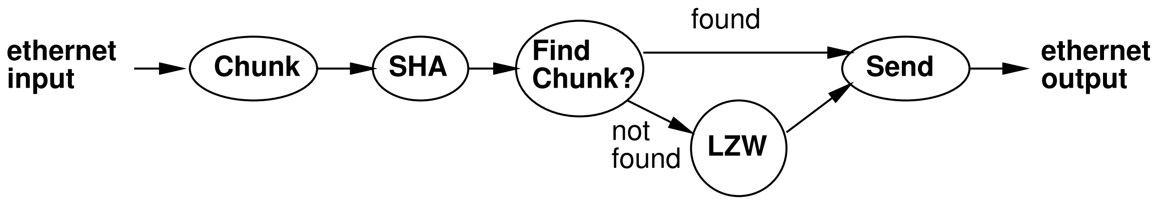
University of Pennsylvania
Department of Electrical and System Engineering
System-on-a-Chip Architecture

ESE532, Fall 2020 Deduplication and Compression Project Wednesday, October 28

Due: Thursday, December 10, 5:00PM

1 Goal

Develop a compressor that can receive data in real time at modern ethernet speeds (1Gb/s) and compress it into memory using deduplication and compression. Specifically, we'll look at Content-Defined Chunking to break the input into chunks, SHA-256 hashes to screen for duplicate chunks, and LZW compression to compress non-duplicate chunks. For full points, your goal for implementation is to achieve real-time guaranteed support of 1 Gb/s input stream, but you may need to consider intermediate goals (e.g. 100Mb/s, 300Mb/s) along the way. Slower designs will receive partial credit for the performance portion of the project grade.



This is a 5-week project assignment; the intent is to allow you to plan and execute a significant, open-ended design exploration and mapping. You will not achieve the implementation goal or the course learning goals by trying to do this in one week. We give you milestones to help provide some structure, but the milestones are minimal and *doing the minimum to hit the milestone each week will be insufficient* to get you where you need to be at the end. We are giving you flexibility in planning and ordering rather than lock-step specifying exactly what you need to do each week.

- Project work is done in teams of 3. You select partners during first week.
- Collaboration between teams is limited as specified on the course web page.
- Milestones writeups and Final report are a team writeup.
- The spirit of this exercise is to optimize the SoC mapping of the algorithm. As such, explorations of alternate solutions that change the algorithms and generally optimize the solution for hardware and software are out of scope. Explorations that tweak or tune the algorithms slightly to better exploit the SoC hardware are potentially in scope.

2 Final Report

Final report is a team writeup. There will be one turnin per team.

- Describe your single ARM processor mapped design. [1 page]
  - Key parameters in the solution
  - Performance achieved
  - Compression achieved
  - Characterization and breakdown of time spent in the major components
- Describe your final Ultra96 mapped design. [5 pages]
  - Performance achieved and energy required
  - Compression achieved
  - Key design aspects: task decomposition, parallelism, mapping to Zynq resources, include diagrams to support
  - Be clear where each component of the final design is performed (e.g., ARM, NEON vector, FPGA logic).
  - Model to explain performance.
  - Current bottleneck preventing higher performance
- Describe how you validated your implementations, including the real-time guarantee for the input rate. The real-time validation likely includes both arguments about the way the code is written and mapped to the Zynq and your testing methodology. [2 pages]
- Describe the key lessons you learned from this design experience. [1 page]
- Describe design space explored and show graphs and models to support design selection. [any number of pages as needed]
- Describe who did what. [1 page]
- Include academic integrity statement for all team members:

> I, *your-name-here*, certify that I have complied with the University of Pennsylvania's Code of Academic Integrity in completing this final exercise.

You can review the Code of Academic Integrity here: https://catalog.upenn.edu/pennbook/code-of-academic-integrity/

3 Final Project Code and Bitstream Submission

We intend to run your compression routines. To make sure the process is consistent across teams, please comply with the following standards.

1. Provide an xclbin, OpenCL host code executable, and decoder executable for your encoder.
   - Turn in a tar file to the designated final implementation assignment on canvas.
   - One turnin for team.
   - Should be a single tar file, containing three files:
     - `encoder.xclbin` for FPGA kernel
     - `encoder` for OpenCL host code executable
     - `decoder` executable configured to work with your encoded file. (Most likely, this is just a compilation of the `Decoder.cpp` we supplied; however, if you chose a different maximum block size, you may need to change `CODE_LENGTH`; so give us back one with that change made.)
2. Your compression program (OpenCL host code) should take one argument:
   - the file name where the program should store the compressed data.

