ESE-532

Final Project Report

Sheil Sarda

Anthony Stewart

Shaokang Xia

December 10, 2020

Ultra96 Design

|  |  |  |  |
| --- | --- | --- | --- |
| **On CPU** | **On FPGA** | | |
| Input |  |  |  |
|  | CDC1 | CDC2 | … |
|  | SHA1 | SHA2 | … |
| LZW |  |  |  |
| Output |  |  |  |

# I/O

**Payload Size**

**Our implementation uses a payload size of 8192 Bytes. We have explored the continuum of payload sizes, as small as 256 Bytes, and as large as the Client would allow.**

**Based on our findings, we believe that the payload size vs compression ratio curve hits an inflection point around 8192 Bytes. Any payload size under 8192 Bytes leads to a compression ratio at or below 1, depending on the input file size. Larger files tend to lead to better compression ratios. Payload sizes over 8192 Bytes start to take too much storage on the CPU and FPGA end, leading to unpredictable behavior.**

**Hence, we decided to stick with this payload size for both our FPGA and CPU implementations.**

|  |  |
| --- | --- |
| *Challenge* | Increasing the payload size reduces memory transfer overhead between calls to the pipeline at the cost of increased storage for larger payloads. |
| *Opportunity* | Increasing the payload size allows LZW more opportunity to compress duplicate occurrences of input chunks and gives CDC more opportunities to identify the best chunk boundaries to maximize duplicates. |
| *Continuum* | Reducing payload size increases the number of CDC, SHA and LZW computations but reduces storage and memory transfer |

**Reading Packets**

**The number of packets our FPGA implementation reads in from the server currently equals the number of CDC compute units we decide to instantiate.**

**This design choice is guided in our belief that we want to minimize the memory our CPU devotes to storing the input, and other intermediate results of our pipeline to avoid unpredictable behavior caused by memory overflow errors.**

**To this end, we have also minimized our use of dynamic memory constructs like unordered maps and vectors which are C++ standard data types based on our empirical findings that they are quite inefficient in memory usage compared to Pure C alternatives.**

**On the flip side, however, by only reading in enough packets that are required by CDC in the next cycle of the pipeline we incur the memory transfer overhead between the server and the host at every iteration.**

**While we have explored the continuum of caching more input payloads in the Host, a potential next step for our analysis would be to copy over > 1 payload to each CDC module we instantiate, and then passing on the completed result (chunks from > 1 payload) to the SHA and later the LZW.**

**This method of *Burst Reads and Writes*, recommended to us by Syed in OH + reference** [here](https://arxiv.org/pdf/2010.01547.pdf)**, could lead to an overall improvement in throughput if the increase in bytes outputted outweighs the corresponding increase in cycle time of the bottleneck pipeline stage (LZW in our case) since each stage is now processing > 1 Payload in a cycle.**

# FPGA Acceleration

CDC

**Max Chunk Length**

**The max chunk length directly affects the compression ratio of our encoder implementation. Larger chunk lengths tend to produce more duplicates within the current payload, and in turn allow the later stages of the pipeline to effectively minimize the number of chunks encoded by the LZW. In our design we used a max chunk length equal to the incoming payload size to maximize the number of duplicates found later by our hash map.**

**Storing CDC chunks**

**In our implementation, instead of modifying the input packet containing the file data, we instead generate an integer, Bounds, of the same size and instantiate it with all 0’s. CDC modifies this Bounds array by changing the index corresponding to the beginning of a chunk so that it references that chunks number instead. This Bounds array gets passed through the pipeline along with the current packet.**

**This simple Bounds array implementation allows for an easy and repeatable way to reference individual chunks within the given payload by eliminating any need for dynamic memory allocation due to CDC finding a variable number of chunks.**

**No Data Level Parallelism**

**Our CDC implementation lacks the ability to add data parallelism due to the nature of the algorithm itself. Since we move our sliding window across the entire payload to generate chunks, this means every chunk we find will depend of the bounds and fingerprint of the previously found chunk.**

