ESE5320: Final Report

Vishnu Venkatesh
Tarush Sharma
Pratyush Mallick

Contents

1 Single ARM processor mapped design

1.1 Overview

Our single processor application design follows this structure.

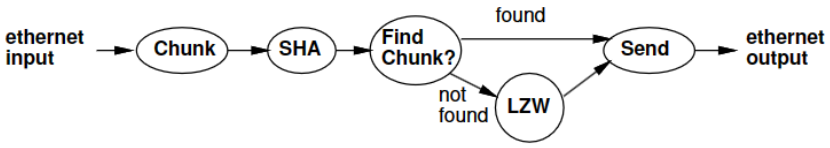

Figure 1: Program pipeline

The chunking algorithm extracted a chunk, and then checked to see if an identical chunk has been encountered before. If not, then the chunk was compressed using the Lempel-Ziv-Welch (LZW) Algorithm and packed into a smaller number of bits, then stored into the encoded file. If the chunk had been encountered before, a reference to the originally compressed chunk is passed instead.

1.2 Content defined chunking

The first operation that is performed is Content Defined Chunking (CDC). We used a rolling hash, computed over a window of 16 values as `input[i] *` 3^i. The powers of 3 were precomputed and stored in an array for performance improvement, and a new hash is computed for each new character as:

$$NewHash = (OldHash * 3) - (input[i] * 3^{16}) + (input[i + 16] * 3)$$

Whenever this hash value was divisible by a predefined value (called MODULUS), it was considered to be new chunk starting point. The CDC loop operates over all characters in a packet. The output was a vector containing indices that marked the start of each new chunk. Once the indices of all chunks in the packet were obtained, a SHA-256 algorithm was run on each chunk.

1.3 SHA-256

SHA-256 is a standard algorithm to compute cryptographic hashes. For every chunk in the indices vector, the function takes input from the input buffer and computes the hash and stores it in a table.

1.4 Deduplication

Once all values had been stored in the table, the deduplication algorithm runs for each hash and compared it against previous hash values. Since cryptographic hashes are designed to minimize collisions, we make the assumption that identical hash values imply identical chunks. If the chunk has been seen before, then the ID of the old chunk is stored in place of the new one.

1.5 LZW encoding

The Lempel-Ziv-Welch algorithm is a standard algorithm that we took from one of the many online references. The LZW implementation uses this hybrid lookup method to find duplicate prefixes. For each prefix code + next character combination, a lookup is performed and then if the value is found, we take that value for the next iteration. If not, we store the code in the hybrid memory, perform a bit-packing operation on that prefix code, and write it to output.

The program branches depending on whether the chunk has been encountered previously. If yes, then a 32-bit header is written to a buffer that holds the ID of the chunk. If no, then LZW compression is called on that chunk to further compress the new chunk and then a header that contains the length of the LZW encoded chunk as well as the encoded chunk is written to the buffer. The LZW encoded output is further packed into an array of 8-bit values - each code is originally 12 bits, so this cuts down the size by 33%.

1.6 Key parameters

- Packet size: 8192 bytes

- Average Chunk Size: 4096 bytes
- Max Chunk Size: 8192 bytes
- Min Chunk size: 0 bytes
- CDC chunking method: Rabin fingerprint
- Size of hash table in LZW: 65536
- Associative Memory size: 64

1.7 Performance achieved

The throughput for the software version was about 12 Mbps, as calculated for the LittlePrince.txt test case in milestone 2. However, this value varies considerably for the test case depending on the amount of redundancy in the chunks.

1.8 Compression achieved

The compression ratio varied between 30% (for LittlePrince.txt) to 81% (for Franklin.txt). The .tar files were somewhere in between. These two files approximately model our best and worst case scenarios, because the LittlePrince.txt has a large amount of intentionally repeated text and therefore many chunks that are simply represented by a 4 byte header. The Franklin.txt test case however has very few repeated chunks due to the lack of redundancy in the data stream.

