ESE532 Project P2 Report

Ritika Gupta, Taylor Nelms, and Nishanth Shyamkumar

November 8, 2019

1 Design Space Axes

1. **CDC:**

   (a) **Axis:** Multiple CDC HW resources
   Challenge: Improving throughput of chunks created from input file. Ideally linear increase in throughput.
   Opportunity: Not a viable opportunity, as although the input space can be divided up for each resource, the chunks generated from the split stages cannot be utilized by later stages until all chunks in the first resource have been computed.
   Continuum: From 1 to how many can fit into fabric. At higher resource numbers, the chunkable data stream would reduce thus creating more overhead in LZW compression. This probably creates diminishing returns.
   Equation for Benefit: Total throughput = (No.of resources * Throughput of individual CDC)

   (b) **Axis:** Using vector engines for computation
   Challenge: Improving throughput by leveraging independent data level parallelism
   Opportunity: CDC design does not give this opportunity because data is operated in a byte wise manner. Secondly, there is no data independent operation as start of next chunk depends on end of previous chunk. Hard to define independent boundaries.
   Continuum: Dividing data as 8x16bit blocks or 4x32 etc.
   Equation for benefit: A linear improvement as earlier:
   N = no.of vector lanes, then N * computation on a single lane

   (c) **Axis:** Using pipelining stages on microlevel
   Challenge: Allowing CPI of 1, removing sequential constraints.
   Opportunity: CDC design isn't ideal for pipelining. Although it works at a byte level granularity with sequential ordered access, where each byte is slid into the window and hash is recalculated for the new window, there is a cyclic dependency with the hash digests between stages due to the rolling hash implementation. This prevents an achievable II of 1.
   Continuum: Even from 2 stage to N stages, the algorithm does not allow for full pipeline utilization.
   Equation for benefit: Throughput increases based on depth of pipeline, in this case, for N stages, Nx increase in calculating chunk boundary throughput.

   (d) **Axis:** Using pipelining stages on macro level
   Challenge: Implementing a dataflow within HW functions to prevent memory accesses and efficient flow of data so that all the HW functions in effect are computing every cycle and thus pipelined.
   Opportunity: Identifying size of local stores to be used as a RAM interface or using streaming the bytes to next level but figuring out a mechanism to indicate chunk boundary. For CDC, the chunk boundary in the worst case is MAXCHUNKSIZE bytes. A local store of this size would be needed to hold this chunk before pushing it through to the next HW function. Alternatively, a smaller buffer of 8bytes could be used with an additional header appended to indicate presence of another 8 bytes. If the chunk boundary has been hit, then a packet with a header signalling last packet can be sent through. This allows quicker data transfer, with a smaller local store, but with the overhead of a header byte.
   Continuum: Using a large RAM interface creates a memory bottleneck, if partitioning the interface, it uses up FPGA resources. Streaming removes this continuum equation as its all at a byte granularity.
   Equation for benefit: Throughput improves at a macro level due to reduction in memory access time and allowing all functions to be busy with computations effectively utilising the parallel nature of the FPGA.

   (e) **Axis:** Data mover selection and data access pattern for CDC.
   Challenge: Selecting the correct data mover between the PS and PL sections and in the process ensuring hardware code is altered to suit the data mover and conforms to the data access pattern.
   Opportunity: Using the right data mover for the hardware function arguments is important in order to reduce resource usage and improve efficiency. Example for small scalar arguments, an AXI LITE data mover would be

the right choice. For CDC, the data entering the rabin logic is either a file content, or network packets. Since these data bytes are stored in DRAM, it can be copied over or more ideally it can be streamed a byte at a time. A sequential interface is ideal for byte accesses in order, e.g. for CDC and LZW. However the code needs to be restructured, such as using local stores like registers and line buffers to enable streaming. For array and structure arguments containing arrays, the choices can be multiple, it can use AXIDMA SIMPLE or AXIDMA SG movers for normal data copy. If we are using shared memory, then an AXI master interface is used. Shared memory access should use buffering of data so that data can be transferred in bursts rather than a single byte access to DDR memory. The data access pattern ensures that for an ap fifo(sequential) interface, data is accessed in order and only once. For random access pattern, a normal data copy can be used.
Continuum: Depending on the data length and number of accesses of the data needed by the HW function, the data mover can vary from the lightweight AXI LITE to a more complicated AXIDMA SG. These factors need to be considered and interfacing code restructured to support the appropriate data mover.