**Adding data level parallelism would involve splitting up each payload and chunking each individual piece in parallel. This could result in unfavorable bounds for each chunk depending on how the initial payload is split, also making it harder to maximize the number of duplicates found and in turn improve the compression ratio.**

SHA

**Number of SHA Hardware units**

**Since the throughput of the SHA stage of the pipeline scales linearly with the number of SHA and CDC compute units we instantiate, this is a crucial axis to determine our overall throughput. Based on how we designed our Ultra96 design with both CDC and SHA on the FPGA, the number of SHA hardware units equals the number of CDC units we are using.**

**Type of memory used to store SHA**

**In our FPGA implementation, we compute a SHA hash for each chunk generated by CDC inside a payload and store it in a 2D array of 4 Byte unsigned integers. The number of rows in this 2D array equals the number of CDC chunks generated for that payload, which is 16 in our implementation based on a Minimum Chunk Size of 512 Bytes and a Payload size of 8192 Bytes.**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Block 1** | b94d27b9 | 934d3e08 | a52e52d7 | da7dabfa | c484efe3 | 7a5380ee | 9088f7ac | e2efcde9 |
| **Block 2** | 96b022ce | 4ca4a6af | aeb8bf7d | 2bd10979 | 883a9257 | 5374c51a | 197cf5e7 | 848575a3 |
| **…** |  |  |  |  |  |  |  |  |

**Other implementation techniques we considered before arriving at this heap allocated memory solution was using stack-based memory such as C++ vectors or unordered maps but decided against them based on our empirical evidence and implementation notes found. We believe the memory overhead of using dynamically sized data structures is too large for our FPGA because they start causing memory map errors like double free or munmap\_chunk. While a workaround we found was to use reserve() on maps and vectors based on theoretical upper bounds we determined specific to our implementation, but the limited memory resources on the CPU were still causing issues.**

**The singular purpose of the SHA array in the pipeline is to use the unique hash of each chunk to identify duplicates and avoid LZW encoding the same chunk twice. The most intuitive solution we found for this was a C++ map of type <int, int> which had the following structure:**

|  |  |
| --- | --- |
| **Key (int)** | **Value (int)** |
| **Index where the SHA256 hash of a unique hash is stored inside the SHA array** | **Index of the LZW encoding of the chunk written to file** |

**Due to the previously referenced issues associated with dynamic sized data structures and the CPU, we were not able to directly implement this idea. Instead, we modeled the functionality of a map using parallel heap-stored arrays for the key and the value. Based on the above implementation details, our pseudocode for handling LZW encoding is as follows:**

**Step-1:**  Compute SHA-256 Hash of chunk

**Step-2:**  Search for a matching hash in the Key array

**Step-3 (a):** If match found, use the index of the matching SHA256 from the keys array to fetch corresponding LZW encoded chunk index from values array

**Step-4:** Generate header for the duplicate chunk using the LZW encoded chunk index and write to output file

**Step-3 (b):** If no match found, generate LZW encoding for the chunk

**Step-4:** Store the SHA256 index of this chunk to the keys array and store the value of the running index of number of LZW encodings generated to the values array

**Step-5:** Generate header for the unique chunk using the LZW encoding and write to output file

**The worst-case size of the parallel arrays we are using for storing the keys and values occurs when all chunks in the payload are found to be unique. Thus, since our implementation has a max number of chunks of 16 (computed in the previous section), we can use that as an upper bound for allocating the heap memory used for these structures.**

# LZW

**Data structure used to store the LZW Tree**

**The size of the LZW tree needs to be dynamic since the LZW algorithm generates new code words when trying to encode the input. Hence, the most intuitive solution is to use a dynamically resizable data structure like a C++ vector, and use the reserve( ) method provided by the class to prevent memory access issues. The theoretical upper bound we used for the reserve( ) method was Payload Size (8192 Bytes).**