1.9 Characterization and breakdown of time spent in major components

Though this was designed to handle an input stream of data, the single processor version was designed to operate sequentially over a single packet. Therefore we could afford to provide several sequences of loops, each one operating at the level of an entire packet (i.e, group of chunks) instead of one chunk at a time, despite the latter giving us some parallelization

opportunities. We realized that we would have to change the framework and algorithm itself to adapt it to an Ultra96.

The breakdown of time for major components is given below, from our milestone 3 report.

```
root@u96v2-sbc-base-2020-2:/media/sd-mmcblk0p1# ./host_fpga hardware_encoding.xclbin
Main program start
INFO: Found Xilinx Platform
INFO: Loading 'hardware_encoding.xclbin'
setting up sever..
14322
table constructed of 14 packets and 14322 bytes
server setup complete!
14

 Executign time of Kernel (Using Profiler) 0.20554ms
--------------- Key Throughputs ---------------
Ethernet Throughput:  389.47 Mb/s.       Latency:0.02128 s.
Cdc Throughput: 309.926 Mb/s.    Latency: 0.37005 s.
LZW FPGA Throughput: 56.6762 Mb/s.       Latency: 46.542 s.
SHA Throughput: 203.488 Mb/s.    Latency: 0.56361 s.
Dedup Throughput: 6450.39 Mb/s.  Latency: 0.01778 s.
```

Figure 2: Breakdown of time in major components in single processor version

2 Ultra96 mapped design

2.1 Key parameters

- Packet size: 8192 bytes
- Average Chunk Size: 2048 bytes
- Max Chunk Size: 4096 bytes
- Min Chunk size: 100 bytes
- CDC chunking method: Rabin fingerprint
- Size of hash table in LZW: 8192×2 (two buckets for each hash key)
- Associative Memory size: 64

2.2 Overview

The CDC, SHA, and Deduplication were mostly the same as the single processor implementation with one important upgrade – to allow for pipelining and unrolling, the CDC, SHA, and deduplication would run immediately if a chunk was found rather than buffering information from a group of chunks and performing each component operation at once. This also did away with many of the intermediate buffers that were used to shuffle information between the different modules, which eliminated some overhead. As soon as a chunk was found, SHA would be called to compute the hash and then deduplication would check if the same chunk has been encountered before – if yes, then the ID of the new chunk was changed to the old one. If not, LZW is immediately invoked. Nothing changed besides the program flow and the size of the interfaces, the type of interfaces and core algorithm of each module remained the same.

The LZW encoding turned out to be the major bottleneck for this algorithm. So that function was chosen for acceleration on the Zynq FPGA. There are two critical differences between the hardware implementation and the golden version used for validation:

- The golden implementation (for C++) used dynamically allocated objects, namely the `std::string` and `std::vector` objects. Since this is unsuitable for mapping functionality to hardware, we pre-allocated fixed size `unsigned char` and `int` arrays that were the same size as the input buffer (because that was the theoretical upper bound) and passed the size (i.e, number of useful data bytes) as independent parameters.
- The golden implementation also used `std::unordered_map`, which is another dynamically allocated object. However, instead of using a hardware appropriate implementation of a custom hash table, we used a hybrid memory model. The hash table was the first level, implemented with the Murmur3 hash, and it had two "buckets" to deal with hash collisions. However, in addition to this, there is also a second level associative memory that takes in a key value that acts as an index to another key value, which is configured by the program so that these second set of keys sequentially index yet another block of memory allocated to store the values to be accessed finally. Though

this is expensive in terms of hardware area, the lookup speed is much faster than a conventional search.

