Accelerate Dictionary Encoding Based Compression on GPU Platform
An Example Dictionary Encoding Algorithm

Xinjie (Edward) HU
Simon Fraser University
Burnaby, Canada
xha102@sfu.ca

PVLDB Reference Format:
Xinjie (Edward) HU. Accelerate Dictionary Encoding Based Compression on GPU Platform
An Example Dictionary Encoding Algorithm. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at.

1 INTRODUCTION

1.1 Features of GPU Architecture

Graphic Processing Unit (GPU) has been widely used in Artificial Intelligence, Machine Learning areas for accelerating scientific computing purpose. It is because of its large scale parallelized computing capability. Thanks to the multi-core design and Single Instruction, Multiple Threads (SIMT) architecture, GPU is tailored for applications which demand many threads working together, with each thread performing very similar but independent operations. Each GPU thread should have simple control flow and its execution be mostly independent to other threads.

Compared with the classical architecture of the Central Processing Unit (CPU) plus main memory, apart from the massive parallelism, GPU also enjoys an order-of-magnitude higher memory bandwidth than CPU. Nevertheless, there are also strong reasons for preventing using GPU over CPU. The device memory capacity in GPU is quite limited, with its highest record reaching only 141 GB [1] till of now [5]. As a comparison, it is very common to equip a computer system with terabyte (TB) level main memory. Additionally, data transmitting overhead across main memory and device memory is high, measured by the bandwidth of Peripheral Component Interconnect Express (PCIe) bus which reaches only hundreds of gigabytes (GB) per second. It is orders-of-magnitude lower than the bandwidth of device memory. That indicates not every use case benefits from shifting the CPU architecture to GPU as the data transmitting work already does not worth it.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

[1] Nvidia's H200 GPU accelerator

1.2 Applying GPU on Data Compression

Data compression is a critical technology for reducing storage cost and improving data transmission efficiency, especially in domains such as large-scale databases, multimedia streaming, and scientific simulations. In Database Management System (DBMS) area, data compression is always used to store bulk size documents, binary objects (videos, images) and data stored in columnar format.

There are great potentials of applying GPU to accelerate data compression process because of: 1. some of the compression algorithms can be designed and executed in parallel, 2. it indirectly achieves filling more data to saturate the limited bandwidth of PCIe. Till now, there are already research works focusing on accelerating data (de)compression with utilising GPU in DBMS cohort. For instance, tile-based decompression execution is developed to keep intermediate results in shared memory and a new GPU-optimized compression format is designed to fully saturate GPU memory bandwidth during decompression [7].

1.3 Benchmark: Simple Dictionary Encoding

To prove the great potential of applying GPU, this research selects a simple dictionary encoding algorithm as the benchmark compression method. It is worth to be mentioned that this algorithm is a simplified version of classical dictionary-based compression algorithms (for example, Lempel-Ziv-Welch (LZW) [8] and Huffman coding [6]) in order to limit the focus on several key steps for further GPU optimizations, instead of keeping all details that could make the problem unnecessarily much more complicated. Although this may sacrifice some level of performance, it is expected that this benchmark is the best prototype to show the value of applying GPU while still achieving acceptable compression performance.

The core idea of this algorithm is to use fewer bits to replace those symbols (characters) that present frequently in the file. For example, one character originally takes 8 bits for representation in the system. If 16 frequent characters are selected, only 4 bits (2^4) are enough to represent those characters. With an additional bit used to mark whether or not the following bit stream is encoded, in total 5 bits are enough to replace the 8 bits representation. The procedures of this algorithm is listed below. Detailed compression and decompression algorithm described in pseudocode is presented in Algorithm 1 and Algorithm 2 respectively.

1.3.1 Procedures.

(1) Preselect a collection of frequently shown symbols, construct mapping relationship between each character and its encoded bit value.

(2) Copy original file data from host to device.
(3) Divide the file stream into chunks, each thread encodes a chunk.
(4) Calculate the offset of each encoded data chunk on the output bit stream buffer (as encoded data varies on size, this step is necessary).
(5) Load encoded chunks to the byte stream according to the offset.
(6) Copy back the byte stream from device to host.

