ESE532 Project Report

Deduplication and Compression

Akhil Gunda, Ruturaj Nanoti, Siddhant Mathur
December 2023

Contents

1 Single ARM Processor Mapped Design

a) The Deduplication and Compression application on a single ARM processor mapped design is described step wise as shown below:

   1. The data, ethernet input, is received in real time using ethernet cable from the client machine and stored in local storage, a buffer, using APIs provided in the project.
   2. The ethernet input data is then moved into a local input buffer of size 16 KB (Number of Packets * Size of each packet = 16 * 1KB).
   3. When the 16 KB buffer is filled, the compression pipeline function is invoked. This compression pipeline includes sub functions (CDC, SHA, DEDUP, LZW and final bit packing) required for compression application.
   4. The operations inside compression pipeline take places sequentially which includes:
      - The input data of 16 KB is passed through the content defined chunking function. The CDC function starts calculating hash value with a window size of 16 (WIN_SIZE) on the incoming packets. If the hash value matches with the target hash value of 1024, (MODULUS), a chunk is generated. The end index of the chunk is stored and starts for the next chunk. Finally all the chunk boundaries are stored in a vector and moved to the next function.
      - The SHA256 function starts calculating the fingerprints for the chunks generated. Next the DEDUP function uses an `unordered_map`. It checks if the SHA fingerprint is already present in the unordered map and if not, it assigns each unique chunk a chunk ID and calls LZW on it. If found, just returns the index of the map where it was previously added.
      - If a chunk was not found, the LZW function is called which accepts 16 KB input data along with the chunk boundaries. The LZW sends out the compressed data. Then the compressed data is bit packed and stored in a file.
      - Iterate until all the chunks are traversed.
   5. Steps 3 and 4 are repeated for all the packets received via Ethernet. At the end the final compressed binary file is generated.

b) Following are the parameters that were used in our initial implementation.

   - **CDC:** The table below shows the default parameters used for CDC.

WINDOW SIZE	16
MODULUS	1024
PRIME	3
TARGET	0
MAX CHUNK SIZE	8192

Table 1: CDC Parameters

- **SHA:** The following table shows the default parameters used for SHA/DEDUP.

SHA256 BLOCK SIZE	32

Table 2: SHA Parameters

  This is set to 32 since the SHA256 digest is 32 byte/256 bit long.

- **DEDUP:** For Insertion and lookup in Deduplication, we used an unordered_map for mapping SHA fingerprints to chunk IDs. The benefit of using unordered_map was that they gave us O(1) lookup time.
  Datatype - The SHA fingerprints were generated as a byte array. When this was used as a key to the ordered_map, we were getting a lot of collisions in the Deduplication stage. To counter this, we converted the byte array to a string since an unordered_map needs a key that can be represented as a single data type to hash into the table.

- **LZW:** Initially we used a hash table of size 65536 and 1 bucket. We used a random hash function initially that consisted of some random bit shifts and manipulations. However, we decided to change the hash function in an attempt to decrease the size of the hash table.

  Hash function - We eventually moved to the murmur hash function using which we were able to reduce the size of the hash table to 8192. However, for bigger chunks we were still getting collisions, hence, we increased the buckets to 2 that were used as a 2D array.

- **Overall Application:** Number of Packets, We were initially receiving 16 packets since we were using a block size of 1024.

c) Key Performance Achieved on Single ARM Processor Design: The throughput of our application is calculated based on the compression latency. The throughput calculation follows the following formula:

$$ApplicationThroughput = \frac{\frac{BytesReadFromEthernet \times 8}{1000000}}{TotalCompressionLatency}$$

In determining the compression latency, we utilized the stopwatch class by encapsulating the start and stop calls around our top function. This top function encompasses all the sub stages, such as CDC, SHA, DEDUP, LZW and includes writing data to the compressed file. Consequently, the calculation throughput is 6.815 Mb/s, equivalent to 0.006815 Gb/s.

d) The following table shows the command line parameters:

Block Size (-b)	1024
Sleep Parameter (-s)	5

Table 3: Command Line Parameters

e) Compression Achieved: For the test case - LittlePrince.txt file, we achieved a compression ratio of,

$$\frac{CompressedFileSize}{OriginalFileSize} = \frac{11825}{14247} = 0.83$$

e) Breakdown of Time spent on each component:

Task	**Time (Days)**
CDC Implementation	2
SHA/DEDUP Implementation	2
LZW (on ARM Processor) Implementation	5
Main Application and Integration	3
Debugging	4
Testing and Validation	3

Table 4: Breakdown of Time Spent

2 Final Ultra96 Mapped Design

a) Performance Achieved on Ultra96 Mapped Design: The throughput of our application is calculated based on the compression latency. The throughput calculation follows the following formula:

$$ApplicationThroughput = \frac{\frac{BytesReadFromEthernet \times 8}{1000000}}{TotalCompressionLatency}$$

In determining the compression latency, we utilized the stopwatch class by encapsulating the start and stop calls around our top function. This top function encompasses all the sub stages, such as CDC, SHA, DEDUP, LZW and includes writing data to the compressed file.

Application Throughput: Linux.tar (Size: 200273920 B)(Binary File)
The throughput achieved is **71.237 Mb/s**. The intermediate throughputs were 1211.76 Mb/s for CDC, 728.032 Mb/s for SHA, 7593.07 Mb/s for Dedup, and 95.3945 Mb/s for LZW. Our output file is compressed to 95829565 Bytes. These values were determined from the “Linux.tar” test case.

Application Throughput: Franklin.txt (Size: 399054 B)(Text File)
Throughput achieved is **67.443 Mb/s**. The intermediate throughputs were 1178.01 Mb/s for CDC, 713.09 Mb/s for SHA, 8704.41 Mb/s for Dedup, and 92.6546 Mb/s for LZW. Our output file is compressed to 291731 Bytes. These values were determined from the “Franklin.txt” test case.

Terminal Output Linux Tar and Franklin respectively:

```
akhilgunda@Akhils-MacBook-Air Primary Files % ../client_mac -i 169.254.73.63 -b 8192 -f linux.tar -s 2000
ip is set to 169.254.73.63
blocksize is 8192
filename is linux.tar
sleep is set to 2000 optarg
bytes_read 200273920
akhilgunda@Akhils-MacBook-Air Primary Files %
```



```
root@u96v2-sbc-base-2020-2:/media/sd-mmcblk0p1# ./encoder -k encoder.xclbin -c 4096
Kernel name is set to encoder.xclbin optarg
Chunk size is set to 4096 optarg
setting up sever...
server setup complete!
INFO: Found Xilinx Platform
INFO: Loading 'encoder.xclbin'
--------------- Total Latencies ---------------
Total latency of CDC is: 1227.72 ms.
Total latency of LZW is: 15595.2 ms.
Total latency of SHA256 is: 2043.45 ms.
Total latency of DeDup is: 195.929 ms.
Total time taken: 20883.8 ms.
---------------- Average Latencies ------------
Average latency of CDC per loop iteration is: 0.108159 ms.
Average latency of LZW per loop iteration is: 1.37391 ms.
Average latency of SHA256 per loop iteration is: 0.0363733 ms.
Average latency of DeDup per loop iteration is: 0.00348751 ms.
Average latency: 1.83982 ms.


--------------- Key Throughputs ---------------
Ethernet Latency: 37.9941s.
Bytes Received: 185962496B.
Latency for Compression: 28.0863s.
Latency for Compression (without fwrite): 20.8838s.

Latency for CDC: 1.22772s.
Latency for SHA: 2.04345s.
Latency for LZW: 15.5952s.
Latency for DEDUP: 0.195929s.

CDC Throughput: 1211.76Mb/s.
SHA Throughput: 728.032Mb/s.
LZW Throughput: 95.3945Mb/s.
DEDUP Throughput: 7593.07Mb/s.

Ethernet Throughput: 39.1561Mb/s.
Application Throughput: 52.969Mb/s.
Application Throughput (without fwrite): 71.237Mb/s.
Bytes Contributed by Deduplication: 44232B.
Bytes Contributed by LZW: 95785333B.

root@u96v2-sbc-base-2020-2:/media/sd-mmcblk0p1#
```

Figure 1: Linux Tar

```
akhilgunda@Akhils-MacBook-Air Primary Files % ../client_mac -i 169.254.73.63 -b 8192 -f Franklin.txt
ip is set to 169.254.73.63
blocksize is 8192
filename is Franklin.txt
bytes_read 399054
akhilgunda@Akhils-MacBook-Air Primary Files %
```



```
root@u96v2-sbc-base-2020-2:/media/sd-mmcblk0p1# ./encoder -k encoder.xclbin -c 4096
Kernel name is set to encoder.xclbin optarg
Chunk size is set to 4096 optarg
setting up sever...
server setup complete!
INFO: Found Xilinx Platform
INFO: Loading 'encoder.xclbin'
--------------- Total Latencies ---------------
Total latency of CDC is: 2.71003 ms.
Total latency of LZW is: 34.4552 ms.
Total latency of SHA256 is: 4.4769 ms.
Total latency of DeDup is: 0.36676 ms.
Total time taken: 47.3352 ms.
---------------- Average Latencies ------------
Average latency of CDC per loop iteration is: 0.108401 ms.
Average latency of LZW per loop iteration is: 1.37821 ms.
Average latency of SHA256 per loop iteration is: 0.0369992 ms.
Average latency of DeDup per loop iteration is: 0.00303108 ms.
Average latency: 1.89341 ms.


--------------- Key Throughputs ---------------
Ethernet Latency: 2.88436s.
Bytes Received: 399054B.
Latency for Compression: 0.0546855s.
Latency for Compression (without fwrite): 0.0473352s.

Latency for CDC: 0.00271003s.
Latency for SHA: 0.0044769s.
Latency for LZW: 0.0344552s.
Latency for DEDUP: 0.00036676s.

CDC Throughput: 1178.01Mb/s.
SHA Throughput: 713.09Mb/s.
LZW Throughput: 92.6546Mb/s.
DEDUP Throughput: 8704.41Mb/s.

Ethernet Throughput: 1.10681Mb/s.
Application Throughput: 58.378Mb/s.
Application Throughput (without fwrite): 67.443Mb/s.
Bytes Contributed by Deduplication: 0B.
Bytes Contributed by LZW: 291731B.

root@u96v2-sbc-base-2020-2:/media/sd-mmcblk0p1#
```

Figure 2: Franklin Text File

Decode Diff Outputs for Linux.tar and Franklin.txt:

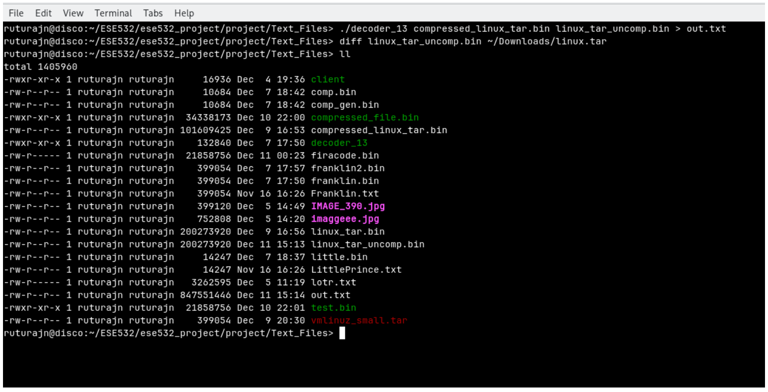


Figure 3: Linux Tar Decoder Diff

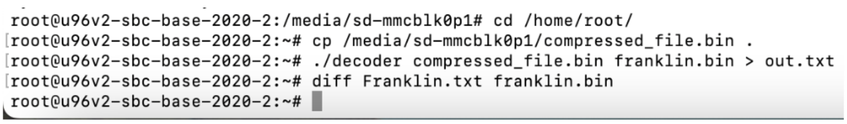


Figure 4: Franklin Decoder Diff

Vivado Analysis:

- Power Consumption for this Design: The estimated total on-chip power is **2.803 W**.

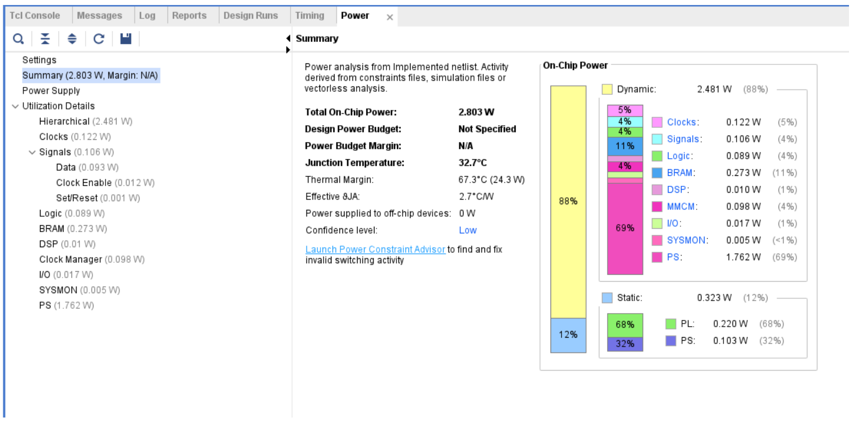
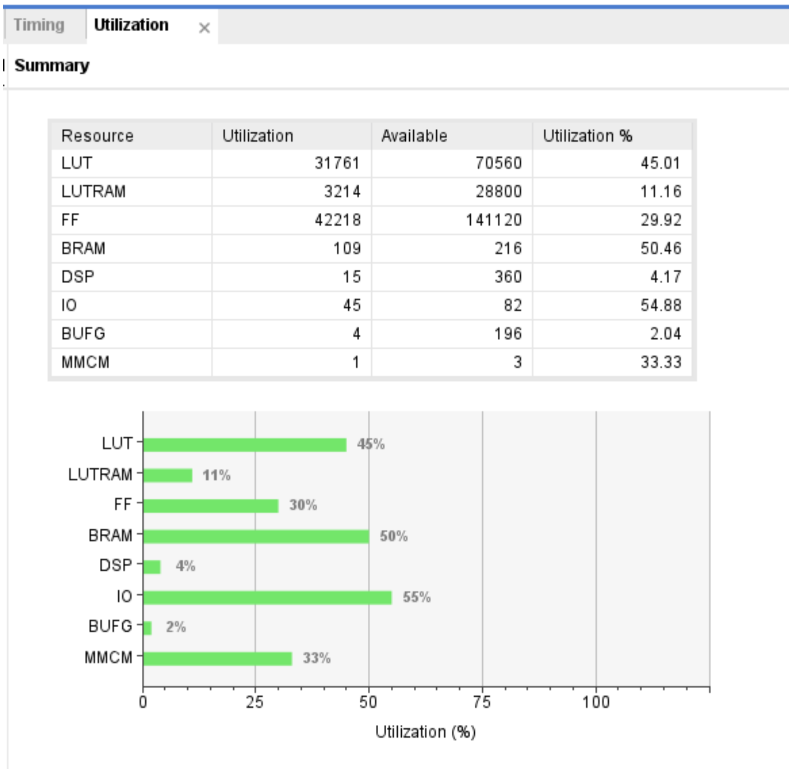


Figure 5: Power Consumption

- Resource Utilization:



Timing Utilization

Summary

Resource	Utilization	Available	Utilization %
LUT	31761	70560	45.01
LUTRAM	3214	28800	11.16
FF	42218	141120	29.92
BRAM	109	216	50.46
DSP	15	360	4.17
IO	45	82	54.88
BUFG	4	196	2.04
MMCM	1	3	33.33

Figure 6: Power Consumption

b) Task Decomposition: For our final implementation, we receive data in buffers of 16KB from the client. Since we have moved to a block size of 8192, we are receiving 16KB in 2 packets.