**However, as mentioned in the considerations for our SHA implementation, this approach gave us many memory related issues when storing or retrieving code words through the C++ vector interface: using the push\_back( ) function to insert, and the [] operator to retrieve elements.**

**The workaround we implemented was creating a heap-allocated array of strings with number of rows equal to payload size, since that was our upper bound on number of unique LZW codewords.**

**Codeword output and padding**

**A code length of 13 bits was used for the LZW encodings due to the 8KB payload size previously mentioned above. Using a code length of 13 introduces the need for padding to ensure the decoder and interpret our compressed output. to eliminate the need for padding we attempted to increase the payload size to 64KB to produce a code word size of 16 bits, but we were limited by the provided client.**

**Aligning codewords to 8-bit boundary**

To properly align the codewords from LZW, we initially converted each code word to a binary string and manually padded with the extra zeroes to ensure it ended on an 8-bit boundary. Once we had our padded string, we used the CPP stoul( ) function, which interprets an unsigned integer value from a string to convert the binary string into u\_int8 bit integers to be written to the output file. While this solution allowed us to quickly test the entirety of the pipeline, in turn we sacrificed both memory and speed due to the use of strings.

**For the final implementation of LZW, we moved to padding the codewords in Pure C, eliminating the calls to String class methods like stoul( ). We achieved this by using bitwise ops to select each codeword from the string and shift in zeros appropriately to achieve the correct 8-bit boundary padding.**

**This implementation opened the possibility of performing LZW encoding on the FPGA if we so desired. Due to time constraints caused by the various memory overflow issues associated with the previous LZW implementation which used CPP vectors, we were not able to take advantage of the FPGA speedup for this segment of the pipeline by porting this over to the ARM FPGA.**

CPU Design



# CDC

The chunk size is maximized, in which we utilize the tradeoff between varying chunk and window sizes and throughput of CDC; after tuning, the final payload size is determined to be 8192 bytes. The smaller the max chunk size, the smaller number of bits that need to be used in the rolling hash computation. The time required per hash computation is independent of the rolling window size, since we use a rolling hash function.

The function is processed in pipeline. Multiple compute units are assigned for CDC, ideally there would be a linear increase in the throughput as we add more compute units.

The data inside CDC is processed in a bitwise manner, and the operations are data independent, thus allowing data level parallelism. We can use vectors to reduce Initialization Interval of the pipeline

# SHA

Multiple SHA compute units are implemented on the CPU to increase the throughput; the throughput increases linearly with the number of compute units. The encoding procedure of SHA has no data-dependency; thus, we can unroll the for-loop and achieve data-level parallelism.

# LZW

The encoded result is taken as the output of the function, then added to the pipeline. Vectors are used for implementing data-level parallelism and reduce the Initialization Interval.

Validation

# I/O

For I/O we needed to ensure that packets were read into the pipeline correctly and that our compressed output was written to the file correctly. For the input, we worked with the provided client to ensure that there was no packet loss by tuning the delay between the sending of each packet.

The output was validated in two separate parts, the header and the compressed chunk. For the header we validated its correctness by using the hexdump Linux command to check the endianness of the bytes being written to the file. For the compressed chunk, we needed to make sure that the output file size was accurately reflecting the compressed chunk size and also padded correctly as specified in the write-up. This was also done using hexdump along with the provided decoder and a simple text file to verify correctness.

# CDC

We initially began the validation of our CDC implementation using various c units test and sample inputs to test the implementations ability to discover duplicate chunks. This form of validation led us to finding a significant flaw in or initial implementation, causing us to move towards using an open-source implementation instead.

The open-source implementation was validated for correctness the same way as above. Once we were confident that the chunking was occurring as we intended, we began testing various parameters to see which yielded the best results for deduplication. This testing was also done using C unit tests.

The FPGA implementation of CDC was the same as the CPU implementation since we made sure to only use basic c code to implement it. This allowed us to validate both implementations simultaneously, only needing minor fixes on the FPGA side to ensure that the hardware synthesized correctly.