Algorithm 1 Compression Process

Input: Data block d_{input}, size input_size
Output: Compressed data $d_{\text{final\_out}}$
Step 1: Parallel Reading and Bit-Length Calculation
for each chunk chunk_id in d_{input} **do**
  for each symbol symbol in the chunk **do**
    if symbol is in dictionary **then**
      Assign 5-bit encoding
    else
      Assign 9-bit encoding
    end if
    Store the bit-length of the symbol
  end for
end for
Step 2: Prefix Sum Calculation and Bit Packing
for chunk chunk_id **do**
  if thread 0 **then**
    Calculate total bit size for chunk
    Initialize temporary output buffer
  end if
  for each symbol in chunk **do**
    if symbol is in dictionary **then**
      Apply 5-bit packing
    else
      Apply 9-bit packing
    end if
  end for
end for
Step 3: Compute Byte Offsets
for chunk chunk_id **do**
  Calculate byte offset based on bit size
end for
Step 4: Assemble Final Compressed Output
Collect all compressed chunks using byte offsets into $d_{\text{final\_out}}$
Step 5: Save Compressed Data to File
Write $d_{\text{final\_out}}$ to the file

There are some opportunities in leveraging GPU to accelerate this algorithm. In step 1, the dictionary used to store the mapping relationship from character to bit value is a pre-determined variable. Since no modifications are to be made on this dictionary during the whole execution runtime, it could be load to constant global device memory, which is a cached and read-only memory block shared by GPU threads. Additionally, both step 3 and step 5 can be executed by multiple threads in parallel. In step 3, each thread is assigned with a data chunk and iteratively go through the character

Algorithm 2 Decompression Process

Input: Compressed data $d_{\text{compressed}}$, chunk bit sizes $d_{\text{chunk\_bit\_sizes}}$, byte offsets $d_{\text{chunk\_byte\_offsets}}$
Output: Decompressed data d_{output}
Step 1: Read Compressed Data
for each chunk chunk_id **do**
  Read compressed data from byte offset $d_{\text{chunk\_byte\_offsets}}$[chunk_id]
end for
Step 2: Bit-Level Decoding
for each symbol in chunk **do**
  Read the first bit to get the flag
  if flag = 1 **then**
    Read the next 4 bits to get the dictionary index
    Retrieve the character from the dictionary
  else
    Read the next 8 bits as the original character
  end if
end for
Step 3: Write Decompressed Data
for each decoded symbol **do**
  Write the symbol to the decompressed output buffer
end for
Step 4: Repeat for All Chunks
Continue the decompression process for all chunks
Step 5: Save Decompressed Data to File
Write the final decompressed data to a file

to check whether it hits the dictionary key. If the key is matched, the corresponding bit string value replaces the original character. This process is carried out independently within each thread and is not intervened by other threads. More details on step 4 is explained in section 2.3.

2 KEY FEATURES

This research particularly introduces the following three techniques adopted on GPU to better boost the performance of compression, namely Lookup Dictionary (LD), Memory Coalesced Access (MCA) and Parallelized Assembly (PA).

2.1 Lookup Dictionary (LD)

As mentioned above, a dictionary mapping each frequently shown character to corresponding bit value is constructed and loaded to the global constant memory in GPU. The time complexity of looking up the dictionary is $O(1)$. As a comparison, the unoptimised implementation would be executing a `for` loop to iterate over a list or vector to check whether hits or not. That significantly cost much time than using Lookup Dictionary.

2.2 Memory Coalesced Access (MCA)

Dictionary-based algorithms require frequent global memory accesses to load data chunks to thread local memory. If this is not designed in a curated manner, memory access is prone to non-coalesced pattern which results in high latency and a waste of GPU memory bandwidth. Therefore, another potential breakthrough lies

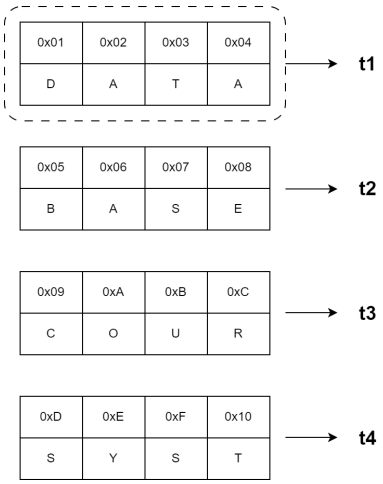

Figure 1: Horizontal Partitioning

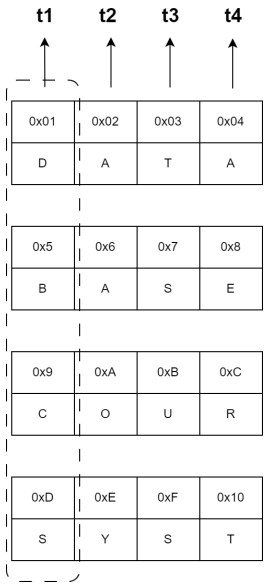

Figure 2: Vertical Partitioning

on aligning thread memory accesses to leverage GPU coalescing capabilities. Preliminary analysis suggests that conventional block-level parallelism (horizontal partitioning) may not fully exploit GPU memory bandwidth due to fragmented access patterns. As illustrated in Figure 1, there are 4 data blocks to be compressed, and each block contains 4 characters (symbols). Suppose *t1, ... , t4* are within the same warp (the minimum scheduling unit in GPU program). Although the memory addresses within each block are consecutive, the memory addresses gained by each thread are not, which prevents those threads from accessing global memory simultaneously.