Once we receive all of the 16KB data from the client, this is sent to the CDC function which makes multiple chunks of 4KB. We are using a different version of CDC called the Fast CDC which ensures that none of the chunks are greater than 4KB. This implementation of Fast CDC uses the GEAR hash. Chunks could be smaller than 4KB but we perform a hard chunk at 4KB. This version of Fast CDC performs a modulus in a different way as opposed to the regular CDC functionality. This works by skipping sub-minimum chunk cut-points. In this implementation, if the current chunking position is less that the average chunk size the modulus operation with the hash is performed with a value higher than the average chunk size and if the current chunking position is greater than the average chunk size the modulus operation with the hash is performed with a value smaller than the average chunk size. This ensures an even distribution of chunk sizes. This is how, a chunk is declared. If not, it declares a hard chunk of 4KB by looking at the previous chunk boundary.

Once all these chunks are made, SHA generates a 256 bit digest for each of these chunks. We have mapped the SHA computation onto the NEON registers which provided a substantial speedup over the software SHA implementation. This was implemented using the MPSOC library that was given in the project handout. This implementation uses the SHA cryptographic intrinsics as well as the NEON intrinsics.

These chunk fingerprints are used as keys in the hash table in the Deduplication stage. Fingerprints associated with duplicate chunks are only added to the hash table once. As of our current implementation, we are performing LZW on each of these chunks irrespective of it being a duplicate or not. This is because branch statements in our kernel were proving to slow down the overall application. Hence, we chose to perform LZW compression on every chunk regardless of it being a duplicate or not. Then on the basis of the output of the Deduplication stage, we decide whether to write just the header or the compressed codes in the form of packets to the file.

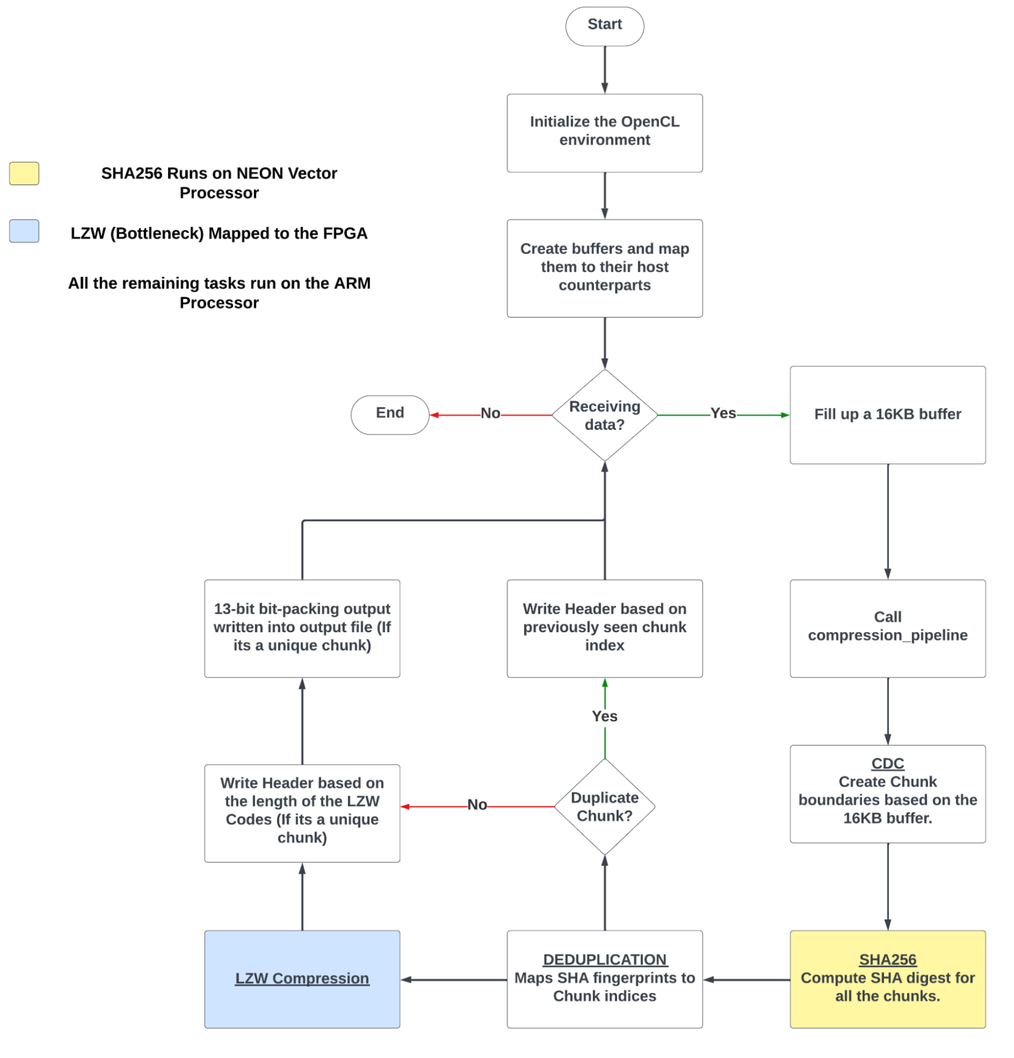


Figure 7: Flowchart

c) Compression Achieved:

For the test case - Franklin.txt file, we achieved a compression ratio of,

$$\frac{CompressedFileSize}{OriginalFileSize} = \frac{291731}{399054} = 0.73$$

For the test case - Linux.tar file, we achieved a compression ratio of,

$$\frac{CompressedFileSize}{OriginalFileSize} = \frac{95829565}{200273920} = 0.48$$

d) Parallelism:

1. For our CDC implementation, we tried to precalculate and store the power values in a local buffer. Since the hash calculation was using powers of 3 up to 17 repetitively for each iteration, we decided to hand calculate these and store them in an array from which they can be read for hash calculations. This made our CDC significantly quicker. To further optimize our CDC, we changed to FastCDC which used the GEAR hash and skips sub-minimum chunk cut-points. This made our final CDC implementation much faster as opposed to regular CDC.

   CDC throughput before changing to FastCDC: 377.822 Mb/s
   CDC throughput after adapting the FastCDC implementation : 1185.7 Mb/s
   Speedup: 3.138x

2. For the SHA implementation, we changed our software implementation of SHA 256 with the NEON intrinsics version of the implementation. This was implemented using the MPSOC library that was given in the project handout. This implementation uses the SHA cryptographic intrinsics as well as the NEON intrinsics which gives us significant speedup for the SHA since it uses 8 vector lanes.

   SHA latency with the software implementation: 0.00058189 s
   SHA latency with the NEON intrinsics implementation: 0.00028 s
   Speedup: 2.078x

3. Pipelining and parallelism can also be exploited within the kernel by enabling streams and data flow between the load, compute and store functions of the LZW compression. These streams act as a FIFO buffer from which data can be written to or read from each iteration. Functions within the dataflow region can execute concurrently.

   We tried to use the dataflow pragma to enable data flow between the load, compute and store function to speed up the kernel. Although, we weren't able to successfully complete it due to issues with synchronization in the output stream generated by the compute function. We noticed that with the dataflow pragma the C simulation worked perfectly fine, and the synthesis also passed, but the post C checking in co-simulation failed due to the output array being filled to zeros.

4. To exploit parallelism using threads in the host code, we tried to map the data collection from the ethernet on one thread and the compression_pipeline

function to another thread. This way we could use double buffering to fill data up in a buffer while another buffer is being processed on. Due to this our data processing pipeline is never waiting for data. Although this too didn't work out, and we were facing issues with the two buffers that were setup, which resulted in a segmentation fault at the end of the application.

e) Zynq Resource Mapping:

Application Stage	**Hardware**	**Memory Utilization**	**FPGA Resource Utilization**
Encoder	ARM Processor	145KB	0
CDC	ARM Processor	17KB	0
SHA	NEON Vector Processor + ARM Processor	10KB	0
DEDUP	ARM Processor	500 KB (Worst Case)	0
LZW	FPGA	Any memory used in the host is included in the Encoder	LUTs: 48550 FFs: 11269 DSPs: 3 BRAM_18Ks: 187

Table 5: Zynq Resource Mapping

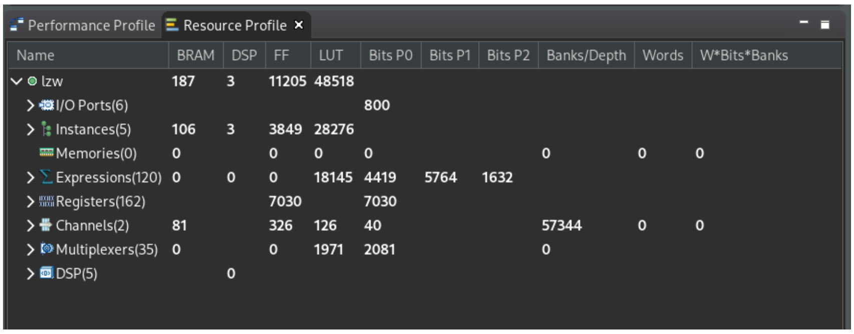
Utilization Estimates

⊟ Summary

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	18145	-
FIFO	81	-	326	126	-
Instance	106	3	3849	28276	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	1971	-
Register	-	-	7094	32	-
Total	187	3	11269	48550	0
Available	432	360	141120	70560	0
Utilization (%)	43	~0	7	68	0

Figure 8: Vitis Resource Utilization

The following screenshot shows the breakdown of the resource utilization on the FPGA for our LZW kernel.



Performance Profile | Resource Profile

Name	BRAM	DSP	FF	LUT	Bits P0	Bits P1	Bits P2	Banks/Depth	Words	W*Bits*Banks
lzw	187	3	11205	48518						
I/O Ports(6)					800					
Instances(5)	106	3	3849	28276						
Memories(0)	0		0	0	0			0	0	0
Expressions(120)	0	0	0	18145	4419	5764	1632			
Registers(162)			7030		7030					
Channels(2)	81		326	126	40			57344	0	0
Multiplexers(35)	0		0	1971	2081			0		
DSP(5)		0								

Figure 9: Vitis Resource Profile

f) Performance Model:
Our final implementation consists of only 1 kernel/ compute unit. Hence, that will be our LZW throughput. The throughput is going to be defined by the bottleneck or the slowest operation. Since, LZW is our bottleneck, it is going to define our overall performance/ throughput to a large extent. Since everything is currently running on 1 thread, our overall application throughput is going to be min (CDC, SHA, Deduplication , LZW Compression).
The overall application throughput does not match LZW's throughput because our current implementation does not overlap the computation of LZW with that of the other functions. As a result of which, the overall application throughput is a little lesser than the throughput of LZW.

Stage	Latency (ms)
CDC	2.71
SHA	4.47
DEDUP	0.36
LZW	34.45

Table 6: Latencies

Total Latency: 41.99 ms

Total Application Latency(including migrating memory back and forth from the host): 47.33 ms

$$ModeledThroughput = \frac{SizeOfFranklin}{TotalLatency} = \frac{\frac{399054 \times 8}{1000000}}{47.33} = 67.45Mb/s$$

The reason this does not match our actual throughput is because of memory transfer overheads.

g) Current Bottleneck Preventing Higher Performance:
The bottleneck in our final implementation continues to be the LZW compression. To improve its performance, we moved the LZW onto the FPGA and tried a variety of methods to improve it. The screenshot below shows the Application timeline of our kernel in Vitis Analyzer.

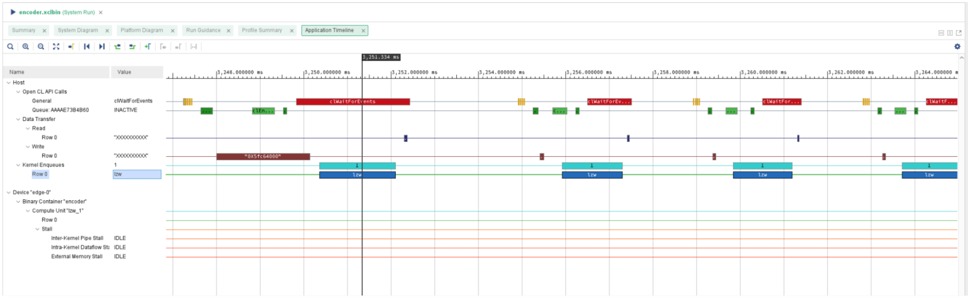


Figure 10: Timeline Trace

3 Validation and Real-Time Input

a) Design Validation:
At the beginning we focused on developing functionality for each of the different stages in the compression pipeline which included CDC, SHA, DEDUP and LZW. To enable seamless integration, during this stage we identified the way in which interfaces needed to be designed for each of the pieces to come together. This meant coming up with a high-level design of the complete control flow of our application, i.e., what data is passed to what stage and how it will process it before proceeding to the next.

Following is a detailed description of the design and validation that we performed for each of the aforementioned stages.

- **CDC:** Initially we started off by using our validated implementation from HW2 that used the rolling hash. This helped us make sure that we were working with a functional CDC and that it would not break the application when integrating it with the other components. As a part of our validation process we made sure that our CDC produced the same output as the python code in HW2 for varying modulus values as a part of further validation process. This was done by printing out chunk indices and redirecting them into a file, and the files created from our implementation and the python code were compared using the diff tool, to validate the functionality.

  We did this at every stage whenever the CDC was optimized as a sanity check and also to preemptively catch any bugs that would arise in the future. Apart from using text files we also used binary data to validate the functionality.

- **SHA:** From the beginning we focused on getting a fully functional SHA implementation since it was a critical part of the whole application. The output from SHA would decide if our chunk was a duplicate and whether it needed to be compressed, and if our SHA produced the same digest for different chunks it would severely harm the overall functionality.

  Our implementation for SHA-256 was based on the mpsoc-crypto library, which contained both the software and NEON versions of SHA-256. Initially aiming for functionality we went with implementing the software version. This library

had its own test test files that we first used to make sure that the SHA implementation was valid and legitimate. Once that was done we integrated that library into our implementation. Once that was done we tested it with our interface to validate that the functionality was preserved. The testing was done in two parts. Firstly we used the linux command line utility called sha256sum as our gold standard, and compared its output against it. Following that to further validate our design we used the NIST examples to test out the SHA implementation, provided here. We tested against the inputs and compared the generated digest with the digest provided in the reference.
We made sure that our design was validated both on text and binary data to ensure full functionality.