# SHA

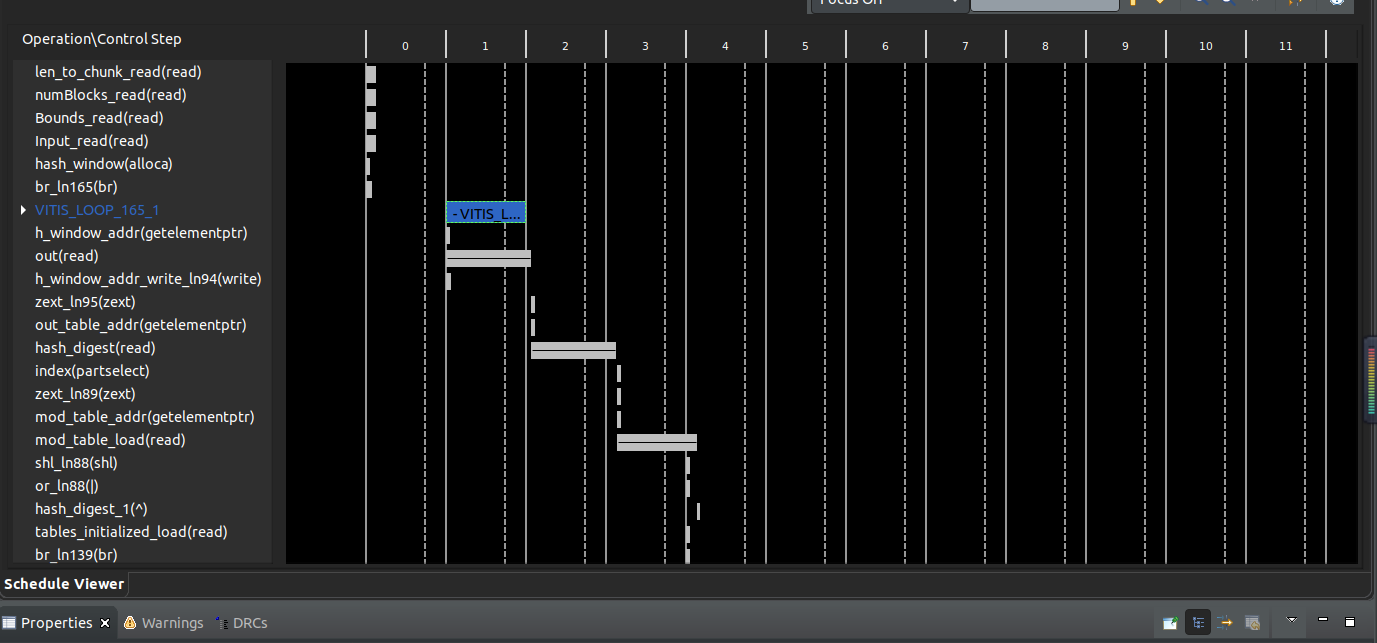
For SHA, we followed a similar validation workflow as we did with CDC, the major difference being the lack of tunable parameters for SHA performance. Since we were using a 256b hash, we needed a way to store the hash given that the max integer size for the ARM is 64 bits. Splitting up each hash across different integers proved to be an issue that required a lot of validation to ensure the endianness of the hash was preserved.