(f) **Axis:** Optimizing memory accesses.
Challenge: As touched upon above, this deals with efficient usage of local store to alleviate DRAM accesses which bottleneck FPGA. Using clever local stores and initialization to prevent DRAM penalty of million cycles.
Opportunity: A great mechanism to prevent computation as well as memory redundancies. For example, if the same computation is repeated multiple times, its efficient to store the output locally and reuse it. Specifically, a buffer of window size bytes can be used for the rabin fingerprint calculation. There are multiple references to the fingerprint value within the rabin logic, which can also be stored locally and accessed. For memory accesses, reusing memory already read from DRAM by storing it locally prevents data memory re-access and heavy data movement penalties. Local store can also be an stop gap between hardware functions in a dataflow setup, once more alleviating DRAM access.
Continuum: Depending on how much data needs to be stored locally, registers with multi port accesses can be an approach, or if a large BRAM is required then, memory ports may turn into bottlenecks. These factors should be considered.
Equation for benefit: saves about million clock cycles per redundant memory access.

(g) **Axis:** Resource Usage
Challenge: Limiting resource usage to fit within the FPGA provided resources.
Opportunity: BRAMs provide a convenient local store for large data types, but also introduce memory bottlenecks in the form of limited read/write ports. Using array partitions, data can be moved across BRAMs or even stored in registers but with its own downsides.
Continuum: Large data stores use BRAMs, but memory bottleneck. Smaller data stores remove bottlenecks but cause heavy resource usage and long FPGA synthesis times.
Equations for benefit: memory port bottleneck = constant * memory store size

(h) **Axis:** Mapping tasks to CPU, FPGA.
Challenge: The individual functions can be split among CPU cores and FPGA logic.
Opportunity: Deciding if data level parallelism can be exploited on the CPU Vector engine, perhaps for SHA 512 bit blocks. Maybe having multi core usage of a functionality so that each core forms a stage in a micro pipeline and the associated syncing issues present with such an approach. CPUs also can benefit from cacheability and higher clock speeds for computations. Can also do rabin and SHA initializations on separate cores initially, although according to Amdahl's law as the incoming data size increases, the initialization stage becomes negligible.
Continuum: How many stages split across multiple cores and the complexity of synchronization between cores as the number of stages increases.
Equations for benefit: Ideally, for N stages, a linear increase in throughput and at higher clock speeds, thus improving on baseline CPU performance.

(i) **Axis:** Varying size of max chunk length and window size
Challenge: Identify the tradeoffs between chunk sizes and other performance.
Opportunity: There is a correlation between max chunk size and the number of lower order bits that need to be checked against a rabin mask. So the smaller the max chunk size, the less number of bits need to be compared. Altering window size is more flexible because of the rolling hash implementation. It always removes a byte, and calculates the rabin fingerprint on previously calculated fingerprint and new byte. This operation will be constant time irrespective of rolling window size. Also enables easier implementations for lzw table.
Continuum: Reducing max chunk size allows more boundaries to be identified, but also reduces the size of chunk that gets duplicated, also involving more comparisons with the SHA table. Increasing chunk size, causes other boundaries to be encapsulated within the identified chunk. But it allows less comparisons with SHA table.
Equations for benefit: Chunk size = constant * (1 / Comparisons with SHA table) Chunk size = constant *

boundaries found

2. **SHA:**

   (a) **Axis:** S, Number of SHA-256 hardware units
   Challenge: Improving throughput of hashing step
   Opportunity: Send chunks to rotating SHA unit index to allow for parallel execution
   Continuum: Anywhere from 1 to however many of our hardware SHA units will fit on the FPGA
   Equation for Benefit: $\texttt{Throughput}(S) = S * \texttt{singleSHAUnitThroughput}$

   (b) **Axis:** K, No. of SHA sub-chunk computation unit for computing on 64 byte sub-chunks
   Challenge: Improving throughput of hashing step
   Opportunity: Send sub chunks of 64 bytes to sub-chunk computation units for parallel execution and each stores its result which can all be added to get the final hash values
   Continuum: Anywhere from 1 to chunk size divided by 64 bytes(size of 1 sub-chunk).
   Equation for Benefit: $\texttt{Throughput}(S) = K * \texttt{singleSHASubUnitThroughput}$

   (c) **Axis:** P, Type of memory(units of partitioned memory) to store the input chunk data of SHA
   Challenge: Improving throughput of hashing step
   Opportunity: Partition the input chunk array so that they can be read simultaneously to send over to sub-chunk computation unit
   Continuum: Anywhere from a depth of 1 to 64 which is the sub-chunk size so that each sub-chunk input data can be read from memory simultaneously. Makes more sense to have a depth of 128 because there are 2 ports and that's how input data for computation of 2 sub chunks can be read simulatneously.
   Equation for Benefit: $\texttt{Throughput}(S) = P * \texttt{singleChunkMemReadTime}$