- **DEDUP:** Our implementation for DEDUP used unordered_maps to lookup SHA fingerprints and check if an entry for it already existed, if it did we would return the value, i.e., the chunk index mapped to that fingerprint or return -1 after inserting the digest into the table with a new chunk index.

  To validate this we made sure that previously seen SHA fingerp rints were producing the same chunk index, upon lookup into the map. This was done by testing the output from the same SHA fingerprint and validating that both the calls to dedup returned the same value. Apart from this we also tested our design by providing unique SHA fingerprints and checking the return value from the function (which should be -1, after the new mapping from SHA to chunk id is inserted into the map). Finally we tested if our design would produce the right chunk index after providing a new SHA fingerprint and then calling dedup again to check if the new fingerprint was inserted into the map with the right chunk index.

- **LZW:** For this part we started off with the LZW implementation from geeks for geeks and modified it so that it did not use any unsupported C++ constructs like strings and vectors. We made sure that all our parts could be easily ported to the FPGA if they were to turn out to be the bottleneck.

  Our interface for LZW required modifying the function to accept start and end indices into a big buffer, which represented the "chunk". Hence, we needed to make sure that the implementation worked for the chunk being somewhere in between the buffer, at the beginning of the buffer and at the end of the buffer. This was done by randomly generating start and end indices and passing them as arguments to the LZW function. This also helped us validate our loop bounds and ensured we were processing all the data that was passed into the function.

  Initially our validation process for this stage consisted of comparing the output from our implementation to the one produced by the geeks for geeks version. We started out by testing small strings like "I am Sam, Sam I am, am I Sam" and made sure that our output produced the same results as compared with the original. We then moved onto LittlePrince and finally moved onto small binary files. Once we were sure that our implementation was functioning correctly, we tested with larger text files like Franklin and the Lord of the Rings, and after this passed. After this we moved to testing large files like a zip archive for a font named FiraCode (20MB) and tar file for a graphical editor named neovide (24 MB).

Now for the application as a whole once we had integrated everything we started out with a simple packet emulation program, that would read blocksize amount of bytes from a file at a time and copy them into a 16KB buffer in a loop. Once this buffer was filled up we would call our compression_pipeline function to process that buffer. This helped us quickly weed out any problems that occurred during the integration phase since we were able to quickly test out our design on the PC itself instead of migrating all the binary files to the board and testing it there. It also aided in simplifying the original implementation so that we could focus on squashing any bugs in the standalone implementations of CDC, SHA, DEDUP and LZW.

After the packet emulation was fully functional we moved onto using the data packets transmitted via ethernet and running the whole application on the board (everything was mapped to a single ARM processor at this point). This helped us validate our design with all the intricacies involved in packet transfer. Apart from that this phase aided in identifying the bottleneck since we used stopwatch class to time and profile the different stages of our application. This data backed our decision of mapping LZW onto the FPGA.

Finally, when LZW was to be mapped to the FPGA we needed to focus on verifying its hardware implementation. This is when we wrote our testbench in Vitis that would compare the output from our kernel with the golden implementation from geeks for geeks. Initially, the testbench would read chunk_size data into a buffer and pass that to the LZW kernel for processing.

This worked for our initial implementations but when we optimized our host code to just call LZW once and process a batch of chunks we realized that to get a good idea of the functionality we had to expand the scope of testing, and emulate the conditions that the kernel is actually going to be running in, i.e., receiving start and end indices to chunks in a large buffer. So, we created an array that would hold randomly generated chunk indices and tested the kernel with them. Moving forward we decided that to emulate and generate realistic chunk indices we needed to use our CDC implementation and hence we included a function call to CDC in our testbench.

Finally, based on some bugs that we faced (which are outlined in the section below) we decided to enhance our testing routine. We would now read in the whole file 16KB at a time in a loop, pass that buffer through CDC, generate chunk indices, and then pass that data to our kernel for testing. Following which the output from the kernel would be compared against the golden implementation at the granularity of a chunk. The packet length and the actual data was compared against the golden implementation to make sure that our kernel was not producing any incorrect values. This helped us understand the level of current functionality and the points at which our implementation would break, and in this way we could fix our design before deploying it onto the FPGA.

b) Real-Time Guarantee:
Based on our extensive testing and validation we can say with confidence that our design is completely functional and stable with a throughput of approximately 70 Mb/s as per the "linux.tar" test for files up to 200 MB with an -s parameter of 2000 at a block size of 8192B. The -s parameter is only required in case of large files like the "linux.tar" and the "FiraCode" archive (which required an -s parameter of 1000). Whereas for smaller files like "Franklin.txt" and "LittlePrince.txt" we noticed that the -s parameter was not required. We observed this trend of a smaller or no -s parameter as the size of the file reduced. As per our reasoning this may be due to the fact that our CDC creates chunks based on the input data, and the number of chunks may vary depending on the content. As the number of chunks increase the processing time for each 16KB data buffer that we maintain increases, and this may change dynamically based on the type of the input, and since the probability of having this worst case delay goes up with a higher number of packets (for large files), the -s parameter also needs to go up.

Commands to run our design:

- **On a Mac (Host):** Run the following command on your board (by default the file name for the compressed file is *compressed_file.bin*,

```
$ ./encoder -k encoder.xclbin -f <Filename>
```

  Run the following command in your host computer,

```
$ ./client_mac -i <board_ip_address> -b 8192 -f <Filename> -s 1000
```

  Now to decode the file run,

```
$ ./decoder <compressedFilename> <outputFilename>
```

- **On a Windows (Host):** Run the following command on your board (by default the file name for the compressed file is *compressed_file.bin*,

```
$ ./encoder -k encoder.xclbin -f <Filename>
```

  Run the following command in your host computer,

```
$ ./client -i <board_ip_address> -b 1024 -f <Filename> -s 1000
```

  Now to decode the file run,

```
$ ./decoder <compressedFilename> <outputFilename>
```

c) Challenges and Debugging:
During the development of this project, we faced a lot of challenges and bugs which were uncovered during the thorough testing that we carried out for each of the different stages. Following is a detailed description of all the bugs that we fixed and

the challenges that we faced broken down based on the different stages of the data pipeline.

- **CDC:** When we were integrating our CDC into the whole application, we noticed that CDC was creating chunk boundaries at every character. After carefully analyzing the code and stepping through the code in GDB we noticed that the offset that we used for filling up the 16KB buffer was invalid and this resulted in CDC operating on garbage data. Once this was identified we fixed our offset calculation to use the length from the header of the ethernet packet and this rectified our issue.

- **SHA/DEDUP:** In this stage we faced more of a challenge than a bug, which was related to storing the SHA fingerprints as keys into an unordered_map that was used to map SHA digests to chunk indices. Now, the unordered_map in C++ can't efficiently map keys which are represented as containers like vectors, arrays, etc. This meant that we couldn't store our SHA digest as a byte array but had to convert it into a form that could be represented by a single data type, which was a string. This conversion of the byte array to a string proved to be really challenging since we had to take special care while converting "0"s presents in the SHA fingerprint since NULL (0x00) characters are considered to be string terminators in C. To mitigate this problem we used the stringstream class in C++. Our solution included creating a hexadecimal stringstream and iterating over the byte array. At each iteration we would cast the current byte into an int using the static_cast directive, and append the int value as a two character wide hexadecimal string to the output string. Finally after every byte was processed we would convert the stringstream object into a string and return that from SHA so that DEDUP could then utilize an unordered_map.

  Apart from that in the SHA library we noticed that the implementation used strlen to find the length of the string and compute the SHA based on that, but we knew that this wouldn't work on binary data since as described earlier NULL (0x00) characters are considered to be string terminators in C, and this would result in the string getting cut out prematurely which in-turn would give out an incorrect SHA. To pre-emptively catch this we adapted the library to use start and end indices into a buffer representing the chunk to ensure that every chunk is processing completely.

- **LZW:** The first bug that we faced in LZW was related to the last character not showing up properly in the decoded output for a chunk. This happened in all the files that we tested, and it pointed us to our incorrect handling of the loop bounds for the last character. When we analyzed the code using gdb we found out that the loop terminated prematurely because our loop bound was set by subtracting 1 from the end index whereas the input passed to LZW already took that into account and this led to an off by one error. Once it was identified we fixed it by removing the subtraction by 1.

  After we had moved to the FPGA mapping of the LZW we noticed that the implementation was failing when working with binary data. This was due to the fact that the first 256 entries in the hash table were initialized with the key being the ASCII values of all the characters followed by a zero to the respective ASCII values of the character itself, i.e., 'a0' was mapped to 97 since the ASCII value of 'a' is 97. This meant that a sequence of characters represented by '00' was already mapped to a value of 0, although actually we haven't seen that mapping before and hence the mapping should really be of an invalid character followed by a valid character to ensure that we aren't mapping any sequences accidentally. To ensure this we can just remove the initialization of the hash table and keep the next code set to 256. This way we ensure that there are no invalid mapping present in the hash table which will prevent from getting any false positive lookups.

Another problem that we faced while processing binary data with LZW was that our implementation would fail at random places within the binary file and work perfectly fine most of the time. It took us a lot of time trying to understand where the problem was occurring, and since we were using Vitis there wasn't a good debugging tool available. So, firstly we included some debugging print statements in the code to find the point of failure (this included the iteration index, offset, etc.). After that we decided to compile the testbench and our kernel using g++ outside vitis and used two gdb instances, to stop at the exact iteration and offset in both our kernel and the golden implementation parallelly. There we first examined the prefix and the next character to ensure that the problem didn't occur in a previous iteration. We found that both of them were the same, then we advanced the code in the golden implementation to check if the sequence was present in the map and was found or was it inserted into the map this iteration, and surprisingly our implementation was also following the same pattern. This suggested that there was a problem with the value returned by the lookup. We checked the current code value and found that the value was exceeding our set maximum of 4096, and due to this our hash lookup returned a value that wasn't associated with that particular key, since our hash table was configured to have a key consisting of a 12-bit value and an 8-bit character. Due to this our complete implementation broke down and we were getting random points of failure.

Once we found this out, we decided that we will need to move to 13-bit hash table configuration and 13-bit packed output since our lzw codes were exceeding the value represented by 12-bits. Hence, we modified the code accordingly. After that we encountered a similar problem. We were still getting random failures. After setting up the parallel gdb instances again we found that the problem was with overflow, the insert function in the associative memory didn't set the proper bit in memory when the fill value exceeded 32, since 1 << 32 exceeded the int maximum value. So, after identifying that issue, we modified the code to use 1UL instead of 1 to increase the length of the data type and prevent overflow, and the bug was fixed.

4 Key Lessons

a) Design and Optimization:

- Gaining insight into the functionalities of various sub-stages in the system design and algorithmic aspects of CDC for processing Ethernet input data. This includes the incorporation of SHA-256 for deduplication, the utilization of LZW compression, and implementing bit-packing techniques.

- Systematically improving function performance by addressing hashing function requirements, selecting data types compatible with both text and binary inputs, and eliminating unnecessary memory copy functions.

- Applying vectorization techniques with the NEON accelerator to enhance bandwidth, informed by lessons learned from Homework 4. Identifying application bottlenecks through initial calculations and a simplified software version.

- Leveraging Vitis Analyzer for debugging, port assignment, and kernel execution tracing. Utilizing VITIS pragmas to optimize FPGA functions in terms of initiation interval (II) and resource utilization.

- Throughout this submission, we found Git to be an invaluable tool, enabling us to maintain tagged code versions. This proved essential for identifying major commits that significantly contributed to functionality or signaled upcoming substantial code revisions.

b) Debugging:

- We found GDB to be quite useful while debugging the code. It enabled us to use breakpoints, to jump to specific areas in the code and examine everything carefully to pinpoint the issue. It also helped us achieve full functionality of the kernel when working with Vitis, since we used it outside Vitis by compiling our kernel and the testbench with g++.

- The packet emulation using file read was also really helpful when testing out the functionality of the complete application.

- Including print statements helped examine the state of the output from the kernel since we couldn't access the host buffer mapped to the kernel using OpenCL in GDB.

- Vitis Analyzer proved to be helpful when we were facing a synchronization problem in the host code, and looking at the timeline trace helped us understand the issue and rectify it.

- The data flow diagram and the scheduler view in Vitis helped us visualize the control flow and the operations scheduled at various cycles enabling us to pinpoint the source code that caused a low II and fix it.

c) Teamwork and Collaboration:

- We gained valuable insights into effective collaboration by using version control systems like Git and GitHub, where we honed our skills in branching, merging, and resolving conflicts seamlessly within the team.

- Task distribution among team members for design, optimization, and debugging was executed with efficiency, ensuring a well-coordinated effort.

- Our regular team meetings became a platform for constructive feedback, focusing on aspects like code quality, efficiency, and addressing any issues promptly.

- Emphasizing the significance of investing time in thorough testing and debugging to prevent integration issues during the final stages of the project.

- Communication played a pivotal role; engaging in discussions about individual tasks, timelines, and challenges faced by team members led to more efficient problem-solving.

- Navigating changes in project requirements was approached with flexibility, adapting to unforeseen challenges and proactively adjusting the project plan accordingly.

- Recognition and celebration of team achievements and milestones became integral, contributing to the fostering of a positive team culture and maintaining high motivation levels.

5 Design Space Exploration, Graphs and Models

5.1 CDC

Before moving to FastCDC:
The main design space that we explored was the chunk size that gets set by the modulus value. Initially this value was set to 1024 but we had to eventually increase this to 4096. A smaller chunk size would imply more deduplication but the compression and overall throughput increases with larger chunks. As mentioned in the project handout, we wanted to chunk with at least a size of 4096 bytes. Another thing that drastically helped improve our CDC throughput was pre calculating powers of 3 (which was our chosen PRIME) and storing them in a local buffer. Since the hash calculation was using powers of 3 up to 17 (this was based on our WIN_SIZE used for the hash) repetitively for each iteration, we decided to pre-calculate these and store them in an array from which they can be read for hash calculations. This made our CDC significantly quicker.

After moving to FastCDC:
To further optimize our CDC, we changed to FastCDC which used the GEAR hash and skipped sub-minimum chunk cut-points. Fast CDC theorizes that some of the hash calculations can be skipped when defining chunk boundaries in favor of a low probability that a chunk will occur at a position less than the desired average chunk size. To explain this let's consider that we want chunks of size 4096, now the FastCDC says that the probability of drawing a chunk boundary at say 1024 (defined as the minimum chunk size) is very low, and hence we can skip hash calculation until the first 1024 bytes in a chunk. This drastically reduces the latency of CDC while efficiently drawing good chunk boundaries.