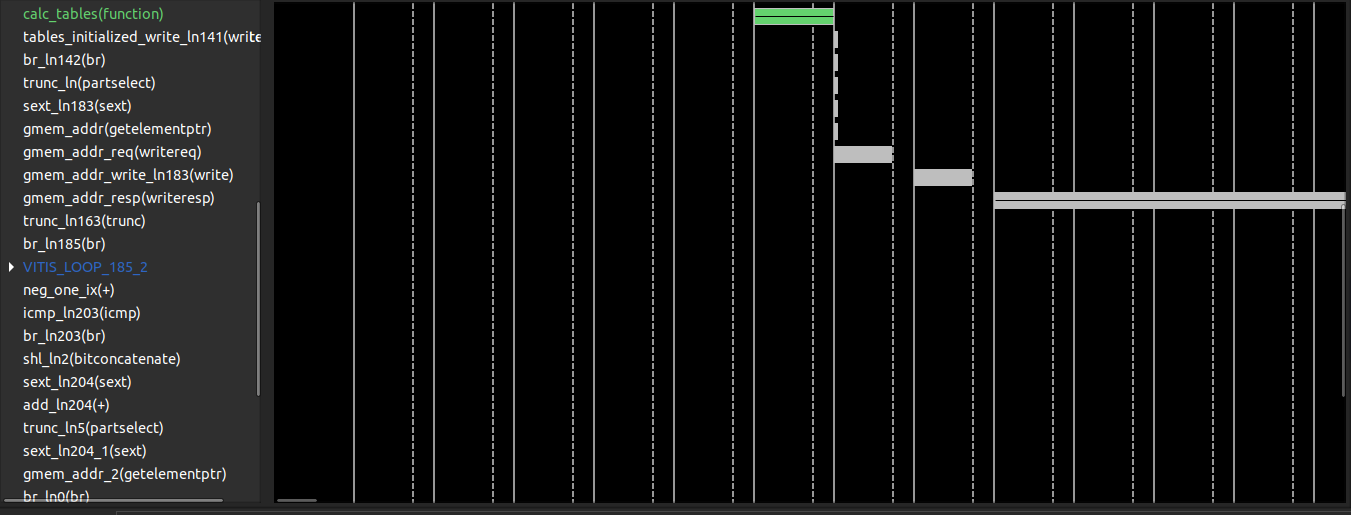
# LZW

Validation for LZW was primarily done by running the entire compression pipeline, producing a compressed output, decoding that output, and comparing it to the original file. To compare files, we utilized the diff and vimdiff Linux commands. Since the LZW is running on the ARM CPU, there was no need to use Vitis HLS to test functionality and hardware compatibility.

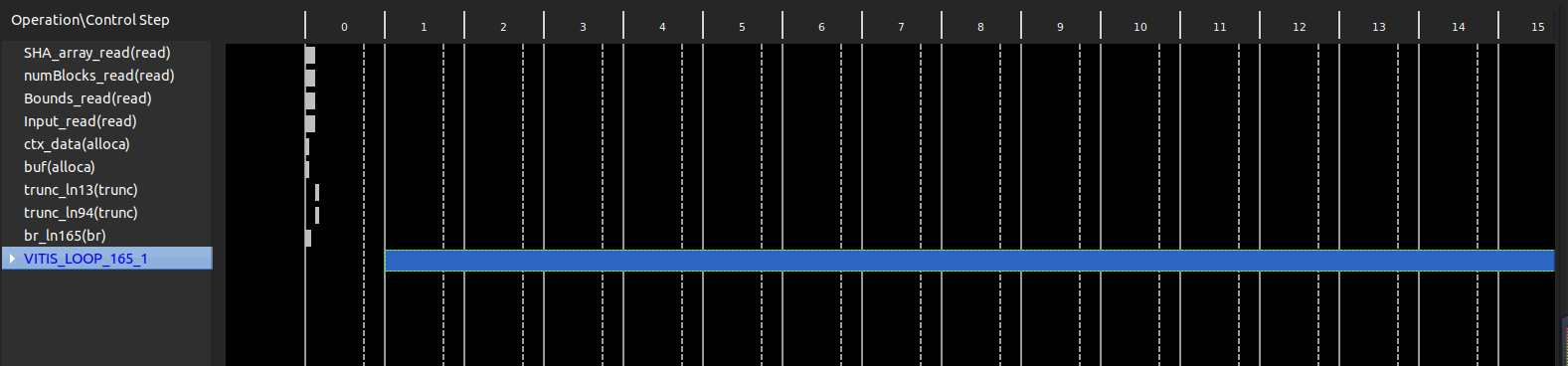
Design space

Maximized chunk size for CDC function; assign multiple compute units for the major components; splitting up the FPGA and CPU resources.