   (d) **Axis:** H, Type of memory to store the hash values
   Challenge: Improving throughput of hashing step
   Opportunity: Partition the array storing eight 32-bit values so that they can be read/written simultaneously
   Continuum: Anywhere from 1 to 8.
   Equation for Benefit: $\texttt{Throughput}(S) = M * \texttt{singleMemRead/Write}$

   (e) **Axis:** A, Pipeline depth for hashing
   Challenge: Improving throughput of hashing step
   Opportunity: Pipeline hashing function so that computation for next chunk can start when current is still being computed
   Continuum: Anything from 2 to a value before or at which pipelining benefits can be observed. After that value, it will start to increase the latency due to pipleine overheads as it results in diminishing returns.
   Equation for Benefit: It increases the throughput. If earlier it was taking n cycles for an output, then with careful setting of pipeline depth and pipeline stages, it has the ability to result in 1 output per cycle.

   (f) **Axis:** DM, Data movement between sub-chunk computation units of SHA - save on memory
   Challenge: Improving latency of SHA computation for a chunk and saving memory to store intermediate hash values
   Opportunity: Stream intermediate hash values from 1 unit to another in order to do perform addition to generate the final hash values
   Continuum: The continuum here is basically about how much memory to be used as FIFO which can vary from 1 32-bit location to 8 of them. The latter is simply the case where all the 8 hash values are being stored on memory, nothing really being streamed.
   Equation for Benefit: For n 64 byte subchunks, memory utilization would be 32*n bytes. With streaming, it could go as low as 1 byte to no memory at all.

3. **LZW:**

   (a) **Axis:** L, Number of LZW hardware units
   Challenge: Improving throughput of LZW step
   Opportunity: Send chunks to rotating LZE unit index to allow for parallel execution
   Continuum: Anywhere from 1 to however many of our hardware LZW units will fit on the FPGA (BRAM likely limiting factor)
   Equation for Benefit: $\texttt{Throughput}(L) = S * \texttt{singleLZWUnitThroughput}$

   (b) **Axis:** Z, Design choice for LZW hash table unit
   Challenge: Allow for efficient access of code-table for LZW step while fitting within hardware specifications
   Opportunity: Use trees or associative memories (or both) to allow for low cycle count for finding relevant table entry

Continuum: $Z \in$ {Tree with Dense RAM, Tree with Fully Associative Memory, Tree with Tree, Tree with Hybrid}
Equation for Benefit: Slide 65 from Day 17 has the relevant tradeoff chart, with implied `implementation_complexity` parameter to consider.

(c) **Axis:** II_L, Pipelining II for LZW hardware implementation
Challenge: Allow for quick compression algorithm
Opportunity: Loosen pipelining constraints for LZW to reduce computational load
Continuum: 1 to `MAX_CHUNK_SIZE`
Equation for Benefit: $\texttt{Throughput(LZW)} = \frac{1byte}{II_L}$

(d) **Axis:** W_L, LZW compression window size
Challenge: Cut down on LZW memory requirements
Opportunity: Restructure how encoding/decoding interprets data to reduce conceptual table depth from `MAX_CHUNK_SIZE` rows down to some smaller W_L
Continuum: `MAX_CHUNK_SIZE` to 1 (the latter of which would make it stop being compression)
Equation for Benefit: $\texttt{memRequirements}_{LZW}* = \frac{W_L}{\texttt{MAX\_CHUNK\_SIZE}}$
Note: there are a number of things this change would affect, which is also highly dependent on Z (defined above). We will likely not change this, but it is a parameter that could be tuned.

(e) **Axis:** D_L, Pipeline depth into LZW implementation
Challenge: Reduce chances of idle LZW unit
Opportunity: Lengthen pipeline depth so that variable chunk size does not prevent parallel execution and effective pipelining between CDC and LZW
Continuum: 1 byte to 1MB
Equation for Benefit: Likely complex and related to a lot of interlocking features (no equation provided)

(f) **Axis:** HM_L, Hash Method for LZW string storage
Challenge: Efficient and effective storage of code strings in LZW calculation
Opportunity: Find a collective hash function that allows for byte-wise calculation of string hash as string grows
Continuum: XOR'ing bits, modular addition, any of a variety of other hash methods
Equation for Benefit: Varies by method; in general, will need to evaluate computational complexity for hash-uniformity against penalties for hash collisions. Analysis would likely be better served in an empirical, rather than theoretical, space.