Now the GEAR hash consists of essentially a lookup table of 256 values and the hash value for a certain character is the value stored at the index represented by the ASCII value of that character. This means that the hash calculation only requires an array lookup. This along with the skipping of hash calculations, made our final CDC implementation much faster as opposed to regular CDC.

Data: Test Case - Franklin Text File (File Size: 399054 B)

Chunk Size	Total Throughput (Mb/s)	CDC Throughput (Mb/s)	SHA Throughput (Mb/s)	LZW Throughput (Mb/s)	DEDUP Throughput (Mb/s)
2048	58.4843	1781.58	551.707	79.0904	5561.34
4096	67.443	1178.01	713.09	92.6546	8704.41
8192	72.8656	1000.7	851.229	100.217	13057

Table 7: Chunk Size Versus Throughput - Franklin

Chunk Size	Contributed DEDUP (B)	Contributed LZW (B)	Compression Ratio
2048	0	327310	0.82
4096	0	291731	0.73
8192	0	259191	0.65

Table 8: Chunk Size Versus Compression Ratio - Franklin

The graph below was generated by using the data points that we have and extrapolating them using the power rule, i.e., $y = Ax^B$. This enabled us to estimate the values of throughput and compression ratios for values of chunk size ranging from 256 to 65536, and then plot them.

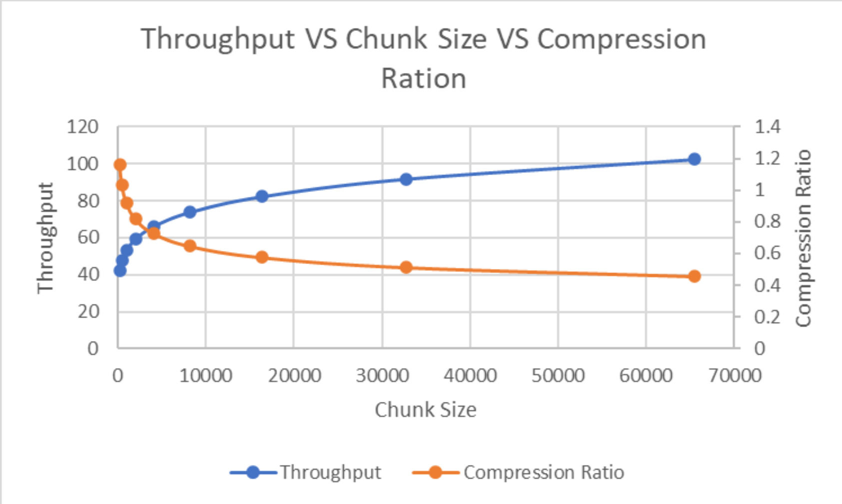


Figure 11: Chunk Size VS Throughput VS Compression Ratio Graph

Data: Test Case - vmlinuz_small.tar (File Size: 399054 B)

Chunk Size	Total Throughput (Mb/s)	CDC Throughput (Mb/s)	SHA Throughput (Mb/s)	LZW Throughput (Mb/s)	DEDUP Throughput (Mb/s)
2048	73.3759	1639.94	593.419	89.3484	10673.8
4096	87.4704	1099.73	776.749	109.426	16190.4
8192	94.3199	965.239	911.699	119.165	20780

Table 9: Chunk Size Versus Throughput - vmlinuz_small

Chunk Size	Contributed DEDUP (B)	Contributed LZW (B)	Compression Ratio
2048	768	643	0.004
4096	380	734	0.003
8192	184	938	0.003

Table 10: Chunk Size Versus Compression Ratio - vmlinuz_small

5.2 SHA 256

Initially we were using a software implementation from the mpsoc-crypto library provided in the project handout. To optimize this further, we moved the SHA implementation onto the ARM NEON cores since this was an effective method of exploiting parallelism.

For our final implementation, we adapted the NEON intrinsics version of the implementation. The mpsoc -crypto library provided in the project handout was used for this. This uses the SHA cryptographic intrinsics as well as the NEON intrinsics which gives us significant speedup for the SHA since it uses 8 vector lanes.

5.3 LZW Kernel

We tried to exploit various design axes while trying to optimize the LZW kernel. We started by removing the for loop from the associative memory lookup (assoc_lookup) function and replaced these with bit shifting and manipulation operations. Since, 'for' loops are computation intensive, this change helped us decrease the latency significantly. Additionally, the main LZW loop initially consisted of a while loop. We changed this to a for loop since they are much faster and to help Vitis HLS know that the number of iterations are already known. We pipelined these loops to help decrease the II and make it more efficient. We also implemented load-store-compute which enabled us to stream data across these functions in the kernel.

We were also getting many hash collisions with the original hash function. To counter this, we also adapted the murmur hash in our LZW kernel implementation to reduce the hash collisions. This also allowed us to decrease the size of our hash table which resulted in reduced lookup times. As mentioned in the previous milestones, we also made use of 2 buckets instead of the one since we were getting multiple hash collisions for bigger binary files. This meant that each index was associated with 2 key-value pairs instead of just one. As a result of which, the associative memory would only come into use when 2 keys have already been previously hashed to the same index and entered into the main hash table. This decreased the entries in our associative memory significantly.

To reduce the number of calls to the kernel and minimize the data transfer overhead we batched all the chunks in a 16KB buffer and sent that as a whole to the kernel in the kernel it would iterate over all the chunks and compute LZW. This gave us a huge speedup in comparison to calling the kernel in a loop based on the output from DEDUP. Once the data was received back from the kernel we would use the output from DEDUP to decide what to write into the compressed file.

Apart from that to minimize the kernel overhead we reduced the amount of input arguments required. Firstly, the chunk_indices array for the batched chunks that needed to be sent to LZW consisted of its first element being set to the length of the array; this way we avoided sending another parameter to the kernel and decreasing the setup time. Similarly for the out_packet_lengths argument of the kernel which populated the length of the encoded LZW chunks had its first element signify if insertion into the associative memory had failed. This helped us further reduce the data transfer time. As described previously in section III under "Challenges and Debugging", we noticed that with binary files the code value in LZW exceeded 4096, which was the configuration for our hash table lookup and bit packing. Due to this we decided to move to a 13-bit packing routine and modified the hash table to support 13-bit values, and updated the size of the key accordingly. This helped us mitigate the issue of the code exceeding 4096 and ensured that our design would be stable and functional for our configured chunk size.

Another optimization that we tried was moving the bit packing inside the kernel and producing the bit-packed output for the batched chunks. We thought that this would give us a significant speedup, instead it slowed down our kernel and the overall performance of the application took a hit. We tried to optimize the bit-packing in Vitis by pipelining and unrolling wherever possible, but it didn't help. Due to this we decided to move the bit-packing routine outside the kernel and back onto the ARM processor.

Finally, another optimization that we tried was to reduce the number of parameters sent to our compression_pipeline function, since the OpenCL buffers were set up in main and then they were passed to the compression_pipeline function along with their host mapped buffers. Apart from this the function took in a lot of other parameters like the OpenCL command queue, the kernel, the file pointer to write into the file, etc. Hence to minimize the amount of arguments, we tried to wrap the OpenCL buffers and their host counterparts into a struct and tried to pass a pointer to that struct, but when we did that we weren't able to properly retrieve data back into our buffer from the kernel and decided to get rid of it, in interest of complete functionality.

6 Future Scope

In an attempt to optimize this further, we thought of implementing thread pooling where different stages could be run on different threads. By using multiple threads, we could explore pipelining LZW with other parts of the pipeline such as data collection, CDC, SHA and Deduplication. This would enable us to overlap the computation of LZW with that of other stages.

Additionally, we could try to run multiple compute units concurrently for different chunks since each chunk is independent of one another. We tried implementing this for our final design, however, since we were using almost 65% of the available LUTs, we could not replicate this design to run on multiple compute units.

As of now when the chunks are batched in the kernel, we are sequentially iterating over all the chunks and computing LZW on them individually at a time. Given a better (lower) resource utilization we could unroll the for loop that processes these chunks, and potentially use the dataflow pragma to simulate multiple compute units, i.e., running multiple instances of LZW concurrently.

Apart from that another optimization that we could potentially make is to parallelize the lookup between the buckets in the hash table and the associative memory which would significantly speed up the LZW process and improve the performance overall. Furthermore, we could also use better hash functions and techniques like double hashing to reduce the size of our hash table and improve our resource utilization. This in-turn ties up to the problem of not being able to completely unroll the loop, and effectively leverage the resources present on the FPGA.

7 Task Distribution

We allocated tasks among ourselves to design and optimize the project. Akhil Gunda focused on enhancing the CDC implementation, optimizing with Fast CDC implementation and the host code, Siddhant Mathur concentrated on adapting SHA to use NEON intrinsics and seamlessly integrating it into the entire system along with the initial version of 12-bit packing, while Ruturaj Nanoti tackled the testbench, optimized portions of the LZW kernel, had to come up with optimized 13 bit packing to support bigger binaries files. Additionally, we collaborated on debugging the final stage performance and issue and on further optimizing the LZW collectively. Our combined efforts ensured a well-rounded approach to achieving the project's complete functionality.

We, **Ruturaj A. Nanoti**, **Akhil Gunda**, and **Siddhant Mathur**, certify that we have complied with the University of Pennsylvania's Code of Academic Integrity in completing this final exercise.

8 Appendix

8.1 Vivado Results

- Control interface of the accelerator wrapper lzw_1 screenshot:

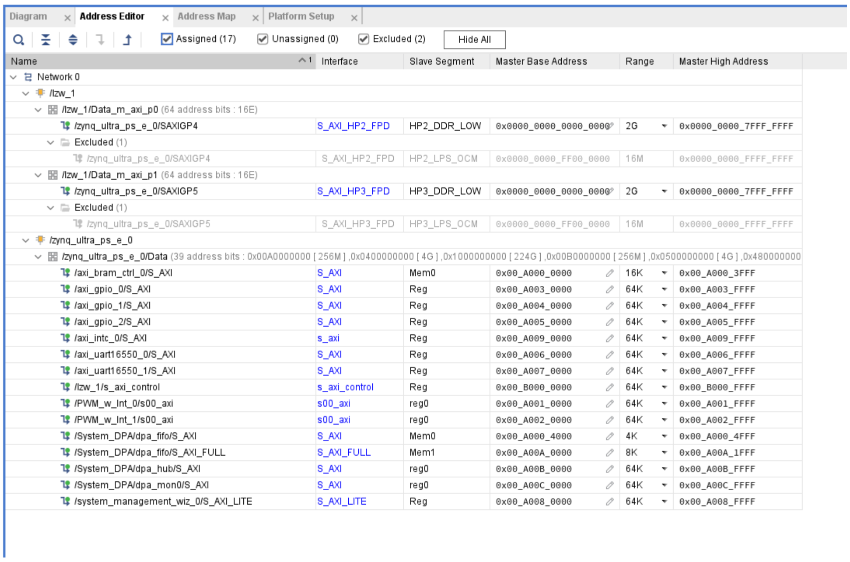


Figure 12: Vivado Ports

For lzw_1 the s_axi_control is mapped to 0x00_B000_0000 to 0x00_B000_FFFF.

- Vivado Block design under IP integrator:

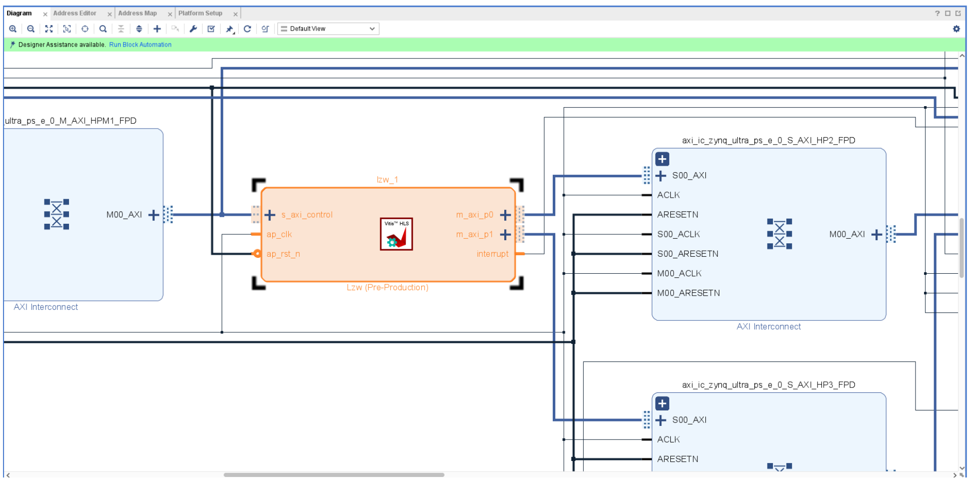


Figure 13: Vivado Design Block Diagram

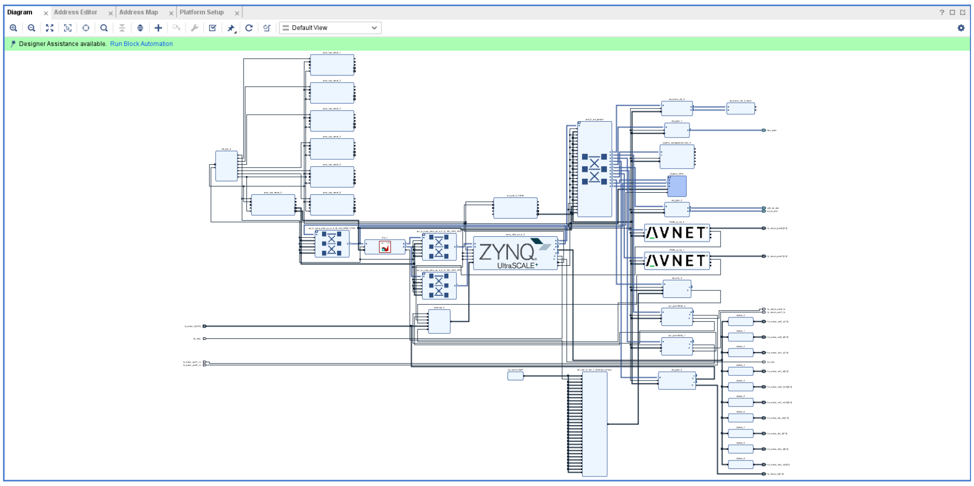


Figure 14: Vivado Design Overall Block Diagram

- Device (Design Mapping):