Instead, alternative data partitioning strategies, such as vertical or columnar partitioning, could enable contiguous memory access across threads. For example, Figure 2 shows one of the possible vertical partitioning strategy under the same example. If the program is well written, the compiler can guide the hardware to combine multiple access into a coalesced one to feed the data block to *t1, ..., t4* simultaneously. By coalescing memory access, the system can achieve higher throughput.

2.3 Parallelized Assembly (PA)

As the data is encoded by each GPU thread, encoded data chunks need to be loaded back to a byte stream buffer stored in device memory. Then the buffer is to be copied back to the counterpart in the host main memory. The detailed breakdown of each step in this stage is illustrated in Figure 3. Firstly, a dedicated thread is adopted to calculate the offset of each encoded data chunk. That gives the information of where the data chunk needs to be placed in the buffer. After knowing the offset of each chunk, the total length of the expected byte stream can be calculated immediately, which triggers to assign a buffer with that length in the device memory. The only remaining work left is to load encoded data to the buffer to assemble the byte stream. Since there is a one-to-one relationship between each thread and each data chunk, and each thread is able to navigate the destination according to the offset, this assembly work can be carried out independently and in parallel. That is the basic idea of Parallel Assembly. Eventually, the offset table, encoded byte stream buffer along with the Lookup Dictionary will be copied back to host and stored persistently.

3 EXPERIMENT

3.1 Implementation

We carried out experiments aiming to achieve the following goals:

(1) Prove the effectiveness of simple dictionary encoding algorithm.
(2) Prove the advantage of adopting GPU over CPU.
(3) Measure the contribution brought by each of the three features developed on GPU platform through ablation study
(4) Learn how the system performance is boosted from micro-architecture level through profiling case study.

The configurations of our experiment are as follow:

- Hardware: Nvidia GPU hosted on free-tier Google Colab. Model unknown.
- Development Tools: NVCC V12.2.140, g++ 11.4.0, CMake 3.22.1.
- Profiling Tool: Nsight Compute [2] provided by Nvidia.
- Workload: Wikipedia top 1000 most-viewed articles concatenated to one single text file (in .txt format). In total the file takes about 28 MB.
- Frequently Shown Character Set: For the Lookup Dictionary, 16 fixed characters are preselected according to rule-of-thumb. For example, vowels and space character are selected because they always show up more frequently than others.
- Hyperparametres on GPU: Each thread is set to process a data chunk with size 1024 characters. The block dimension in device is 256.