   Your program should assume that `encoder.xclbin` is in the same directory as the host executable.
3. Your compression program should start up ready to receive inputs.

We may provide further clarification or revision on this final implementation turnin as we get closer.

4 Milestones

We will provide precise requirements for milestones each week. These may include a few exercises to help prepare you for questions that may be on the final in addition to the project specific components. Milestones and feedback feed into the final report. In most cases, the milestones can serve as a first draft of a component of your report, and the feedback we give you will help provide guidance on how to refine it for the report.

1. Analysis, parallelism, placeholder encoder, and teaming [11/6]
2. Functional version and design space [11/13]
3. Real-Time input integration and first operator on FPGA [11/20]
4. Intermediate throughput design (e.g., try for 200Mb/s) [12/4]
5. Final Report [12/10]

5 Components

The components we will use are standard enough that the wikipedia pages are useful, and there are several other nice tutorial blog posts out there. Here's a roundup of starting points.

- Content-Defined Chunking (Rabin Fingerprint)
  - https://moinakg.wordpress.com/tag/rabin-fingerprint/
  - https://restic.net/blog/2015-09-12/restic-foundation1-cdc
  - https://en.wikipedia.org/wiki/Rabin_fingerprint
- SHA 256 Hashing
  - https://tools.ietf.org/html/rfc6234
  - https://en.wikipedia.org/wiki/SHA-2
  - An example possibly useful for testing: https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Standards-and-Guidelines/documents/examples/SHA256.pdf
  - You may use the hardwired SHA unit on the ZCU3EG.
- Lempel-Ziv-Welch Compression
  - http://www.geeksforgeeks.org/lzw-lempel-ziv-welch-compression-technique/
  - https://en.wikipedia.org/wiki/Lempel-Ziv-Welch

6 Some Suggested Parameters

There will be some discretion in picking implementation parameters. From the start we will suggest considering:

- 4KB average chunk size with a 8KB maximum
- Use on-board DRAM for the chunk dictionary and hash fingerprints
- Full, maximum chunk size as the LZW compression window

You may want to experiment with some of the parameters when tuning your implementation.

7 Examples of Use

1. Yan Zhang, Nirwan Ansari, Mingquan Wu, and Heather Yu. "On Wide Area Network Optimization." In *IEEE Communications Surveys & Tutorials*, vol. 4, issue 4, pp. 1090–113, Oct. 2013. http://ieeexplore.ieee.org/document/6042388/ Sections III A and B survey the role of compresison and decompression in optimizing WAN data traffic.
2. Ashok Anand, Chitra Muthukrishnan, Aditya Akella, and Ramachandran Ramjee. "Redundancy in Network Traffic: Findings and Implications". In *Proceedings of ACM SIGMETRICS/Performance*, 2009. https://www.microsoft.com/en-us/research/publication/redundancy-in-network-traffic-findings-and-implications/ Characterizes redundancy in network traffic.
3. Athicha Muthitacharoen, Benjie Chen, and David Mazieres. 2001. A low-bandwidth network file system. In *Proceedings of the eighteenth ACM symposium on Operating systems principles (SOSP '01)*. pp. 174-187. https://dl.acm.org/citation.cfm?id=502052 Use of deduplication for optimizing a file system operating across a low bandwith link.

8 Chunk Validation

Using an SHA-256 signature, the probability of having a collision where two chunks share the same signature is extremely low. For the project, we will consider equality of SHA-256 signatures adequate to determine that a chunk is a duplicate. This means you do not need to read back the chunk and validate that it is, in fact, identical. If you had terabytes of data, or if the consequences of error were high, you would want to perform the check. This only applies to the full 256b signature. If you use smaller hashes for indexing, you will still need to validate that there is a match on the 256b signature.

9 Other Resources

- Xilinx HLS Tiny Tutorials https://github.com/Xilinx/HLS-Tiny-Tutorials. These show examples of how to write code for specific things in Vitis HLS.
- Vitis Tutorials https://xilinx.github.io/Vitis-Tutorials/master/docs/README.html. These are written for data center platforms (AWS F1/Alveo cards), but the decomposition and core routines should still be useful.
- Vitis Accel Examples https://xilinx.github.io/Vitis_Accel_Examples/master/html/index.html These show examples of how to write code for specific things in both Vitis OpenCL and HLS code.
- For examples of other applications that have been converted to run with PL accelerators in HLS, you can look at:
  - the Rosetta Stone Benchmarks https://github.com/cornell-zhang/rosetta from Cornell. These were written for SDSoC, but the decomposition and core routines should still be useful.
  - Parallel Programming for FPGAs by Kastner et al. http://kastner.ucsd.edu/wp-content/uploads/2018/03/admin/pp4fpgas.pdf

.

