

# ESE532 Project P2 Report

Ritika Gupta, Taylor Nelms, and Nishanth Shyamkumar

November 7, 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:

$N = \text{no.of vector lanes, then } N * \text{computation on a single lane}$

- (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,  $N_x$  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.

**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

**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 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. 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.

**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. 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.

**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.

## 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:**  $\text{Throughput}(S) = S * \text{singleSHAUnitThroughput}$

- (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:**  $\text{Throughput}(S) = K * \text{singleSHASubUnitThroughput}$

- (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 simultaneously.

**Equation for Benefit:**  $\text{Throughput}(S) = P * \text{singleChunkMemReadTime}$

- (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:**  $\text{Throughput}(S) = M * \text{singleMemRead/Write}$

- (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 pipeline 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:**  $\text{Throughput}(L) = S * \text{singleLZWUnitThroughput}$

- (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:**  $Z \in \{\text{Tree with Dense RAM, Tree with Fully Associative Memory, Tree with Tree, Tree with Hybrid}\}$

**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:**  $\text{Throughput}(\text{LZW}) = \frac{1\text{byte}}{II_L}$

- (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:**  $\text{memRequirements}_{\text{LZW}} * = \frac{W_L}{\text{MAX\_CHUNK\_SIZE}}$

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.

#### 4. Interfacing:

- (a) **Axis:**  $N$ , Number of bytes at a time transferred to CDC unit

**Challenge:** Balance memory transfer overhead against memory storage for incoming data

**Opportunity:** Loosen pipelining constraints for LZW to reduce computational load

**Continuum:** 1 to 1MB (this could be a fake limit)

**Equation for Benefit:**  $\text{memTransferTime}_{total} = \frac{\text{totalInput}}{N} * (\text{memTransferOverhead} + N * \text{memTransferRate})$

- (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:**

$$\begin{aligned} \text{numRows } C &= 2^H \\ \text{probCollision} &= \binom{N}{m} \left(\frac{1}{C}\right)^m \left(1 - \frac{1}{C}\right)^{N-m} \end{aligned}$$

## 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 Documentatiion

- i. All the source codes have been documented and the references have been documented.

- ii. **Compression ratio:**  
**Breakdown:**
- iii. **Overall throughput:**
- iv. **Validation:**
- v. **Contribution:**

## **6 Challenges in collaboration and integration**