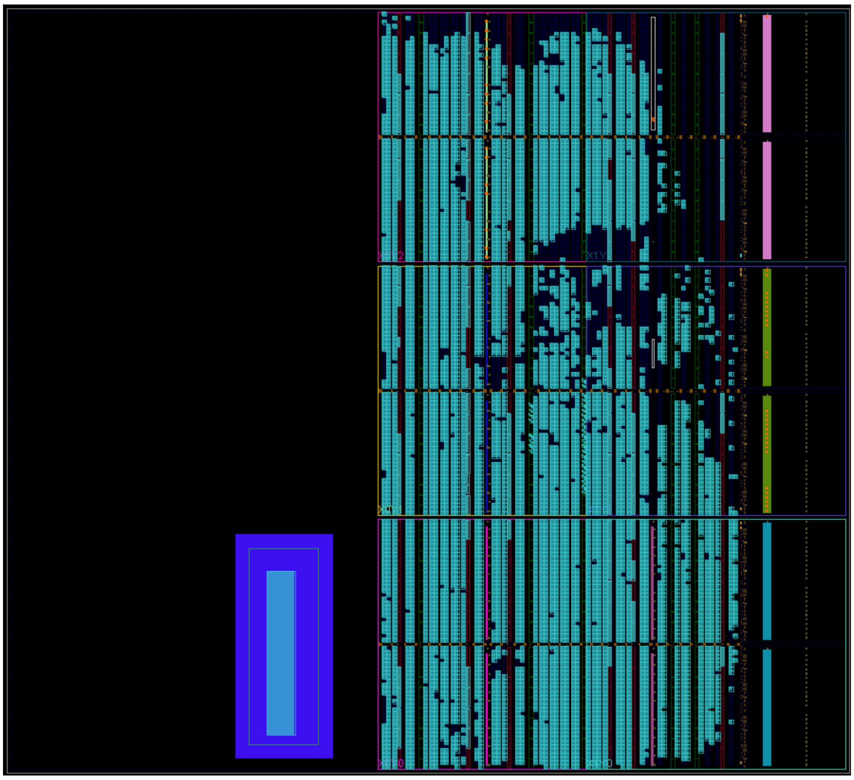


Figure 15: Vivado Design Mapping

8.2 Vitis HLS and Vitis Analyzer Results

- Co-Simulation Screenshot:

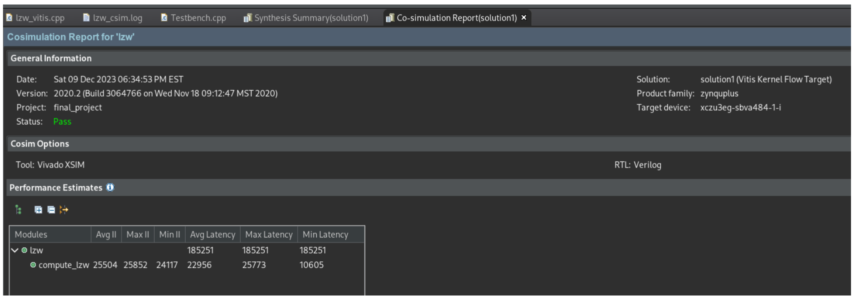


Figure 16: Vitis Co-Sim Result

- System Diagram:

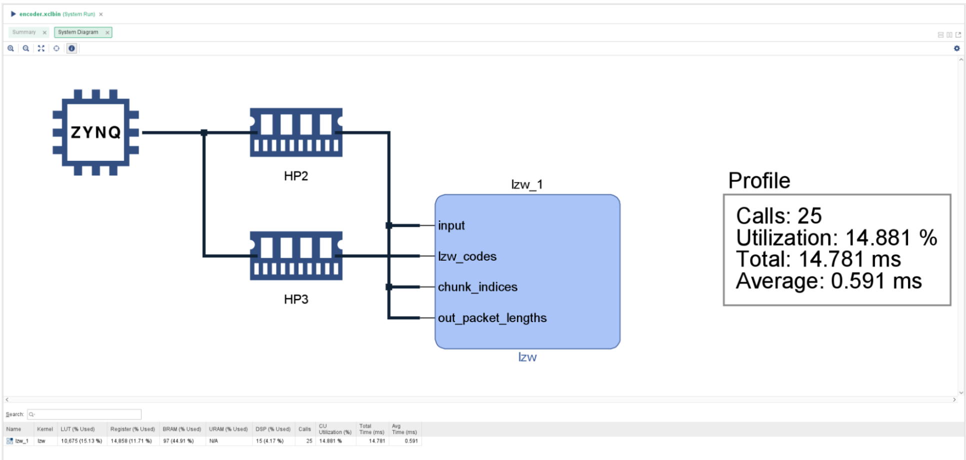


Figure 17: Vitis System Diagram

- Profile Summary:

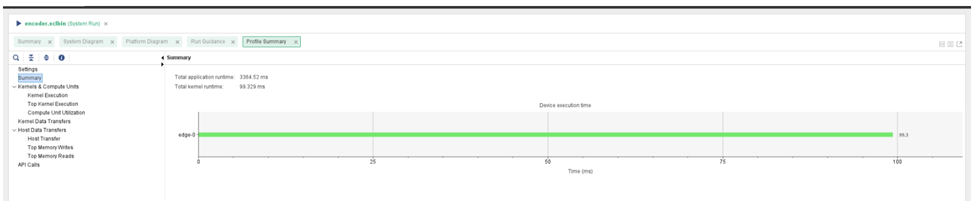


Figure 18: Vitis Profile Summary

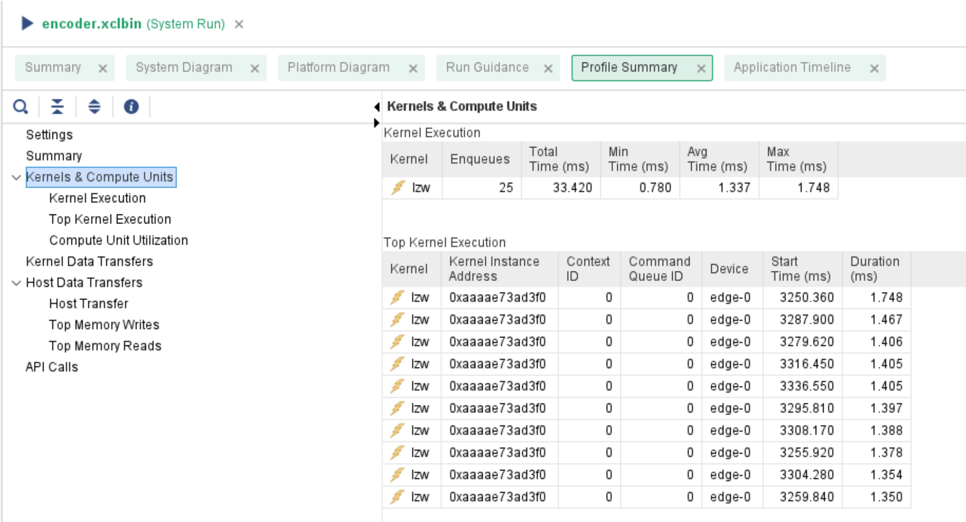


Figure 19: Kernel and Compute Units

8.3 Code

8.3.1 CDC

```
#include <iostream>
#include <math.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <vector>

using namespace std;

#define CHUNK_SIZE 4096
#define MODULUS_MASK (CHUNK_SIZE - 1)
#define MODULUS_MASK_S ((CHUNK_SIZE * 2) - 1)
#define MODULUS_MASK_L ((CHUNK_SIZE / 2) - 1)
#define TARGET 0
#define BITS 12
#define MINSIZE 1024
#define BUFFER_LEN 16384

uint64_t GEAR[256] = {
    1553318008, 574654857,  759734804,  310648967,  1393527547,
   1195718329,
    694400241,  1154184075, 1319583805, 1298164590, 122602963,
   989043992,
    1918895050, 933636724,  1369634190, 1963341198, 1565176104,
   1296753019,
    1105746212, 1191982839, 1195494369, 29065008,   1635524067,
   722221599,
    1355059059, 564669751,  1620421856, 1100048288, 1018120624,
   1087284781,
    1723604070, 1415454125, 737834957,  1854265892, 1605418437,
   1697446953,
    973791659,  674750707,  1669838606, 320299026,  1130545851,
   1725494449,
```

```
 939321396,  748475270,  554975894,  1651665064, 1695413559,
671470969,
 992078781,  1935142196, 1062778243, 1901125066, 1935811166,
1644847216,
 744420649,  2068980838, 1988851904, 1263854878, 1979320293,
111370182,
 817303588,  478553825,  694867320,  685227566,  345022554,
2095989693,
 1770739427, 165413158,  1322704750, 46251975,   710520147,
700507188,
 2104251000, 1350123687, 1593227923, 1756802846, 1179873910,
1629210470,
 358373501,  807118919,  751426983,  172199468,  174707988,
1951167187,
 1328704411, 2129871494, 1242495143, 1793093310, 1721521010,
306195915,
 1609230749, 1992815783, 1790818204, 234528824,  551692332,
1930351755,
 110996527,  378457918,  638641695,  743517326,  368806918,
1583529078,
 1767199029, 182158924,  1114175764, 882553770,  552467890,
1366456705,
 934589400,  1574008098, 1798094820, 1548210079, 821697741,
601807702,
 332526858,  1693310695, 136360183,  1189114632, 506273277,
397438002,
 620771032,  676183860,  1747529440, 909035644,  142389739,
1991534368,
 272707803,  1905681287, 1210958911, 596176677,  1380009185,
1153270606,
 1150188963, 1067903737, 1020928348, 978324723,  962376754,
1368724127,
 1133797255, 1367747748, 1458212849, 537933020,  1295159285,
2104731913,
 1647629177, 1691336604, 922114202,  170715530,  1608833393,
62657989,
 1140989235, 381784875,  928003604,  449509021,  1057208185,
1239816707,
 525522922,  476962140,  102897870,  132620570,  419788154,
2095057491,
 1240747817, 1271689397, 973007445,  1380110056, 1021668229,
12064370,
 1186917580, 1017163094, 597085928,  2018803520, 1795688603,
1722115921,
 2015264326, 506263638,  1002517905, 1229603330, 1376031959,
763839898,
 1970623926, 1109937345, 524780807,  1976131071, 905940439,
1313298413,
 772929676,  1578848328, 1108240025, 577439381,  1293318580,
1512203375,
 371003697,  308046041,  320070446,  1252546340, 568098497,
1341794814,
 1922466690, 480833267,  1060838440, 969079660,  1836468543,
2049091118,
 2023431210, 383830867,  2112679659, 231203270,  1551220541,
1377927987,
 275637462,  2110145570, 1700335604, 738389040,  1688841319,
1506456297,
```

```
    1243730675, 258043479,  599084776,  41093802,   792486733,
    1897397356,
    28077829,   1520357900, 361516586,  1119263216, 209458355,
    45979201,
    363681532,  477245280,  2107748241, 601938891,  244572459,
    1689418013,
    1141711990, 1485744349, 1181066840, 1950794776, 410494836,
    1445347454,
    2137242950, 852679640,  1014566730, 1999335993, 1871390758,
    1736439305,
    231222289,  603972436,  783045542,  370384393,  184356284,
    709706295,
    1453549767, 591603172,  768512391,  854125182,
};

void fast_cdc(unsigned char *buff, unsigned int buff_size,
              unsigned int chunk_size, vector<uint32_t> &vect) {
    unsigned int hash = 0;
    unsigned int i = MINSIZE;
    unsigned int prev = 0;

    vect.push_back(0);
    unsigned int modulus_mask_s = (chunk_size * 2) - 1;

    while (i < buff_size) {
        hash = (hash >> 1) + GEAR[buff[i]];
        if (((hash & modulus_mask_s) == TARGET) ||
            (((i - prev)) == chunk_size)) {
            vect.push_back(i);
            prev = i;
            i += MINSIZE;
            hash = 0;
        }
        i += 1;
    }

    if (vect[vect.size() - 1] != buff_size)
        vect.push_back(buff_size);
}
```

8.3.2 SHA

```
#include "common.h"
#include "sha256/sha256.h"
#include <iomanip>
#include <iostream>
#include <sstream>
#include <stdio.h>
#include <string.h>
#include <vector>

using namespace std;

string bytearray_hex_string(unsigned char *bytes, int size) {
    stringstream ss;
    ss << hex << setfill('0');
    for (int i = 0; i < size; i++) {
```

```
        ss << setw(2) << static_cast<int>(bytes[i]);
    }
    return ss.str();
}

string sha_256(unsigned char *chunked_data, uint32_t chunk_start_idx,
               uint32_t chunk_end_idx) {
    SHA256_CTX ctx;
    BYTE hash_val[32];

    sha256_hash(&ctx, (const BYTE *)(chunked_data + chunk_start_idx),
                chunk_start_idx, chunk_end_idx, hash_val, 1);

    string sha_fingerprint = bytearray_hex_string(hash_val, 32);

    return sha_fingerprint;
}
```

8.3.3 DEDUP

```
#include "common.h"
#include <cstdlib>
#include <iostream>
#include <stdbool.h>
#include <unordered_map>

using namespace std;

// Return -1 on failure.
int64_t dedup(string sha_fingerprint) {

    static unordered_map<string, int64_t> sha_chunk_id_map;
    static int64_t chunk_id = 0;

    bool found =
        (sha_chunk_id_map.find(sha_fingerprint) == sha_chunk_id_map.end()
             ? false
             : true);

    // Perform lookup in map here.
    if (!found) {
        // Insert into map before calling LZW.
        sha_chunk_id_map[sha_fingerprint] = chunk_id;
        ++chunk_id;
        return -1;
    } else
        return sha_chunk_id_map[sha_fingerprint];
}
```

8.3.4 LZW

```
#include <hls_stream.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>

#define CAPACITY                                                              \
    16384 // hash output is 15 bits, and we have 1 entry per bucket, so capacity
          // is 2^15
#define SEED 524057
#define ASSOCIATIVE_MEM_STORE 64
#define CHUNK_SIZE 4096
#define MAX_CHUNK_SIZE 8192
#define INFO_LEN 4
#define BUFFER_LEN 16384
#define MAX_ITERATIONS 20
#define MAX_OUTPUT_CODE_SIZE 40960

#define SEED_MURMUR 524057
#define SEED_CRAP 75768593

#define FNV_PRIME 16777619
#define FNV_OFFSET_BASIS 2166136261U

typedef enum InfoParams {
    INFO_START_IDX,
    INFO_END_IDX,
    INFO_OUT_PACKET_LENGTH,
    INFO_FAILURE,
} InfoParams;

//******************************************************************
typedef struct {
    // Each key_mem has a 9 bit address (so capacity = 2^9 = 512)
    // and the key is 20 bits, so we need to use 3 key_mems to cover all the key
    // bits. The output width of each of these memories is 64 bits, so we can
    // only store 64 key value pairs in our associative memory map.

    unsigned long upper_key_mem[512]; // the output of these  will be 64 bits
                                      // wide (size of unsigned long).
    unsigned long middle_key_mem[512];
    unsigned long lower_key_mem[512];
    unsigned long
        value[ASSOCIATIVE_MEM_STORE]; // value store is 64 deep, because the
                                      // lookup mems are 64 bits wide
    unsigned int fill; // tells us how many entries we've currently stored
} assoc_mem;

// cast to struct and use ap types to pull out various feilds.
//******************************************************************

static inline uint32_t murmur_32_scramble(uint32_t k) {
    k *= 0xcc9e2d51;
    k = (k << 15) | (k >> 17);
    k *= 0x1b873593;
```

```
    return k;
}

unsigned int inline murmur_hash(unsigned long key) {
    uint32_t h = SEED;
    uint32_t k = key;

    h ^= murmur_32_scramble(k);
    h = (h << 13) | (h >> 19);
    h = h * 5 + 0xe6546b64;

    h ^= murmur_32_scramble(k);
    /* Finalize. */
    h ^= h >> 16;
    h *= 0x85ebca6b;
    h ^= h >> 13;
    h *= 0xc2b2ae35;
    h ^= h >> 16;
    return h;
}

uint32_t inline Crap8(unsigned int key) {
#define c8fold(a, b, y, z)                              \
    {                                                   \
        p = (uint32_t)(a) * (uint64_t)(b);              \
        y ^= (uint32_t)p;                               \
        z ^= (uint32_t)(p >> 32);                       \
    }
#define c8mix(in)                                       \
    {                                                   \
        h *= m;                                         \
        c8fold(in, m, k, h);                            \
    }

    const uint32_t m = 0x83d2e73b, n = 0x97e1cc59;
    const uint32_t key4[4] = {((key)&0xFF), ((key >> 8) & 0xFF),
                              ((key >> 16) & 0xFF), ((key >> 24) & 0xFF)
    };
    uint32_t h = SEED_CRAP, k = n;
    uint64_t p;

    c8mix(key4[0]) c8mix(key4[1]) c8mix(key4[2])
        c8mix(key4[3] & ((1 << (2 * 8)) - 1)) c8fold(h ^ k, n, k, k)
    return k;
}

unsigned int fnv1a_hash(unsigned int key) {
    uint32_t hash = FNV_OFFSET_BASIS;