2.3 Performance achieved

The final throughput we achieved was 23.4675 Mbps.

```
EXE: /media/sd-mmcblk0p1/host_fpga
[XRT] WARNING: unaligned host pointer '0xfffffba8f83c' detected, this leads to e
--------------- Key Latency ---------------
Total latency of Packet Collection: 3698.01 ms.
Total latency of CDC + SHA + Dedup + LZ[  688.781833] [drm] bitstream 8b2addc7-c
W: 4.54338 ms.
Total latency of CDC: 0.53496 ms.
Total latency[  688.787769] [drm] Pid 1613 closed device
 of SHA: 0.10485 ms.
Total latency of Dedup: 0.01839 ms.
Total latency of LZW HW: 3.88518 ms.
Total time taken: 3702.89 ms.
-----------------------------------------------------------------
Average Latency of Packet Collection: 1849 ms.
Average latency of CDC + SHA + Dedup + LZW: 2.27169 ms.
Average latency of CDC: 1.61389 ms.
Average latency of SHA: 0.0087375 ms.
Average latency of Dedup: 0.0015325 ms.
Average latency of LZW HW: 0.64753 ms.
Average time taken: 3702.89 ms.
--------------- Data Processed ---------------
Total Compressed Data: 5745 bytes
Total Recieved Data: 14322 bytes
Input Throughput Encoder: 23.4675 Mb/s. (Latency: 0.00488232s).
```

Figure 3: Latency measurements for FPGA implementation

The kernel running duration was 3.57ms.

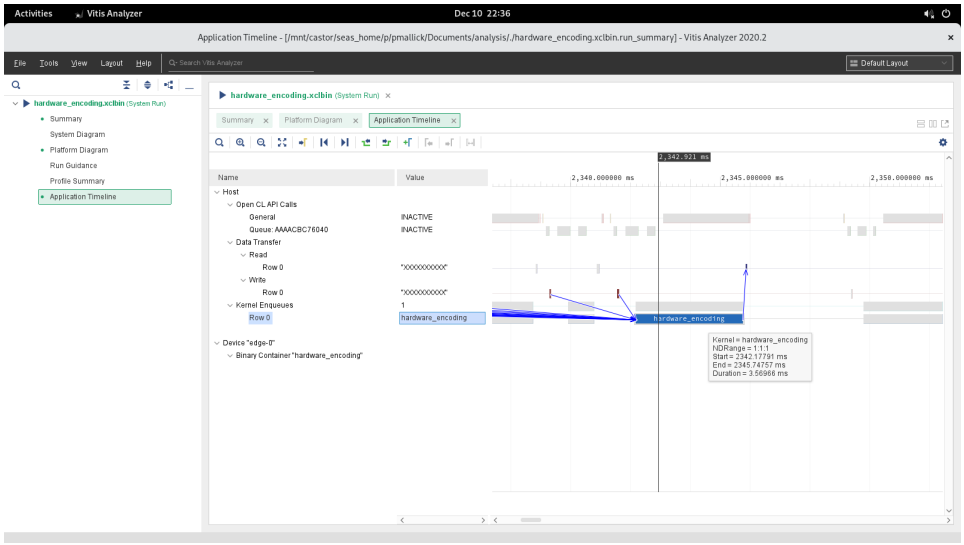

Figure 4: Application timeline for hardware_encoding kernel

The reading and writing times between host memory and FPGA memory is quite low because we implemented dataflow and FIFO inputs and outputs.