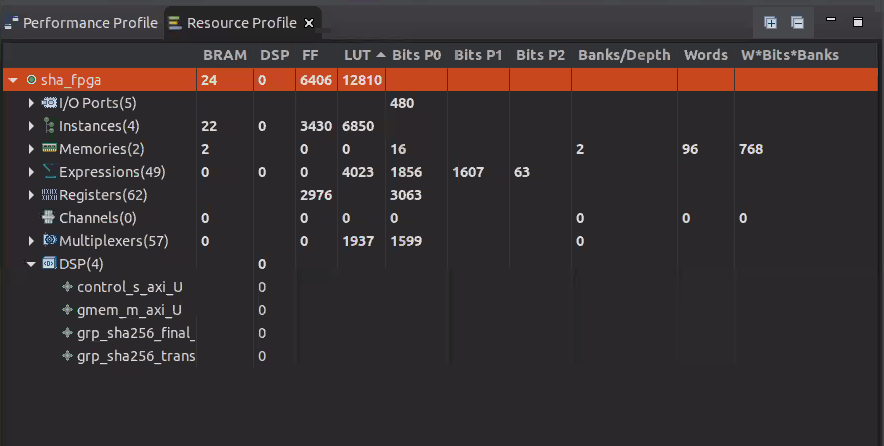
Data variables on memory: (for CDC)



Data variables on memory: (for SHA)



Resource utilization



Teamwork + Learnings = FUN :)

Sheil Sarda

* Revised Design Axes for different pipeline modules based on Prof. DeHon’s feedback on Milestone 2.
* Created Vitis HLS projects with testbenches to synthesize and run C simulations on our CDC and SHA FPGA designs.
* Modified our FPGA Host.cpp code to support multiple instantiations of the CDC + LZW pipeline, using the Vitis HLS synthesized modules. Helped debug the code on the FPGA.
* Helping port over CPP code into Pure C for SHA.

Shaokang Xia

* Design Axis
* Timing analysis in Vitis HLS
* Milestone reports

Anthony Stewart

* Encoder I/O, validating encoder input and outputs
* LZW Implementation, c++ vector implementation in Milestone 2 and pure C implementation for final submission
* CDC optimization, parameter tuning based on common industry implementations (Restic)
* Porting CDC and SHA to FPGA, modifying pure C implementation from GitHub

Coding Resources

CDC

* <https://github.com/fd0/rabin-cdc/blob/master/rabin.c>\*  
  Implements Content Defined Chunking (CDC) based on a rolling Rabin Checksum. Uses Pure C so that it can easily be ported to FPGA.
* <https://github.com/fanzhang312/Data-deduplication>  
  Implements Fixed Size chunking as opposed to CDC, so not useful for identifying duplicate chunks. Hence, we moved over to the above implementation after Milestone 2.

SHA

* <https://github.com/B-Con/crypto-algorithms/blob/master/sha256.c>\*  
  Implements SHA256 in Pure C, as opposed to the below repository which uses CPP constructs like Vectors and Maps.
* <https://github.com/amosnier/sha-2/blob/master/sha-256.c>

LZW

* <https://www.geeksforgeeks.org/lzw-lempel-ziv-welch-compression-technique/>  
  Provides an overview of the LZW algorithm as well as pseudocode for a potential implementation.

\*Code used in final implementation

STATEMENT

I, *Sheil Sarda*, certify that I have complied with the University of Pennsylvania’s Code of Academic Integrity in completing this final exercise.

I, *Anthony I. Stewart*, certify that I have complied with the University of Pennsylvania’s Code of Academic Integrity in completing this final exercise.

I, *Shaokang Xia*, certify that I have complied with the University of Pennsylvania’s Code of Academic Integrity in completing this final exercise.