(g) **Axis:** MCS_L, Max Chunk Size for LZW
Challenge: Efficient and effective storage of code strings in LZW calculation
Opportunity: Reducing maximum chunk size across program to allow for fully-in-BRAM LZW table memory
Continuum: Anywhere from 256 bytes (not recommended, likely) to the class-recommended $8kB$
Equation for Benefit: Memory requirements likely grow along the scale of $O\left(MCS_L \log(MCS_L)\right)$, given that we are storing up to MCS_L values of size $\log(MCS_L)$

4. **Interfacing:**

(a) **Axis:** N, Number of bytes at a time transfered to CDC unit
Challenge: Balance memory transfer overhead against memory storage for incoming data
Opportunity: Loosen pipelining constraints for LZW to reduce computational loa
Continuum: 1 to $1MB$ (this could be a fake limit)
Equation for Benefit: $\texttt{memTransferTime}_{total} = \frac{\texttt{totalInput}}{N} * (\texttt{memTransferOverhead} + N * \texttt{memTransferRate})$

(b) **Axis:** O Memory requirement for sending data from CDC to SHA and LZW
Challenge: Reduce the amount of memory required to save chunk data after CDC to be input into SHA and LZW which could be 8KB in the worst case.
Opportunity: Stream the chunk data from CDC into LZW and SHA
Continuum: memory requirement for streaming data could vary from using just 1 byte to a few bytes of FIFO.
Equation for Benefit: Saves memory from using $MAX_CHUNK(8KB)$ amount of memory to just a few bytes

(c) **Axis:** P, Granularity of data that can be sent from one unit to the next in the application flow
Challenge: Reduce the time spent on waiting for input until the previous stage is done computing its output
Opportunity: Use the DATAFLOW pragma to make use of the input data as soon as it is available from the previous stage
Continuum: Could stream one stage's output to next stage's input with a granularity of chunk size to sending data as soon as 1 byte is available
Equation for Benefit: saves on time spent in waiting for the input data to be available as now it can start operating as soon as 1 byte of input data is available

(d) **Axis:** H, Number of bits in hash of SHA value for storing SHA values
Challenge: Effectively storing mapping between SHA values of previous chunks and the chunk index
Opportunity: Tune hash table size to reduce conflicts but also remain compact
Continuum: Could be any small number of bits (call it 5 as a low value) through 256 for the full SHA value.
Equation for Benefit:

$$\texttt{numRows } C = 2^H$$

$$\texttt{probCollision} = \binom{N}{m} \left(\frac{1}{C}\right)^m \left(1 - \frac{1}{C}\right)^{N-m}$$

2 Placeholder Refinement

All the components are fully functional and produce the correct output when run on arm core.

3 Zip file submission

Zip file of fully functional code submitted.

4 Binaries

Zip file of binaries submitted.

5 Documentation

i. All the source codes have been documented and the references have been documented.
ii. **Compression ratio:** For a file of 50KB, the compressed output was 24KB, which is a compression ratio of 0.449.
Breakdown: LZW compressed 34.5KB of data into 24KB of data, which is a compression of 10.5KB. Chunking compressed 15.5KB of data into 36B. Which is a data compression of 15.5KB Therefore LZW contribution is $10.5/26 = 0.403$ and 0.597 from chunking deduplication.
iii. **Overall throughput:** To process an average chunk size of 1765 bytes, our application takes 24.8M cycles. This generates a throughput of 47Kbps.
iv. **Validation:** Validated with the Decoder output as the gold standard. If the diff between the Decoder output of the compress.dat and ucomp.txt are the same, it shows that the algorithm is working. SHA was validated by running the same chunks across SHA multiple times and checking if the output produced is the same everytime. Chunk was tested by adding a byte in the initial chunk, which should cause all other chunks to be the same except for the first.
v. **Contribution:** Nishanth has taken the lead on the CDC section, Ritika has led the progress for SHA, and Taylor has taken on the LZW portion of the project so far. The overall design and interfacing came from a collective meeting towards the top of the process, with incremental changes happening in subsequent meetings.

6 Challenges in collboration and integration

With at least one team member working remotely, collaboration across distance can be tricky. We have a slack channel that helps with overall communication, but there's no great substitute for on-site collaboration. The integration of all the 3 components was a little tricky because it involved communication between these components and making sure that a component produces the output in the same format / size the next component is expecting its input as.
Perhaps the biggest obstacle to integration and implementation this week was the workload of developing 20-30 design space axes, as opposed to a world where we could really start putting our actual hardware implementation together. That said, we ended last week with our single-core CPU code close to working, so the implementation changes we've needed to enact were, fortunately, not too heavy of a load. But one challenge that we faced while integrating last week was that for the first milestone, some of us worked on running the individual algos on out own x86 PCs, which at the end was a little more time taking to port it to arm. So, we plan to run our codes on HLS directly to save time.