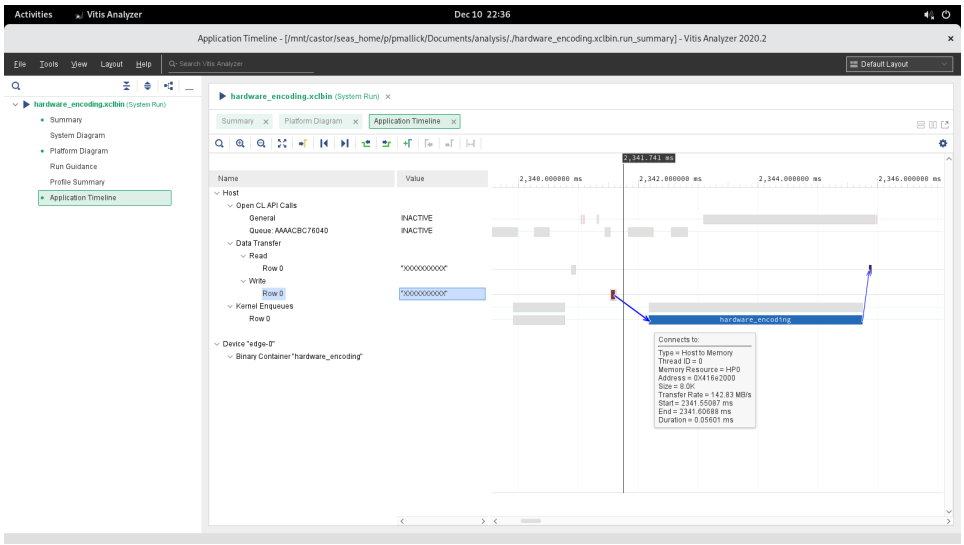

Figure 5: Timeline for reading from host memory

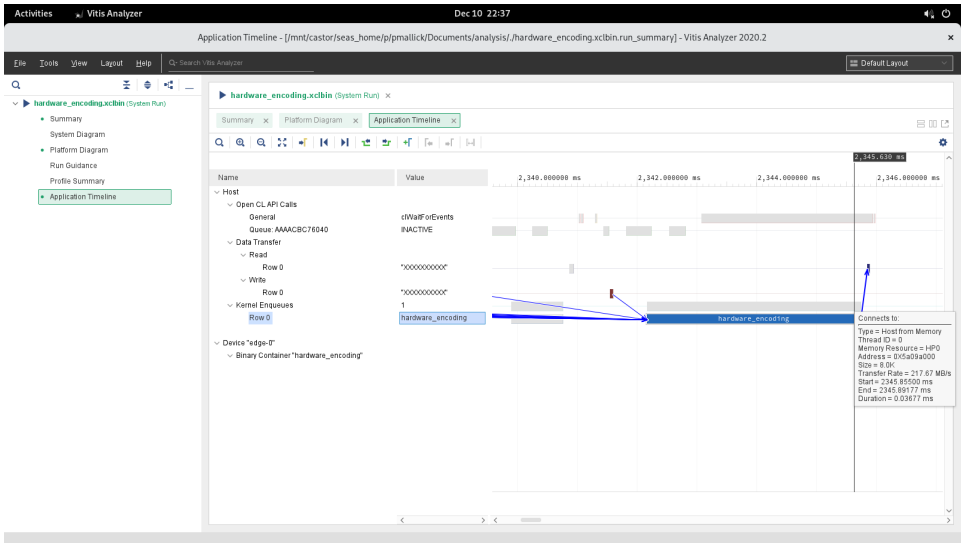

Figure 6: Timeline for writing to host memory

2.4 Mapping to different resources of Zynq

The Etherent packet fetching, CDC, and deduplication components ran on the ARM processor. SHA256 ran on the NEON processor, and LZW encoding ran on the FPGA.

The Lempel-Ziv-Welch algorithm is a lossless compression technique that uses redundancy in the input chunk to group character sequences that have been encountered before, mapping them to code numbers that form the output of the compression algorithm. For this implementation, it turned out to be the bottleneck of our application - so it was the obvious choice for FPGA acceleration.

The code running on the ARM host processor uses OpenCL interfaces to interact with the FPGA and transfer input and output data. The host code uses an in-order command queue and `cl::Buffer` type objects to transfer data to the FPGA. The two `std::vector` objects and one `int` pointer are allocated using C++ template allocators that ensure allocation according to memory alignment boundaries, which makes it easier when transferring data to custom hardware. We have implemented multithreading on the ARM processor discussed in more depth below.

2.5 Compression achieved

The compression ratio varied between 30% (for LittlePrince.txt) to 81% (for Franklin.txt), exactly the same as in the single processor implementation. The .tar files were somewhere in between. These two files approximately model our best and worst case scenarios, because the LittlePrince.txt has a large amount of intentionally repeated text and therefore many chunks that are simply represented by a 4 byte header. The Franklin.txt test case however has very few repeated chunks due to the lack of redundancy in the data stream.

