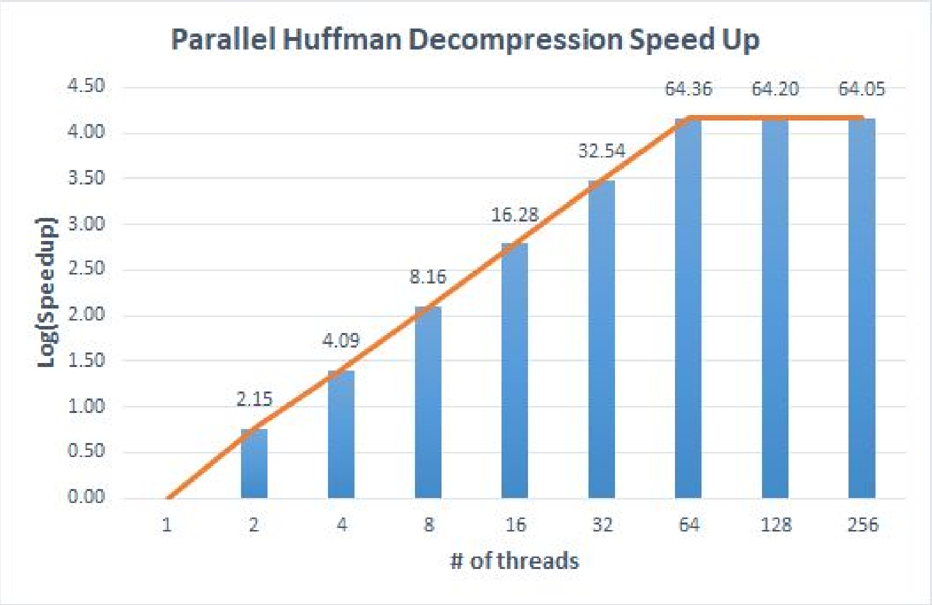
Parallel Encoding + Parallel Histogram



Xeon Phi (KNL) Co-processor (68 Cores, 256 Threads), 5.5GB Wiki dataset

Speedup on Xeon Phi - Linear Speedup



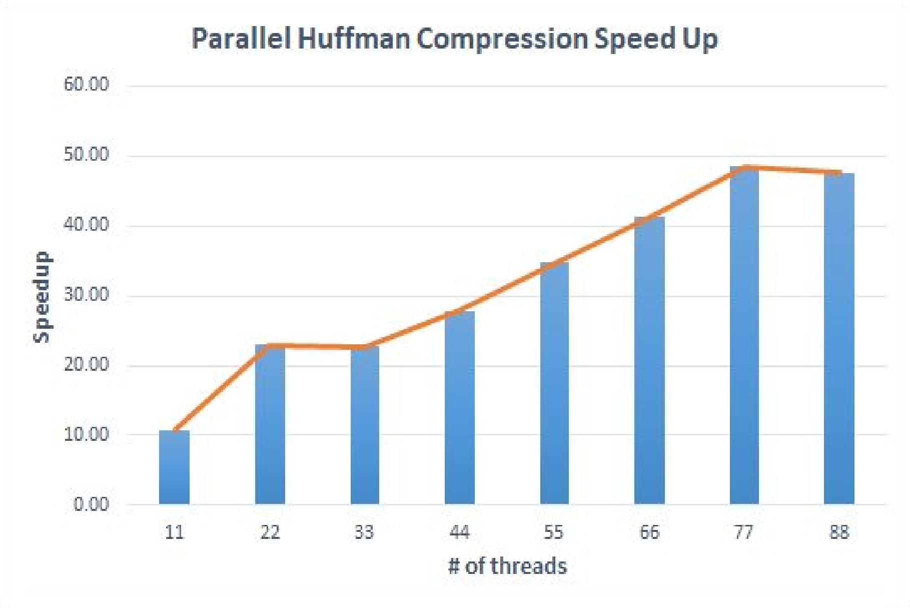


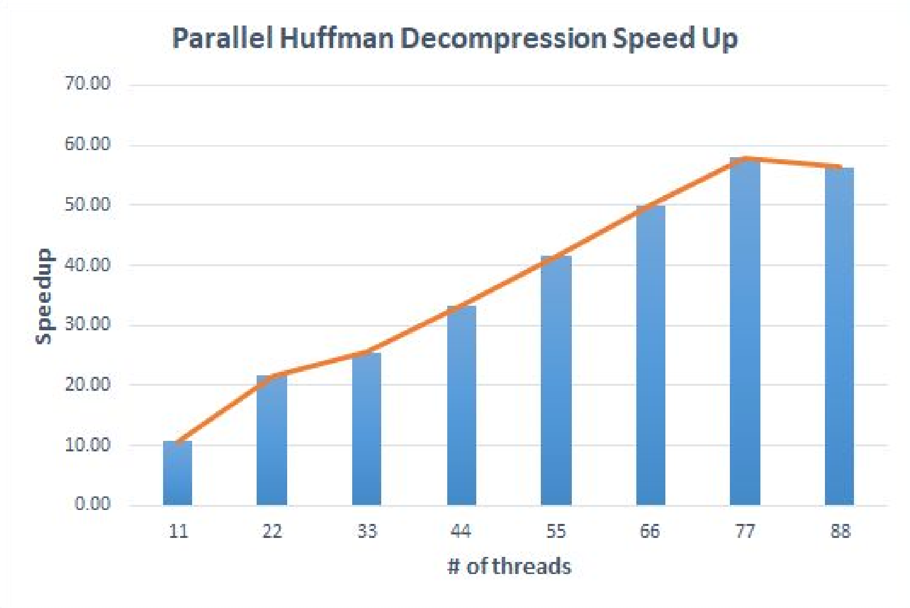
Xeon Phi (KNL) Co-processor (68 cores, 256 threads)

Speedup Analysis

- Memory Bandwidth