10 Encoded Data Storage

For your timing runs, store the compressed data in DRAM. The largest test case we provide is under 200MB. Copy your encoded data from DRAM to the SD-Card outside of your test timing. You will need to plan how you divide the DRAM among buffers, chunk hash storage, and compressed output. During early testing, before adding external input, you may also want to start with the uncompressed data in DRAM.

11 Compressed Format

- Compressed stream is a sequential concatenation of chunks.
- Each chunk has a 32b header that identifies it as Duplicate Chunk or LZW Chunk.
  - A Duplicate Chunk is a 32b value
    - bit 0 is a 1 to signify a Duplicate Chunk
    - bits 31–1 is the Chunk Index of previously encoded block to be duplicated. Only LZW Chunks are indexed. The first LZW chunk has index 0, the next 1, etc.
  - An LZW Chunk is
    - a 32b bit header
      - bit 0 is 0 to signify an LZW Chunk
      - bits 31–1 is the *compressed* chunk length in bytes
    - LZW-compressed contents of the chunk. LZW implementations vary. Our implementation satisfies the following properties:
      - Entries 0–255 of the dictionary are initialized to the 256 literals, e.g. a byte with value 27 would be encoded as 27.
      - The next dictionary entries are used for prefixes: sequences of 2 or more bytes.
      - No special keywords such as end-of-file are contained in the dictionary.
      - The dictionary size is only limited by the chunk size limitation.
      - For simplicity, we set all code lengths to be $\lceil \log_2 MaxChunkSize \rceil$ in this project. The provided *decoder* works with the *MaxChunkSize = 8192 bytes*. You can change the $MaxChunkSize$ by changing the defined parameter `CODE_LENGTH`.
      - Code words are output MSB-first. Assuming nothing has been output yet, a 13b code with binary value $x_{12}x_{11}x_{10}x_9x_8x_7x_6x_5x_4x_3x_2x_1x_0$ results in two consecutive bytes with values $x_{12}x_{11}x_{10}x_9x_8x_7x_6x_5$ and $x_4x_3x_2x_1x_0000$. The next code word with value $y_{12}y_{11}y_{10}y_9y_8y_7y_6y_5y_4y_3y_2y_1y_0$ changes the second byte to, $x_4x_3x_2x_1x_0y_{12}y_{11}y_{10}$, the third to $y_9y_8y_7y_6y_5y_4y_3y_2$ and the fourth to $y_1y_0000000$.
    - Padding so that the entire LZW chunk ends on an 8b boundary; that is, chunks of either type always begin on 8b boundaries.

Because we are using a uniform length of $\lceil \log_2 MaxChunkSize \rceil > 8$ for encoding codewords, it is possible that an encoded chunk could be longer than the unencoded chunk. To deal with this, real implementations will often compare the length of the LZW encoded chunk to the raw chunk length and send the unencoded data if it is smaller. For simplicity, we are not asking you to perform that optimization (and our encoded chunk format does not support it).

12 Supplied Resources

- Laptop code and/or scripts to send data to your Ultra96 at a fixed (tunable) frequency. *To be provided later.*
- Dummy design to receive packets from sender. *To be provided later.*
- Reference implementation of decoder (see `project/Decoder` in course code repository).
- We provide several datasets that you can use for testing. We encourage you to create your own simple datasets for unit testing. Note that the tar-files are not meant to be unpacked. Following are the datasets that we provide.
  - The Little Prince unencoded. As an encoding example, we also provide The Little Prince compressed.
  - Simple example. This archive contains three files, two of which are identical.
  - Benjamin Franklin's autobiography. This is a simple text file that you can modify for your own purposes. The current file probably has few duplicate areas. (390 KB)
  - GTK+ source code. This file contains several subsequent versions of the GTK+ source, which provides ample opportunity for deduplication. (177 MB)
  - Linux source code. This file contains several subsequent versions of the source. (191 MB)
  - Several Linux kernels. As opposed to the other data sets, this set contains prevalently binary data. (66 MB)

  Note: you should take these as examples, not a definitive list of test cases. In particular, *you should create many other focused test examples to facilitate your debugging and validation.*