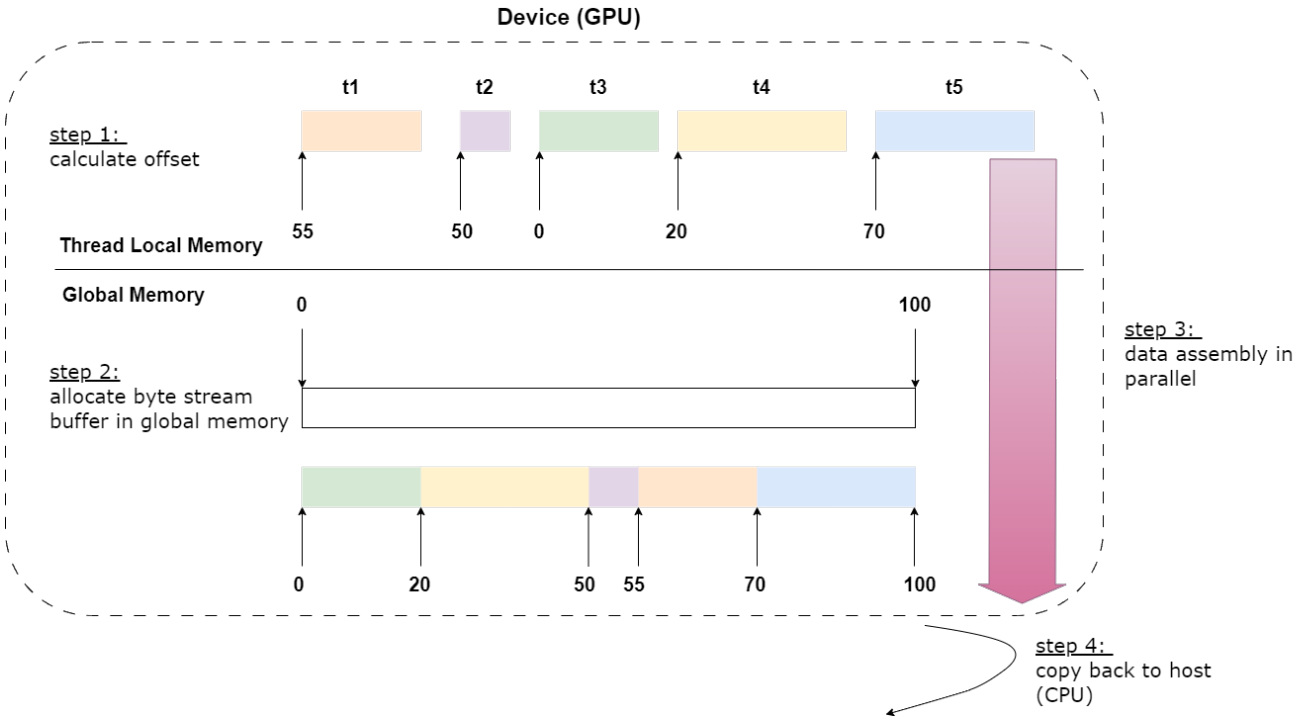

Figure 3: Assembly Process for Encoded Data

4 EVALUATION

4.1 Effectiveness of Simple Dictionary Encoding Algorithm

Space saving ratio is a metric to measure the percentage of space saved after using data compression. It is calculated by dividing the size after compression by the original file size. Figure 4 shows the histogram of the space saving ratio achieved by pure CPU implementation and GPU implementation respectively, with utilising the same simple dictionary encoding algorithm mentioned in section 1.3. Both implementations achieve comparable performance which is expected due to the final space saving performance should be independent to the platform used for implementation. In fact, it is only determined by the logic of compression algorithm. On average, the space saving ratio is around 28%, which is a decent performance given the complexity of this simple dictionary encoding algorithm is much lower than outperforming algorithms like snappy [4]. The underlying reason of the success of this algorithm lies on the highly skewed distribution of characters in natural language.

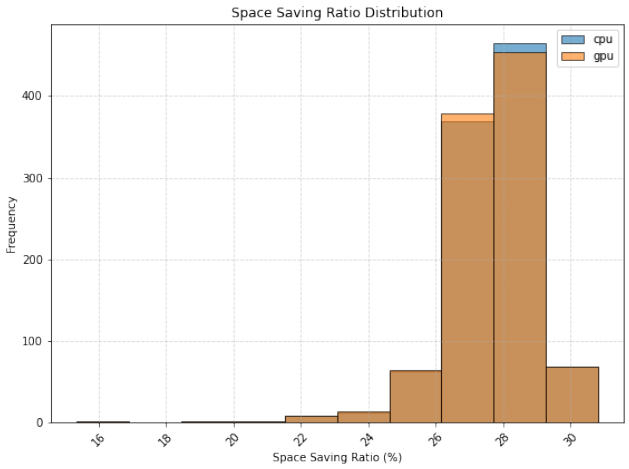

Figure 4: Histogram of Space Saving Ratio Achieved by CPU and GPU Implementation

4.2 Ablation Study

The detailed breakdown of end-to-end compression and decompression latency is presented in Table 1. The compression process is further divided into several stages for better understanding. Column CPU shows the latencies in CPU implementation, while other columns are all in GPU implementation, with different combinations on the features utilised.

Overall, all experiments performed on GPU platform enjoys lower latencies on both compression and decompression process. Our main focus is to study the features contribute most to latency reduction on GPU platform. As Lookup Dictionary feature is disabled, the latency of character replacement significantly increases, while further turning off Memory Coalesced Access feature does not change the latency at all. That shows the Lookup Dictionary plays an important role in accelerating the encoding process thanks to its $O(1)$ lookup time complexity and global constant memory

Table 1: Compression & Decompression Latency Breakdown (unit: ms)

Stages	All ON	OFF: LD	OFF: LD&MCA	OFF: LD&PA	All OFF	CPU
Host to Device	42	36	36	35	35	816
Character Replacement	92	**132**	132	132	132	
Compute Offset	5	5	5	5	5	
Assemble Data	9	9	9	**370**	371	
Device to Host	6	6	5	6	6	
Compression Latency	**157**	**191**	**192**	**552**	**553**	**816**
Decompression Latency	**118**	**117**	**117**	**117**	**117**	**586**

access pattern. Even though the logic of memory coalescing is already explicitly coded by grouping consecutive threads to process consecutive characters, MCA still fails. Its failure is possibly due to that the hyperparametres including block size and chunk size are not set in an optimal way, which causes the stride being too big to trigger memory coalescing.