  - Symbol list is 256 * 24 = 6144 bytes. Reading sequentially from input file
  - Total Working Set Size: ~6kb. L1 Cache Size: 64 kb
- Workload Balance
  - Each thread finishes roughly at the same time. Only 5% difference.

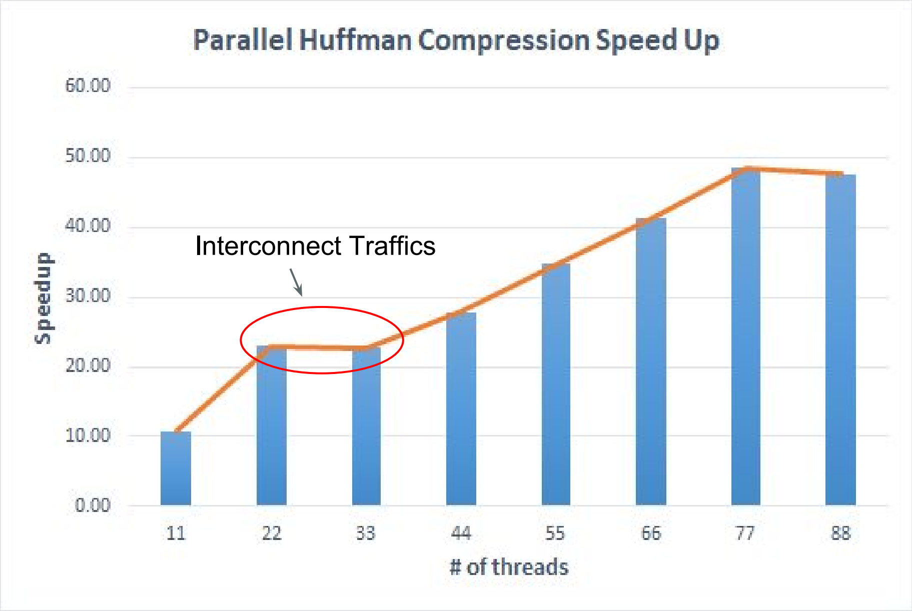
Speedup on Xeon E5-2699 (NUMA)

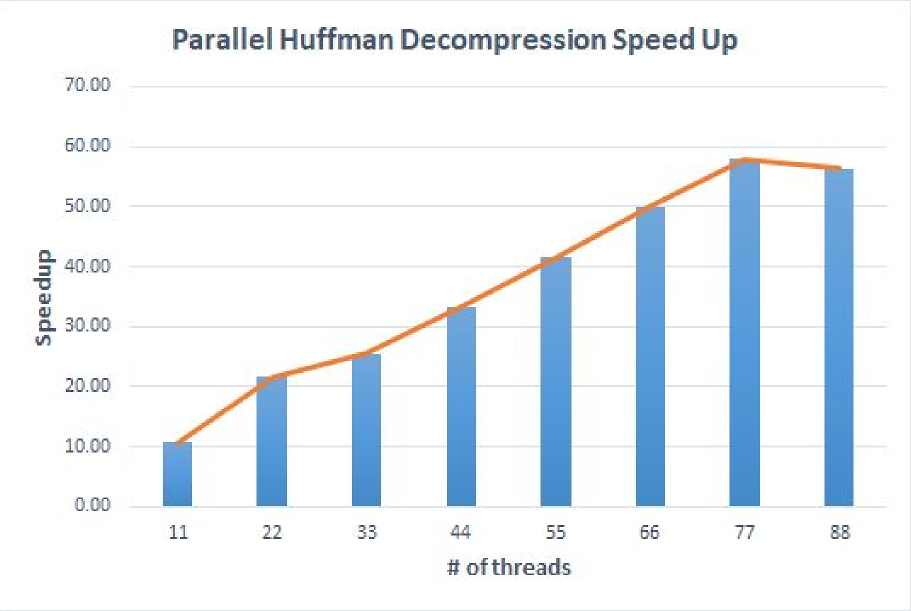




Xeon E5-2699 v4 @2.20GHz (88 Cores, 4 NUMA Socket)