2.6 Task decomposition

2.7 Parallelism

2.8 Multithreading

We implemented two threads for the processes running on the ARM processor. The first thread is responsible for reading data from the ethernet stream. The second thread runs the other parts of the code in parallel. This design was chosen because we observed that ethernet data transfer took a large amount of time. The threads coordinate with each other to fill the buffer (ethernet data streaming thread) and empty it (data processing thread), with a mutex-protected flag to indicate the state of the buffer.

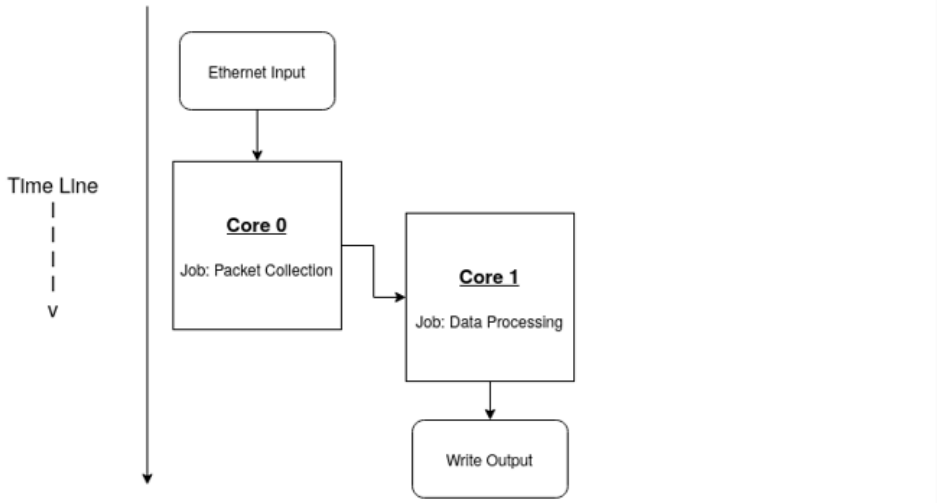

Figure 7: Simplified view of multithreading algorithm

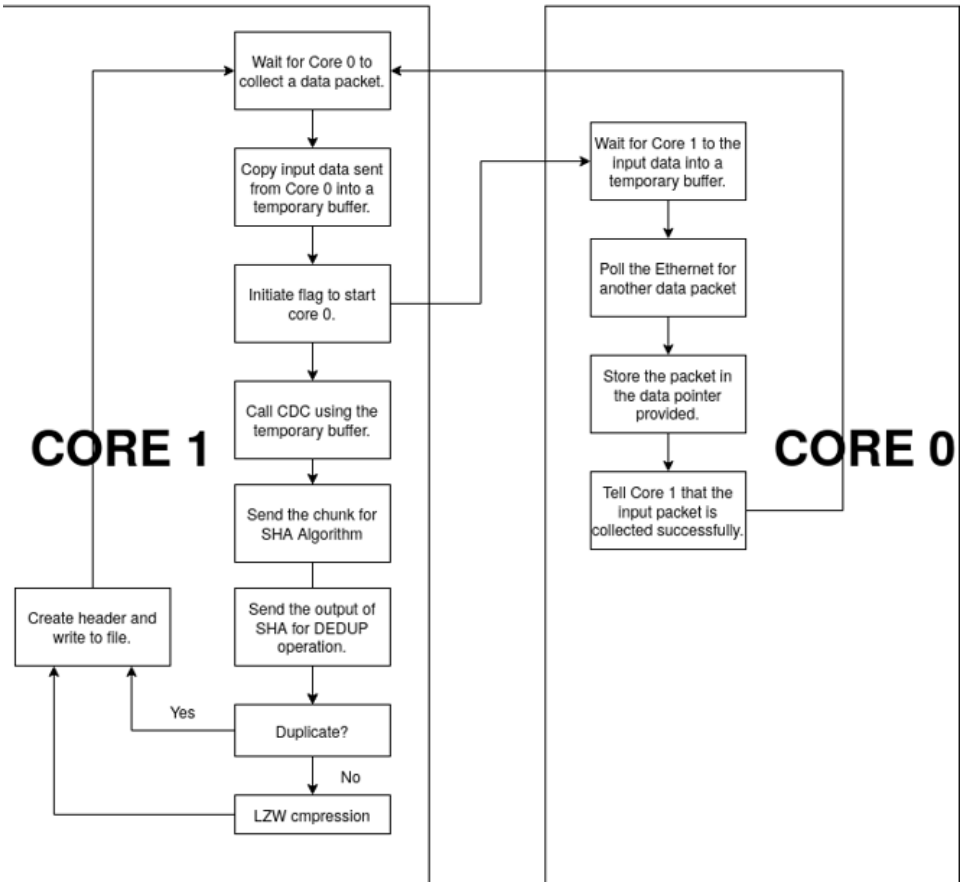

Figure 8: Detailed view of multithreading algorithm

2.9 Miscellaneous Optimizations

- The LZW kernel is mapped to the FPGA. It takes in a character array (the chunk to be compressed) and the size of that array. The outputs are a bitstream representing the bit-packed version of the output LZW codes, as well as the output size. These inputs and outputs are mapped to specific AXI high performance ports (HP0, HP1, HP2). We have an internal hash table of size 8192x2 - we found that a large hash table couldn't be implemented in the available BRAM, and we used 2 buckets to deal with hash collisions. The output buffer was allocated as an array of the same size of the input value, and the number of bytes

the output actually was would be passed as a separate parameter – the only reason for this was that the size of the input is the theoretical upper limit.

- The associative memory was initially implemented as a structure with the high, middle, and low key arrays. However, we found that implementing them separately allowed for parallel access and improved the read/write times to associative memory. In addition to that, the inputs and outputs were partitioned as well. We created two local buffers and implemented a load-store-compute mechanism to handle data transfer to and from the FPGA.