```

```c
    hash ^= ((key >> 24) & 0xFF);
    hash *= FNV_PRIME;

    hash ^= ((key >> 16) & 0xFF);
    hash *= FNV_PRIME;

    hash ^= ((key >> 8) & 0xFF);
    hash *= FNV_PRIME;

    hash ^= ((key)&0xFF);
    hash *= FNV_PRIME;

    return hash;
}

unsigned int djb2_hash(unsigned int key) {
    unsigned int hash = 5381;
    int c;

    hash = ((hash << 5) + hash) + ((key >> 24) & 0xFF); // hash * 33 + c
    hash = ((hash << 5) + hash) + ((key)&0xFF);         // hash * 33 + c
    hash = ((hash << 5) + hash) + ((key >> 8) & 0xFF);  // hash * 33 + c
    hash = ((hash << 5) + hash) + ((key >> 24) & 0xFF); // hash * 33 + c

    return hash;
}

unsigned int my_hash(unsigned long key) {
    unsigned int hash_1 = murmur_hash(~key);
    // unsigned int hash_2 = Crap8(hash_1);
    // unsigned int hash_1 = djb2_hash(~(key));
    // unsigned int hash_2 = fnv1a_hash(key);
    // return ((((hash_2 >> 3) & 0x3F) << 7) | (((hash_3 >> 5) & 0x7) << 4) |
    // ((hash_1 >> 7) & 0xF)); return ((((hash_2 >> 3) & 0x7F) << 7) | ((hash_1
    // >> 7) & 0x7F)); return ((hash_1 + (SEED_CRAP ^ hash_2)) >> 7) &
    // (CAPACITY_MOD);
    return hash_1 & (CAPACITY - 1);
}

void inline hash_lookup(unsigned long (*hash_table)[2], unsigned int key,
                        bool *hit, unsigned int *result) {

    key &= 0x1FFFFF; // make sure key is only 21 bits

    unsigned int hash_val = my_hash(key);

    unsigned long lookup = hash_table[hash_val][0];

    // [valid][value][key]
    unsigned long stored_key = lookup & 0x1FFFFF;      // stored key is 21 bits
    unsigned long value = (lookup >> 21) & 0x1FFF;     // value is 13 bits
    unsigned long valid = (lookup >> (21 + 13)) & 0x1; // valid is 1 bit

```

```
    if (valid && (key == stored_key)) {
        *hit = 1;
        *result = value;
        //*is_exists = 1;
        return;
    }

    lookup = hash_table[hash_val][1];

    // [valid][value][key]
    stored_key = lookup & 0x1FFFFF;       // stored key is 21 bits
    value = (lookup >> 21) & 0x1FFF;      // value is 13 bits
    valid = (lookup >> (21 + 13)) & 0x1; // valid is 1 bit
    if (valid && (key == stored_key)) {
        *hit = 1;
        *result = value;
        return;
    }
    *hit = 0;
    *result = 0;
}

void inline hash_insert(unsigned long (*hash_table)[2], unsigned int key,
                        unsigned int value, bool *collision) {

    key &= 0x1FFFFF; // make sure key is only 21 bits
    value &= 0x1FFF; // value is only 13 bits

    unsigned int hash_val = my_hash(key);

    unsigned long lookup = hash_table[hash_val][0];
    unsigned long valid = (lookup >> (21 + 13)) & 0x1;

    if (!valid) {
        hash_table[hash_val][0] =
            (1UL << (21 + 13)) | ((unsigned long)(value) << 21) | key;
        *collision = 0;
        return;
    }

    lookup = hash_table[hash_val][1];
    valid = (lookup >> (21 + 13)) & 0x1;
    if (valid) {
        *collision = 1;
        return;
    }
    hash_table[hash_val][1] =
        (1UL << (21 + 13)) | ((unsigned long)(value) << 21) | key;
    *collision = 0;
}

void inline assoc_insert(assoc_mem *mem, unsigned int key, unsigned int value,
                         bool *collision) {
    key &= 0x1FFFFF; // make sure key is only 21 bits
    value &= 0x1FFF; // value is only 13 bits

```

```
    unsigned int mem_fill = mem->fill;

    if (mem_fill < ASSOCIATIVE_MEM_STORE) {
        unsigned int key_high = (key >> 18) & 0x1FF;
        unsigned int key_middle = (key >> 9) & 0x1FF;
        unsigned int key_low = (key)&0x1FF;
        mem->upper_key_mem[key_high] |=
            (1UL << mem_fill); // set the fill'th bit to 1, while
   preserving
                               // everything else
        mem->middle_key_mem[key_middle] |=
            (1UL << mem_fill); // set the fill'th bit to 1, while
   preserving
                               // everything else
        mem->lower_key_mem[key_low] |=
            (1UL << mem_fill); // set the fill'th bit to 1, while
   preserving
                               // everything else
        mem->value[mem_fill] = value;
        mem->fill = mem_fill + 1;
        *collision = 0;
        return;
    }
    *collision = 1;
}

void inline assoc_lookup(assoc_mem *mem, unsigned int key, bool *hit,
                         unsigned int *result) {
    key &= 0x1FFFFF; // make sure key is only 21 bits
    unsigned int key_high = (key >> 18) & 0x1FF;
    unsigned int key_middle = (key >> 9) & 0x1FF;
    unsigned int key_low = (key)&0x1FF;

    unsigned long match_high = mem->upper_key_mem[key_high];
    unsigned long match_middle = mem->middle_key_mem[key_middle];
    unsigned long match_low = mem->lower_key_mem[key_low];

    unsigned long match = match_high & match_middle & match_low;

    if (match == 0) {
        *hit = 0;
        return;
    }

    unsigned int address = 0; //(unsigned int)(log2(match & -match) + 1)
   ;

    // Right shift until the rightmost set bit is found
    address += ((match & 0x00000000FFFFFFFFUL) == 0) ? 32 : 0;
    match >>= ((match & 0x00000000FFFFFFFFUL) == 0) ? 32 : 0;

    address += ((match & 0x000000000000FFFFUL) == 0) ? 16 : 0;
    match >>= ((match & 0x000000000000FFFFUL) == 0) ? 16 : 0;

    address += ((match & 0x00000000000000FFUL) == 0) ? 8 : 0;
    match >>= ((match & 0x00000000000000FFUL) == 0) ? 8 : 0;

    address += ((match & 0x000000000000000FUL) == 0) ? 4 : 0;
```

```
    match >>= ((match & 0x000000000000000FUL) == 0) ? 4 : 0;

    address += ((match & 0x0000000000000003UL) == 0) ? 2 : 0;
    match >>= ((match & 0x0000000000000003UL) == 0) ? 2 : 0;

    address += ((match & 0x0000000000000001UL) == 0) ? 1 : 0;
    match >>= ((match & 0x0000000000000001UL) == 0) ? 1 : 0;

    if (address != ASSOCIATIVE_MEM_STORE) {
        *result = mem->value[address];
        *hit = 1;
        return;
    }
    *hit = 0;
}

void inline insert(unsigned long (*hash_table)[2], assoc_mem *mem,
                   unsigned int key, unsigned int value, bool *collision
    ) {
    hash_insert(hash_table, key, value, collision);
    if (*collision) {
        assoc_insert(mem, key, value, collision);
    }
}

void inline lookup(unsigned long (*hash_table)[2], assoc_mem *mem,
                   unsigned int key, bool *hit, unsigned int *result) {
    hash_lookup(hash_table, key, hit, result);
    if (!*hit) {
        assoc_lookup(mem, key, hit, result);
    }
}

static void compute_lzw(hls::stream<unsigned char> &input_stream,
                        hls::stream<uint32_t> &out_stream,
                        uint32_t generic_info[4]) {

    // create hash table and assoc mem
    unsigned long hash_table[CAPACITY][2];
    assoc_mem my_assoc_mem;

#pragma HLS array_partition variable = hash_table complete dim = 2

// make sure the memories are clear
LOOP1:
    for (int i = 0; i < CAPACITY; i++) {
#pragma HLS UNROLL factor = 512
        hash_table[i][0] = 0;
        hash_table[i][1] = 0;
    }
    my_assoc_mem.fill = 0;

LOOP2:
    for (int i = 0; i < 512; i++) {
#pragma HLS UNROLL
        my_assoc_mem.upper_key_mem[i] = 0;
        my_assoc_mem.middle_key_mem[i] = 0;
        my_assoc_mem.lower_key_mem[i] = 0;
```

```
    }

    unsigned int next_code = 256;
    uint8_t failure = 0;
    uint32_t start_idx = generic_info[INFO_START_IDX];
    uint32_t end_idx = generic_info[INFO_END_IDX];
    unsigned int prefix_code = input_stream.read();
    unsigned int code = 0;
    unsigned char next_char = 0;
    uint64_t j = 0;

LOOP3:
    for (int i = start_idx; i < end_idx - 1; i++) {
        next_char = input_stream.read();
        bool hit = 0;
        lookup(hash_table, &my_assoc_mem, ((prefix_code << 8) + next_char),
               &hit, &code);
        if (!hit) {
            out_stream.write(prefix_code);
            bool collision = 0;
            insert(hash_table, &my_assoc_mem, ((prefix_code << 8) + next_char),
                   next_code, &collision);
            if (collision) {
                failure = 1;
            }
            next_code += 1;
            ++j;
            prefix_code = next_char;
        } else {
            prefix_code = code;
        }
    }

    out_stream.write(prefix_code);
    generic_info[INFO_OUT_PACKET_LENGTH] = j + 1;
    if (failure)
        generic_info[INFO_FAILURE] = failure;
}

static void store_data(hls::stream<uint32_t> &out_stream, uint32_t *lzw_codes) {
    unsigned int i = 0;

    while (!out_stream.empty()) {
        if (i < MAX_OUTPUT_CODE_SIZE) {
            lzw_codes[i] = out_stream.read();
            i++;
        }
    }
}

static void load_data(unsigned char input[BUFFER_LEN],
                      hls::stream<unsigned char> &input_stream) {
    for (int i = 0; i < BUFFER_LEN; i++) {
        input_stream.write(input[i]);
    }
```

```
}

static void compute_data(hls::stream<unsigned char> &input_stream,
                         hls::stream<uint32_t> &out_stream,
                         uint32_t out_packet_lengths[MAX_ITERATIONS],
                         uint32_t temp_chunk_indices[MAX_ITERATIONS]) {

    // This is an array that packs generic information for the LZW
    // function. The elements in the this array are as follows:
    // 0 - start_idx
    // 1 - end_idx
    // 2 - out_packet_length
    // 3 - failure
    uint32_t generic_info[INFO_LEN];
    generic_info[INFO_FAILURE] = 0;
    generic_info[INFO_OUT_PACKET_LENGTH] = 0;

    // The first element in the chunk_indices array contains the size
    // of the chunk_indices array.
    uint32_t chunk_indices_len = temp_chunk_indices[0];


    const int bound = chunk_indices_len - 1;

    for (int i = 1; i <= MAX_ITERATIONS; i++) {
#pragma HLS UNROLL factor = 2
        if (i <= bound) {
            generic_info[INFO_START_IDX] = temp_chunk_indices[i];
            generic_info[INFO_END_IDX] = temp_chunk_indices[i + 1];
            compute_lzw(input_stream, out_stream, generic_info);
            out_packet_lengths[i] = generic_info[INFO_OUT_PACKET_LENGTH
   ];
        }
    }

    while (!input_stream.empty()) {
        input_stream.read();
    }

    // The first element of the out_packet_lengths is going to signify
    // failure to insert into the associative memory.
    out_packet_lengths[0] = generic_info[INFO_FAILURE];
}

static void perform_lzw(unsigned char input[BUFFER_LEN],
                        uint32_t temp_chunk_indices[MAX_ITERATIONS],
                        uint32_t *lzw_codes,
                        uint32_t out_packet_lengths[MAX_ITERATIONS]) {

    hls::stream<unsigned char> in_stream("chunk_in");
    hls::stream<uint32_t> out_stream("lzw_out");

#pragma HLS STREAM variable = in_stream depth = 16384
#pragma HLS STREAM variable = out_stream depth = 40960

    // #pragma HLS DATAFLOW
    load_data(input, in_stream);
    compute_data(in_stream, out_stream, out_packet_lengths,
```

```
    temp_chunk_indices);
    store_data(out_stream, lzw_codes);
}

void lzw(unsigned char input[BUFFER_LEN], uint32_t *lzw_codes,
         uint32_t chunk_indices[MAX_ITERATIONS],
         uint32_t out_packet_lengths[MAX_ITERATIONS]) {

#pragma HLS INTERFACE m_axi port = input depth = 16384 bundle = p0
#pragma HLS INTERFACE m_axi port = lzw_codes depth = 40960 bundle = p1
#pragma HLS INTERFACE m_axi port = chunk_indices depth = 20 bundle = p0
#pragma HLS INTERFACE m_axi port = out_packet_lengths depth = 20 bundle
    = p0

    uint32_t temp_chunk_indices[MAX_ITERATIONS] = {0};

#pragma HLS array_partition variable = temp_chunk_indices block factor =
          \
    10 dim = 1

LOOP4:
    for (int i = 0; i < MAX_ITERATIONS; i++) {
#pragma HLS UNROLL
        temp_chunk_indices[i] = chunk_indices[i];
    }

    perform_lzw(input, temp_chunk_indices, lzw_codes, out_packet_lengths
    );
};
```