Speedup on Xeon E5-2699 (NUMA)





Xeon E5-2699 v4 @2.20GHz (88 Cores, 4 NUMA Socket)

Future Work

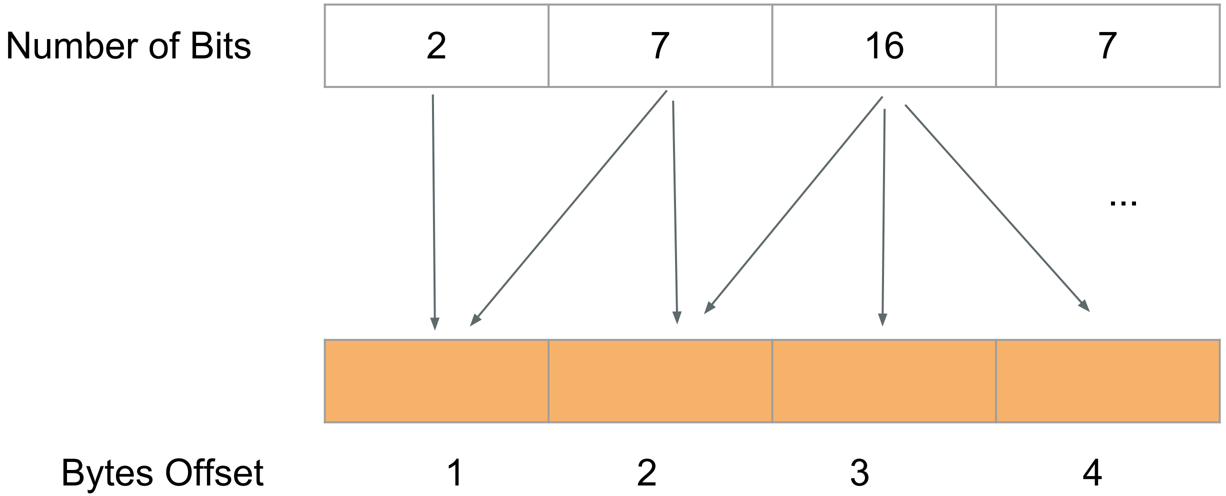
- Modify our compression algorithm to work better on NUMA architecture
    - Each thread reads a file chunk into its local memory.
- Adapt similar techniques to other compression algorithms

Other Alternatives

- ISPC and SIMD instructions
    - Huffman Compression is not a perfect workload for SIMD
    - Why?

Bit-level Conflicts

- ISPC and SIMD instructions
  - ○ Huffman Compression is not a perfect workload for SIMD
  - ○ Why?



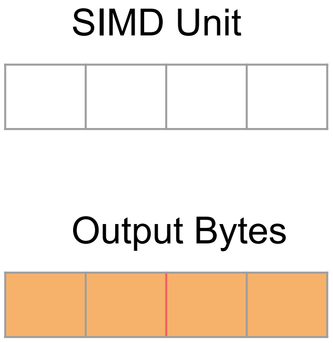
Number of Bits

2 | 7 | 16 | 7

...