In Assemble Data stage, by turning off the switch of Parallel Assembly, the latency significantly increases by 40 times from 9 ms to 370 ms. It strongly shows the importance of fully leveraging the parallelism capability of GPU, or else the program returns back to serialized execution which loses the motivation of utilizing GPU.

4.3 Case Study: Lessons Learned from Profiling

Nvidia provides Nsight Compute (ncu) and Nsight System (nsys) [3] to profile the programme in kernel level and programme level, respectively. This section shows an example of metrics gained from ncu to better understand what Lookup Dictionary brings to the kernel of character encoding. Table 2 compares key metrics from *GPU Speed Of Light Throughput* section in the profiling report before and after applying the Lookup Dictionary. The optimization has resulted in significant performance improvements across multiple metrics. Major improvements are: 23% faster execution, 35% better memory utilization, 4% better compute utilization, 24% reduction in cycles. Overall, the results suggest that the current memory access pattern is reasonably optimized.

5 CONCLUSION

As GPU platform is being applied to many areas for boosting performance purpose, this works specifically focuses on its potential on accelerating data compression process. A simple dictionary encoding algorithm is designed to serve as a benchmark. Particularly, Lookup Dictionary, Memory Coalescing Access and Parallel Assembly are the three features whose contributions are mainly studied. With leveraging official profiling tool provided by GPU vendor, detailed micro-architecture level metrics are measured to learn how the performance is boosted. The main discoveries can be concluded from this work are:

(1) Dictionary encoding (or bit-packing) is an efficient, economic compression algorithm, given its conciseness and easy-to-implement.
(2) GPU generally outperforms CPU implementation in both data compression and decompression process.
(3) Features including Lookup Dictionary, Parallel Assembly significantly boost performance.
(4) Profiling result shows that the optimization appears to have focused on reducing instruction count and improving memory access efficiency, with good results.

However, this work is by no means the end of the story. There are many further tasks could be done to make improvements. One of the directions is to further tune hyperparametres in the algorithm (for example, block size and chunk size), or use shared memory to benefit from cache tuning and achieve better memory reuse to reduce DRAM traffic. Another direction is to apply it to a real query processing scenario to benefit real-world users. Combined with GPUDirect [1] technology developed by Nvidia, the data compression could be used to decrease end-to-end latency and increase the throughput when flushing pages and logs to persistant storage.

Table 2: Performance Comparison with and without LD

Metric	Metric Unit	without LD	with LD
Elapsed Cycles	cycle	7,606,703	5,838,846
Memory Throughput	%	39.08	52.51
Duration	ms	3.40	2.61
L1/TEX Cache Throughput	%	34.43	52.93
L2 Cache Throughput	%	39.08	52.32

REFERENCES

[1] [n.d.]. Magnum IO GPUDirect Storage. https://developer.nvidia.com/gpudirect-storage
[2] [n.d.]. NVIDIA Nsight Compute. https://developer.nvidia.com/nsight-compute
[3] [n.d.]. NVIDIA Nsight Systems. https://developer.nvidia.com/nsight-systems
[4] 2025. google/snappy. https://github.com/google/snappy original-date: 2014-03-03T21:58:09Z.
[5] 2025. List of Nvidia graphics processing units. https://en.wikipedia.org/w/index.php?title=List_of_Nvidia_graphics_processing_units&oldid=1284113523#Data_Center_GPUs Page Version ID: 1284113523.
[6] David A. Huffman. 1952. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE* 40, 9 (1952), 1098–1101. https://doi.org/10.1109/JRPROC.1952.273898
[7] Anil Shanbhag, Bobbi W. Yogatama, Xiangyao Yu, and Samuel Madden. 2022. Tile-based Lightweight Integer Compression in GPU. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1390–1403. https://doi.org/10.1145/3514221.3526132
[8] Welch. 1984. A Technique for High-Performance Data Compression. *Computer* 17, 6 (1984), 8–19. https://doi.org/10.1109/MC.1984.1659158