8.3.5 Host

```
#include "../Server/encoder.h"
#include "../Server/server.h"
#include "../Server/stopwatch.h"
#include "Utilities.h"
#include "common.h"
#include <chrono>
#include <condition_variable>
#include <errno.h>
#include <fcntl.h>
#include <iostream>
#include <pthread.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <thread>
#include <unistd.h>
#include <unordered_map>
#include <vector>

#define NUM_PACKETS 2
#define pipe_depth 4
#define DONE_BIT_L (1 << 7)
#define DONE_BIT_H (1 << 15)
```

```
#define CL_HPP_CL_1_2_DEFAULT_BUILD
#define CL_HPP_TARGET_OPENCL_VERSION 120
#define CL_HPP_MINIMUM_OPENCL_VERSION 120
#define CL_HPP_ENABLE_PROGRAM_CONSTRUCTION_FROM_ARRAY_COMPATIBILITY 1
#define CL_USE_DEPRECATED_OPENCL_1_2_APIS

using namespace std;

uint64_t dedup_bytes = 0;
uint64_t lzw_bytes = 0;

stopwatch time_cdc;
stopwatch time_lzw;
stopwatch time_sha;
stopwatch time_dedup;
stopwatch total_time;

typedef struct __attribute__((packed)) RawData {
    int length_sum;                 /// The total length of the input buffer.
    unsigned char *pipeline_buffer; /// Input data that needs to be processed.
    FILE *fptr_write;               /// File pointer to write the output.
    unsigned char *host_input;
    uint32_t *output_codes;
    uint32_t *chunk_indices;
    uint32_t *output_code_lengths;
} RawData;

typedef struct CLDevice {
    cl::Kernel kernel;
    cl::CommandQueue queue;
} CLDevice;

void handle_input(int argc, char *argv[], int *blocksize, char **filename,
                  char **kernel_name, unsigned int *chunk_size) {
    int x;
    extern char *optarg;

    while ((x = getopt(argc, argv, ":b:f:k:c:")) != -1) {
        switch (x) {
        case 'k':
            *kernel_name = optarg;
            printf("Kernel name is set to %s optarg\n", *kernel_name);
            break;
        case 'b':
            *blocksize = atoi(optarg);
            printf("blocksize is set to %d optarg\n", *blocksize);
            break;
        case 'c':
            *chunk_size = atoi(optarg);
            printf("Chunk size is set to %d optarg\n", *chunk_size);
            break;
        case 'f':
            *filename = optarg;
```

```
            printf("filename is %s optarg\n", *filename);
            break;
        case ':':
            printf("-%c without parameter\n", optopt);
            break;
        }
    }
}

static unsigned char *create_packet(int32_t chunk_idx,
                                    uint32_t out_packet_length,
                                    uint32_t *out_packet, uint32_t
   packet_len) {
    unsigned char *data =
        (unsigned char *)calloc(packet_len, sizeof(unsigned char));
    CHECK_MALLOC(data, "Unable to allocate memory for new data packet");

    uint32_t data_idx = 0;
    uint16_t current_val = 0;
    int bits_left = 0;
    int current_val_bits_left = 0;

    for (uint32_t i = 0; i < out_packet_length; i++) {
        current_val = out_packet[i];
        current_val_bits_left = CODE_LENGTH;

        if (bits_left == 0 && current_val_bits_left == CODE_LENGTH) {
            data[data_idx] = (current_val >> 5) & 0xFF;
            bits_left = 0;
            current_val_bits_left = 5;
            data_idx += 1;
        }

        if (bits_left == 0 && current_val_bits_left == 5) {
            if (data_idx < packet_len) {
                data[data_idx] = (current_val & 0x1F) << 3;
                bits_left = 3;
                current_val_bits_left = 0;
                continue;
            } else
                break;
        }

        if (bits_left == 3 && current_val_bits_left == CODE_LENGTH) {
            data[data_idx] |= ((current_val >> 10) & 0x07);
            bits_left = 0;
            data_idx += 1;
            current_val_bits_left = 10;
        }

        if (bits_left == 0 && current_val_bits_left == 10) {
            if (data_idx < packet_len) {
                data[data_idx] = ((current_val >> 2) & 0xFF);
                bits_left = 0;
                data_idx += 1;
                current_val_bits_left = 2;
            } else
                break;
```

```
        }

        if (bits_left == 0 && current_val_bits_left == 2) {
            if (data_idx < packet_len) {
                data[data_idx] = (current_val & 0x03) << 6;
                bits_left = 6;
                current_val_bits_left = 0;
                continue;
            } else
                break;
        }

        if (bits_left == 6 && current_val_bits_left == CODE_LENGTH) {
            data[data_idx] |= ((current_val >> 7) & 0x3F);
            bits_left = 0;
            data_idx += 1;
            current_val_bits_left = 7;
        }

        if (bits_left == 0 && current_val_bits_left == 7) {
            if (data_idx < packet_len) {
                data[data_idx] = (current_val & 0x7F) << 1;
                bits_left = 1;
                current_val_bits_left = 0;
                continue;
            } else
                break;
        }

        if (bits_left == 1 && current_val_bits_left == CODE_LENGTH) {
            data[data_idx] |= ((current_val >> 12) & 0x1);
            bits_left = 0;
            data_idx += 1;
            current_val_bits_left = 12;
        }

        if (bits_left == 0 && current_val_bits_left == 12) {
            if (data_idx < packet_len) {
                data[data_idx] = ((current_val >> 4) & 0xFF);
                bits_left = 0;
                data_idx += 1;
                current_val_bits_left = 4;
            } else
                break;
        }

        if (bits_left == 0 && current_val_bits_left == 4) {
            if (data_idx < packet_len) {
                data[data_idx] = (current_val & 0x0F) << 4;
                bits_left = 4;
                current_val_bits_left = 0;
                continue;
            } else
                break;
        }

        if (bits_left == 4 && current_val_bits_left == CODE_LENGTH) {
            data[data_idx] |= ((current_val >> 9) & 0x0F);
```

```
                data_idx += 1;
                bits_left = 0;
                current_val_bits_left = 9;
            }

            if (bits_left == 0 && current_val_bits_left == 9) {
                if (data_idx < packet_len) {
                    data[data_idx] = ((current_val >> 1) & 0xFF);
                    bits_left = 0;
                    data_idx += 1;
                    current_val_bits_left = 1;
                } else
                    break;
            }

            if (bits_left == 0 && current_val_bits_left == 1) {
                data[data_idx] = (current_val & 0x01) << 7;
                bits_left = 7;
                current_val_bits_left = 0;
                continue;
            }

            if (bits_left == 7 && current_val_bits_left == CODE_LENGTH) {
                data[data_idx] |= ((current_val >> 6) & 0x7F);
                bits_left = 0;
                current_val_bits_left = 6;
                data_idx += 1;
            }

            if (bits_left == 0 && current_val_bits_left == 6) {
                if (data_idx < packet_len) {
                    data[data_idx] = ((current_val)&0x3F) << 2;
                    bits_left = 2;
                    current_val_bits_left = 0;
                    continue;
                } else
                    break;
            }

            if (bits_left == 2 && current_val_bits_left == CODE_LENGTH) {
                if (data_idx < packet_len) {
                    data[data_idx] |= ((current_val >> 11) & 0x03);
                    bits_left = 0;
                    data_idx += 1;
                    current_val_bits_left = 11;
                } else
                    break;
            }

            if (bits_left == 0 && current_val_bits_left == 11) {
                if (data_idx < packet_len) {
                    data[data_idx] = ((current_val >> 3) & 0xFF);
                    bits_left = 0;
                    data_idx += 1;
                    current_val_bits_left = 3;
                } else
                    break;
            }
```

```

        if (bits_left == 0 && current_val_bits_left == 3) {
            if (data_idx < packet_len) {
                data[data_idx] = ((current_val)&0x07) << 5;
                bits_left = 5;
                current_val_bits_left = 0;
                continue;
            } else
                break;
        }

        if (bits_left == 5 && current_val_bits_left == CODE_LENGTH) {
            data[data_idx] |= ((current_val >> 8) & 0x1F);
            bits_left = 0;
            current_val_bits_left = 8;
            data_idx += 1;
        }

        if (bits_left == 0 && current_val_bits_left == 8) {
            if (data_idx < packet_len) {
                data[data_idx] = (current_val & 0xFF);
                bits_left = 0;
                current_val_bits_left = 0;
                data_idx += 1;
                continue;
            } else
                break;
        }
    }
    return data;
}

static void compression_pipeline(
    RawData *r_data, unsigned char *host_input, uint32_t *output_codes,
    uint32_t *chunk_indices, uint32_t *output_code_lengths, CLDevice dev
    ,
    cl::Buffer lzw_input_buffer, cl::Buffer lzw_output_buffer,
    cl::Buffer chunk_indices_buffer, cl::Buffer
    out_packet_lengths_buffer,
    unsigned int chunk_size) {

    vector<uint32_t> vect;
    string sha_fingerprint;
    int64_t chunk_idx = 0;
    uint32_t packet_len = 0;
    uint32_t header = 0;
    vector<int64_t> dedup_out;
    vector<pair<pair<int, int>, unsigned char *>> final_data;

    //
    -----------------------------------------------------------------
    // Step 3: Run the kernel
    //
    -----------------------------------------------------------------

    std::vector<cl::Event> write_event(1);
    std::vector<cl::Event> compute_event(1);
    std::vector<cl::Event> done_event(1);
```

```
    // double total_time_2 = 0;

    memcpy(host_input, r_data->pipeline_buffer,
           sizeof(unsigned char) * r_data->length_sum);

    total_time.start();

    // RUN CDC
    time_cdc.start();
    fast_cdc(r_data->pipeline_buffer, r_data->length_sum, chunk_size,
   vect);
    time_cdc.stop();

    chunk_indices[0] = vect.size();

    std::copy(vect.begin(), vect.end(), chunk_indices + 1);

    for (int i = 0; i < (int)(vect.size() - 1); i++) {
        // RUN SHA
        time_sha.start();
        sha_fingerprint =
            sha_256(r_data->pipeline_buffer, vect[i], vect[i + 1]);
        time_sha.stop();

        // RUN DEDUP
        time_dedup.start();
        chunk_idx = dedup(sha_fingerprint);
        dedup_out.push_back(chunk_idx);
        time_dedup.stop();
    }

    // RUN LZW
    time_lzw.start();

    dev.kernel.setArg(0, lzw_input_buffer);
    dev.kernel.setArg(1, lzw_output_buffer);
    dev.kernel.setArg(2, chunk_indices_buffer);
    dev.kernel.setArg(3, out_packet_lengths_buffer);

    dev.queue.enqueueMigrateMemObjects({lzw_input_buffer,
   chunk_indices_buffer},
                                        0 /* 0 means from host*/, NULL,
                                        &write_event[0]);

    dev.queue.enqueueTask(dev.kernel, &write_event, &compute_event[0]);

    // Profiling the kernel.
    /* compute_event[0].wait(); */
    /* total_time_2 +=
     * compute_event[0].getProfilingInfo<CL_PROFILING_COMMAND_END>() -
   */
    /* compute_event[0].getProfilingInfo<CL_PROFILING_COMMAND_START>();
   */

    dev.queue.enqueueMigrateMemObjects(
        {lzw_output_buffer, out_packet_lengths_buffer},
        CL_MIGRATE_MEM_OBJECT_HOST, &compute_event, &done_event[0]);
```

```
    clWaitForEvents(1, (const cl_event *)&done_event[0]);
    time_lzw.stop();

    if (output_code_lengths[0] & 0x1) {
        printf("FAILED TO INSERT INTO ASSOC MEM!!\n");
        exit(EXIT_FAILURE);
    }

    uint32_t *output_codes_ptr = output_codes;

    for (int i = 1; i < (int)chunk_indices[0]; i++) {
        if (dedup_out[i - 1] == -1) {
            packet_len = ((output_code_lengths[i] * 13) / 8);
            packet_len = ((output_code_lengths[i] & 0x7) != 0) ?
packet_len + 1
                                                               :
packet_len;

            unsigned char *data_packet =
                create_packet(chunk_idx, output_code_lengths[i],
                              output_codes_ptr, packet_len);

            header = packet_len << 1;
            final_data.push_back({{header, packet_len}, data_packet});
            lzw_bytes += 4;

            lzw_bytes += packet_len;

        } else {
            header = (dedup_out[i - 1] << 1) | 1;
            final_data.push_back({{header, -1}, NULL});
            dedup_bytes += 4;
        }

        output_codes_ptr += output_code_lengths[i];
    }
    total_time.stop();

    for (auto it : final_data) {
        if (it.second != NULL) {
            fwrite(&it.first.first, sizeof(uint32_t), 1, r_data->
fptr_write);
            fwrite(it.second, sizeof(unsigned char), it.first.second,
                   r_data->fptr_write);
            free(it.second);
        } else {
            fwrite(&it.first.first, sizeof(uint32_t), 1, r_data->
fptr_write);
        }
    }

    /* cout << "Total Kernel Execution Time using Profiling Info: " <<
     * total_time_2 */
    /*      << " ms." << endl; */
}

int main(int argc, char *argv[]) {