Bytes Offset

1 | 2 | 3 | 4

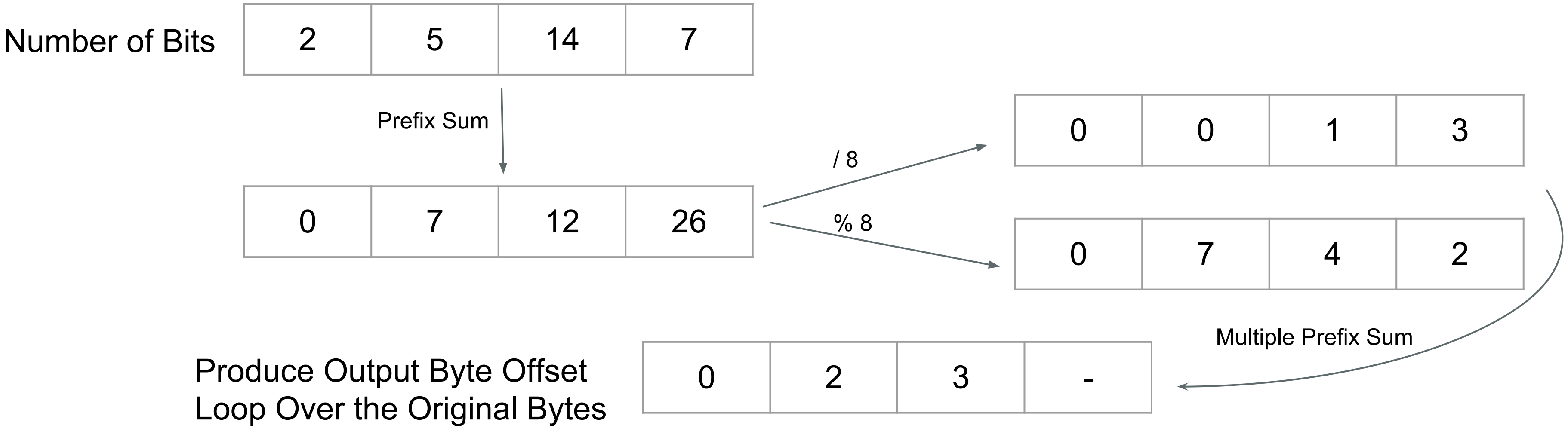


SIMD Unit

Output Bytes

More instructions

- ISPC and SIMD instructions
  - Handle structs is tricky
  - Want each SIMD write independent bytes, but require many instructions



Number of Bits

2	5	14	7

Prefix Sum

0	7	12	26

/ 8

0	0	1	3

% 8

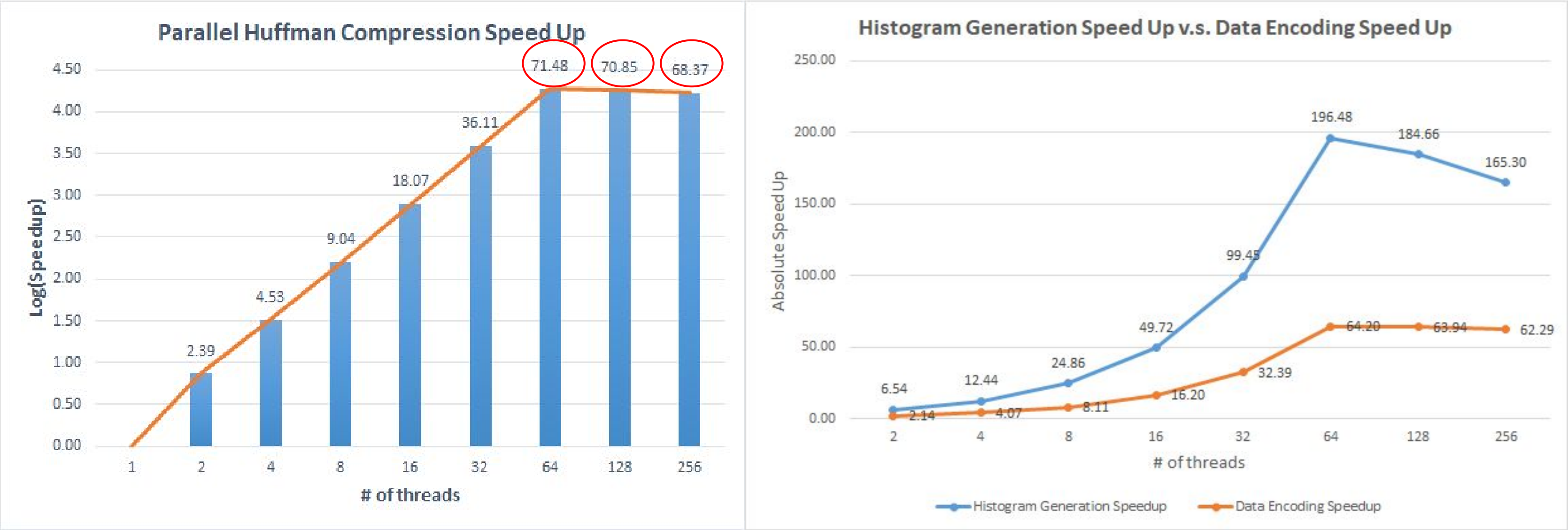
0	7	4	2

Multiple Prefix Sum

Produce Output Byte Offset
Loop Over the Original Bytes

0	2	3	-

Backup Slide: Speedup on Xeon Phi



Xeon Phi (KNL) Co-processor (68 cores, 256 threads)