```

```
    stopwatch ethernet_timer;
    stopwatch compression_timer;
    unsigned char *input[NUM_PACKETS];
    int writer = 0;
    int done = 0;
    int length = -1;
    uint64_t offset = 0;
    ESE532_Server server;

    int blocksize = BLOCKSIZE;
    char *file = strdup("compressed_file.bin");
    char *kernel_name = strdup("lzw.xclbin");
    unsigned int chunk_size = CHUNK_SIZE;

    // set blocksize if decalred through command line
    handle_input(argc, argv, &blocksize, &file, &kernel_name, &chunk_size);

    RawData *r_data = (RawData *)calloc(1, sizeof(RawData));
    CHECK_MALLOC(r_data, "Unable to allocate memory for raw data");

    r_data->fptr_write = fopen(file, "wb");
    if (r_data->fptr_write == NULL) {
        printf("Error creating file for compressed output!!\n");
        exit(EXIT_FAILURE);
    }

    r_data->pipeline_buffer =
        (unsigned char *)calloc(NUM_PACKETS * blocksize, sizeof(unsigned char));
    CHECK_MALLOC(r_data->pipeline_buffer,
                 "Unable to allocate memory for pipeline buffer");

    for (int i = 0; i < (NUM_PACKETS); i++) {
        input[i] = (unsigned char *)calloc((blocksize + HEADER),
                                           sizeof(unsigned char));
        CHECK_MALLOC(input, "Unable to allocate memory for input buffer");
    }

    server.setup_server(blocksize);

    writer = 0;

    // ----------------------------------------------------------------------
    // Step 1: Initialize the OpenCL environment
    // ----------------------------------------------------------------------
    cl_int err;
    std::string binaryFile = kernel_name;
    unsigned fileBufSize;
    std::vector<cl::Device> devices = get_xilinx_devices();
    devices.resize(1);
    cl::Device device = devices[0];
    cl::Context context(device, NULL, NULL, NULL, &err);
    char *fileBuf = read_binary_file(binaryFile, fileBufSize);
    cl::Program::Binaries bins{{fileBuf, fileBufSize}};
```

```
    cl::Program program(context, devices, bins, NULL, &err);
    CLDevice dev;
    dev.queue = cl::CommandQueue(context, device,
                                    CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE,
    &err);
    dev.kernel = cl::Kernel(program, "lzw", &err);

    //
   -----------------------------------------------------------------------
    // Step 2: Create buffers and initialize test values
    //
   -----------------------------------------------------------------------

    cl::Buffer lzw_input_buffer =
        cl::Buffer(context, CL_MEM_READ_ONLY,
                   sizeof(unsigned char) * NUM_PACKETS * blocksize, NULL
   , &err);
    unsigned char *host_input = (unsigned char *)dev.queue.
   enqueueMapBuffer(
        lzw_input_buffer, CL_TRUE, CL_MAP_WRITE, 0,
        sizeof(unsigned char) * NUM_PACKETS * blocksize);

    cl::Buffer lzw_output_buffer =
        cl::Buffer(context, CL_MEM_WRITE_ONLY,
                   sizeof(uint32_t) * MAX_OUTPUT_BUF_SIZE, NULL, &err);
    uint32_t *output_codes = (uint32_t *)dev.queue.enqueueMapBuffer(
        lzw_output_buffer, CL_TRUE, CL_MAP_READ, 0,
        sizeof(uint32_t) * MAX_OUTPUT_BUF_SIZE);

    cl::Buffer chunk_indices_buffer =
        cl::Buffer(context, CL_MEM_READ_ONLY, sizeof(uint32_t) *
   MAX_LZW_CHUNKS,
                   NULL, &err);
    uint32_t *chunk_indices = (uint32_t *)dev.queue.enqueueMapBuffer(
        chunk_indices_buffer, CL_TRUE, CL_MAP_WRITE, 0,
        sizeof(uint32_t) * MAX_LZW_CHUNKS);

    cl::Buffer out_packet_lengths_buffer =
        cl::Buffer(context, CL_MEM_WRITE_ONLY,
                   sizeof(uint32_t) * MAX_LZW_CHUNKS, NULL, &err);
    uint32_t *output_code_lengths = (uint32_t *)dev.queue.
   enqueueMapBuffer(
        out_packet_lengths_buffer, CL_TRUE, CL_MAP_READ, 0,
        sizeof(uint32_t) * MAX_LZW_CHUNKS);

    compression_timer.start();

    // Loop until last message
    while (!done) {

        ethernet_timer.start();
        server.get_packet(input[writer]);
        ethernet_timer.stop();

        // get packet
        unsigned char *buffer = input[writer];

        // decode
```

```
     done = buffer[1] & DONE_BIT_L;
     length = buffer[0] | (buffer[1] << 8);
     length &= ~DONE_BIT_H;

     offset += length;
     r_data->length_sum += length;

     // Perform the actual computation here. The idea is to maintain a
     // buffer, that will hold multiple packets, so that CDC can chunklength)
     // at appropriate boundaries. Call the compression pipeline function
     // after the buffer is completely filled.
     if (length != 0)
         memcpy(r_data->pipeline_buffer + (writer * blocksize),
                input[writer] + 2, length);

     if (writer == (NUM_PACKETS - 1) || (length < blocksize && length > 0) ||
         done == 1) {
         compression_pipeline(
             r_data, host_input, output_codes, chunk_indices,
             output_code_lengths, dev, lzw_input_buffer, lzw_output_buffer,
             chunk_indices_buffer, out_packet_lengths_buffer, chunk_size);
         writer = 0;
         r_data->length_sum = 0;
     } else
         writer += 1;
 }

 for (int i = 0; i < (NUM_PACKETS); i++)
     free(input[i]);

 fclose(r_data->fptr_write);
 dev.queue.enqueueUnmapMemObject(lzw_input_buffer, host_input);
 dev.queue.enqueueUnmapMemObject(lzw_output_buffer, output_codes);
 dev.queue.enqueueUnmapMemObject(chunk_indices_buffer, chunk_indices);
 dev.queue.enqueueUnmapMemObject(out_packet_lengths_buffer,
                                 output_code_lengths);
 dev.queue.finish();

 free(r_data->pipeline_buffer);
 free(r_data);

 compression_timer.stop();

 // Print Latencies
 cout << "--------------- Total Latencies ---------------" << endl;
 cout << "Total latency of CDC is: " << time_cdc.latency() << " ms." << endl;
 cout << "Total latency of LZW is: " << time_lzw.latency() << " ms." << endl;
 cout << "Total latency of SHA256 is: " << time_sha.latency() << " ms."
```

```
        << endl;
  cout << "Total latency of DeDup is: " << time_dedup.latency() << " ms."
        << endl;
  cout << "Total time taken: " << total_time.latency() << " ms." << endl;
  cout << "---------------- Average Latencies ------------" << endl;
  cout << "Average latency of CDC per loop iteration is: "
        << time_cdc.avg_latency() << " ms." << endl;
  cout << "Average latency of LZW per loop iteration is: "
        << time_lzw.avg_latency() << " ms." << endl;
  cout << "Average latency of SHA256 per loop iteration is: "
        << time_sha.avg_latency() << " ms." << endl;
  cout << "Average latency of DeDup per loop iteration is: "
        << time_dedup.avg_latency() << " ms." << endl;
  cout << "Average latency: " << total_time.avg_latency() << " ms."
        << std::endl;

  std::cout << "\n\n";

  std::cout << "--------------- Key Throughputs ---------------" << std::endl;
  float ethernet_latency = ethernet_timer.latency() / 1000.0;
  float compression_latency = compression_timer.latency() / 1000.0;
  float compression_latency_total_time = total_time.latency() / 1000.0;

  float cdc_latency_total_time = time_cdc.latency() / 1000.0;
  float sha_latency_total_time = time_sha.latency() / 1000.0;
  float lzw_latency_total_time = time_lzw.latency() / 1000.0;
  float dedup_latency_total_time = time_dedup.latency() / 1000.0;

  float compression_throughput =
      (offset * 8 / 1000000.0) / compression_latency; // Mb/s
  float compression_throughput_2 =
      (offset * 8 / 1000000.0) / compression_latency_total_time; // Mb/s

  float cdc_throughput =
      (offset * 8 / 1000000.0) / cdc_latency_total_time; // Mb/s
  float sha_throughput =
      (offset * 8 / 1000000.0) / sha_latency_total_time; // Mb/s
  float lzw_throughput =
      (offset * 8 / 1000000.0) / lzw_latency_total_time; // Mb/s
  float dedup_throughput =
      (offset * 8 / 1000000.0) / dedup_latency_total_time; // Mb/s

  float ethernet_throughput =
      (offset * 8 / 1000000.0) / ethernet_latency; // Mb/s

  cout << "Ethernet Latency: " << ethernet_latency << "s." << endl;
  cout << "Bytes Received: " << offset << "B." << endl;
  cout << "Latency for Compression: " << compression_latency << "s." << endl;
  cout << "Latency for Compression (without fwrite): "
        << compression_latency_total_time << "s." << endl;

  std::cout << "\n";
```

```
    cout << "Latency for CDC: " << cdc_latency_total_time << "s." <<
    endl;
    cout << "Latency for SHA: " << sha_latency_total_time << "s." <<
    endl;
    cout << "Latency for LZW: " << lzw_latency_total_time << "s." <<
    endl;
    cout << "Latency for DEDUP: " << dedup_latency_total_time << "s." <<
     endl;

    std::cout << "\n";

    cout << "CDC Throughput: " << cdc_throughput << "Mb/s." << endl;
    cout << "SHA Throughput: " << sha_throughput << "Mb/s." << endl;
    cout << "LZW Throughput: " << lzw_throughput << "Mb/s." << endl;
    cout << "DEDUP Throughput: " << dedup_throughput << "Mb/s." << endl;

    std::cout << "\n";

    cout << "Ethernet Throughput: " << ethernet_throughput << "Mb/s." <<
     endl;
    cout << "Application Throughput: " << compression_throughput << "Mb/
    s."
         << endl;
    cout << "Application Throughput (without fwrite): "
         << compression_throughput_2 << "Mb/s." << endl;
    cout << "Bytes Contributed by Deduplication: " << dedup_bytes << "B.
    "
         << endl;
    cout << "Bytes Contributed by LZW: " << lzw_bytes << "B." << endl;

    std::cout << "\n";

    return 0;
}
```

8.3.6 Testbench

```
#include "../Encoder/common.h"
#include <iostream>
#include <stdlib.h>
#include <string.h>
#include <unordered_map>
#include <vector>

#define FILE_SIZE 16383
#define MAX_LZW_CODES (4096 * 20)

using namespace std;

// "Golden" functions to check correctness
std::vector<int> encoding(std::string s1) {
    // std::cout << "Encoding\n";
    std::unordered_map<std::string, int> table;
    for (int i = 0; i <= 255; i++) {
        std::string ch = "";
        ch += char(i);
```

```
        table[ch] = i;
    }

    std::string p = "", c = "";
    p += s1[0];
    int code = 256;
    std::vector<int> output_code;
    // std::cout << "String\tOutput_Code\tAddition\n";
    for (int i = 0; i < s1.length(); i++) {
        if (i != s1.length() - 1)
            c += s1[i + 1];
        if (table.find(p + c) != table.end()) {
            p = p + c;
        } else {
            //              std::cout << p << "\t" << table[p] << "\t\t"
            //                        << p + c << "\t" << code << std::endl;
            output_code.push_back(table[p]);
            table[p + c] = code;
            code++;
            p = c;
        }
        c = "";
    }
    // std::cout << p << "\t" << table[p] << std::endl;
    output_code.push_back(table[p]);
    return output_code;
}

//************************************************************************
int main() {
    // FILE *fptr =
    // fopen("/home1/r/ruturajn/ESE532/ese532_project/project/Text_Files/Franklin.txt",
    // "r"); FILE *fptr =
    // fopen("/home1/r/ruturajn/ESE532/ese532_project/project/Text_Files/lotr.txt",
    // "r"); FILE *fptr =
    // fopen("/home1/r/ruturajn/ESE532/ese532_project/project/Text_Files/imaggeee.jpg",
    // "r"); FILE *fptr =
    // fopen("/home1/r/ruturajn/ESE532/ese532_project/project/Text_Files/IMAGE_390.jpg",
    // "r"); FILE *fptr = fopen("./ruturajn.tgz", "r");
    FILE *fptr = fopen("/home1/r/ruturajn/ESE532/ese532_project/project/"
                       "Text_Files/LittlePrince.txt",
                       "r");
    // FILE *fptr =
    // fopen("/home1/r/ruturajn/ESE532/ese532_project/project/Text_Files/small_prince.txt",
    // "r"); FILE *fptr = fopen("/home1/r/ruturajn/Downloads/embedded_h5.JPG",
    // "rb"); FILE *fptr =
    // fopen("/home1/r/ruturajn/Downloads/ESE5070_Assignment3_ruturajn-1.pdf",
    // "rb");
    // FILE *fptr = fopen("/home1/r/ruturajn/Downloads/FiraCode.zip", "rb");
```

```
    // FILE *fptr =
    // fopen("/home1/r/ruturajn/ESE532/ese532_project/project/encoder.xo
    ", "r");
    if (fptr == NULL) {
        printf("Unable to open file!\n");
        exit(EXIT_FAILURE);
    }

    fseek(fptr, 0, SEEK_END);       // seek to end of file
    int64_t file_sz = ftell(fptr); // get current file pointer
    fseek(fptr, 0, SEEK_SET);       // seek back to beginning of file

    unsigned char file_data[16384];
    uint32_t chunk_indices[20];
    uint32_t *lzw_codes = (uint32_t *)calloc(40960, sizeof(uint32_t));
    memset(lzw_codes, 23, 40960 * sizeof(uint32_t));
    if (lzw_codes == NULL) {
        cout << "Unable to allocate memory for lzw codes!" << endl;
        exit(EXIT_FAILURE);
    }
    uint32_t out_packet_lengths[20];
    unsigned int count = 0;
    bool fail_stat = false;

    while (file_sz > 0) {

        size_t bytes_read = fread(file_data, 1, 16384, fptr);

        if (file_sz >= 16384 && bytes_read != 16384)
            printf("Unable to read file contents");

        vector<uint32_t> vect;

        if (file_sz < 16384)
            fast_cdc(file_data, (unsigned int)file_sz, 4096, vect);
        else
            fast_cdc(file_data, (unsigned int)16384, 4096, vect);

        chunk_indices[0] = vect.size();

        std::copy(vect.begin(), vect.end(), chunk_indices + 1);

        lzw(file_data, lzw_codes, chunk_indices, out_packet_lengths);

        uint32_t *lzw_codes_ptr = lzw_codes;

        if (out_packet_lengths[0]) {
            cout << "TEST FAILED!!" << endl;
            cout << "FAILED TO INSERT INTO ASSOC MEM!!\n";
            exit(EXIT_FAILURE);
        }

        uint32_t packet_len = 0;
        uint32_t header = 0;
        vector<pair<pair<int, int>, unsigned char *>> final_data;

        for (int i = 1; i <= chunk_indices[0] - 1; i++) {
            std::string s;
```

```
            char *temp = (char *)file_data + chunk_indices[i];
            int count = chunk_indices[i];

            while (count++ < chunk_indices[i + 1]) {
                s += *temp;
                temp += 1;
            }

            std::vector<int> output_code = encoding(s);

            if (out_packet_lengths[i] != output_code.size()) {
                cout << "TEST FAILED!!" << endl;
                cout << "FAILURE MISMATCHED PACKET LENGTH!!" << endl;
                cout << out_packet_lengths[i] << "|" << output_code.size
   ()
                     << "at i = " << i << endl;
                fail_stat = true;
                exit(EXIT_FAILURE);
            }

            for (int j = 0; j < output_code.size(); j++) {
                if (output_code[j] != lzw_codes_ptr[j]) {
                    if (!fail_stat)
                        fail_stat = true;
                    cout << "FAILURE!!" << endl;
                    cout << output_code[j] << "|" << lzw_codes_ptr[j]
                         << " at j = " << j << " and i = " << i << endl;
                }
            }

            lzw_codes_ptr += out_packet_lengths[i];
        }

        file_sz -= 16384;

        cout << "Iteration : " << count << endl;
        count++;
    }

    fclose(fptr);

    if (fail_stat)
        cout << "TEST FAILED!!" << endl;
    else
        cout << "TEST PASSED!!" << endl;

    return 0;
}
```

We, **Ruturaj A. Nanoti**, **Akhil Gunda**, and **Siddhant Mathur**, certify that we have complied with the University of Pennsylvania's Code of Academic Integrity in completing this final exercise.