- The bitpacking was implemented for each character added to the prefix code. Initially the bitpacking loop ran after the entire LZW encoded string was collected into an intermediate buffer; however, this gave us an II of greater than 1. We implemented a second version of the bitpacking algorithm did away with the intermediate buffer and directly wrote the bitpacked output code to the output array – this gave us an II = 1.

- We mapped our input and output buffers to arbitrary precision FIFOs to mimic a streaming effect. Going by the documentation on array interfaces, this can be implemented if the array accesses are sequential which it is in this case, and so we can use this to emulate streaming and improve performance.

- The initial loop that resets the hash table seemed to take a huge amount of time - almost two orders of magnitude above the time taken for the actual LZW operations. Unrolling this reduced the time by one order of magnitude.

- We set the burst size to 8K bytes, which defines the amount of data that can be sent together to the FIFO inputs and outputs. We ensure that on the host side, alignment of the `cl::Buffer` objects is done to improve performance. This improves performance because

```
std::vector<unsigned char, aligned_allocator<unsigned char>>output_hw(PACKET_SIZE);
int out_hw_size __attribute__((aligned(sizeof(int))));
std::vector<unsigned char, aligned_allocator<unsigned char>>chunk_arr(PACKET_SIZE);
```

Figure 9: Aligned allocation templates for hardware transfer

```
chunk_arr_cl =    cl::Buffer(context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY, sizeof(unsigned char) * PACKET_SIZE, chunk_arr.data(), &err);
output_hw_cl =    cl::Buffer(context, CL_MEM_USE_HOST_PTR  | CL_MEM_WRITE_ONLY, sizeof(unsigned char) * PACKET_SIZE, output_hw.data(), &err);
out_hw_size_cl = cl::Buffer(context, CL_MEM_USE_HOST_PTR | CL_MEM_WRITE_ONLY, sizeof(int) * 1, &out_hw_size, &err);
```

Figure 10: Ensuring that CL buffers are read/write

2.10 Power consumed

The on-chip power consumption of this implementation comes out to be 2.519W.

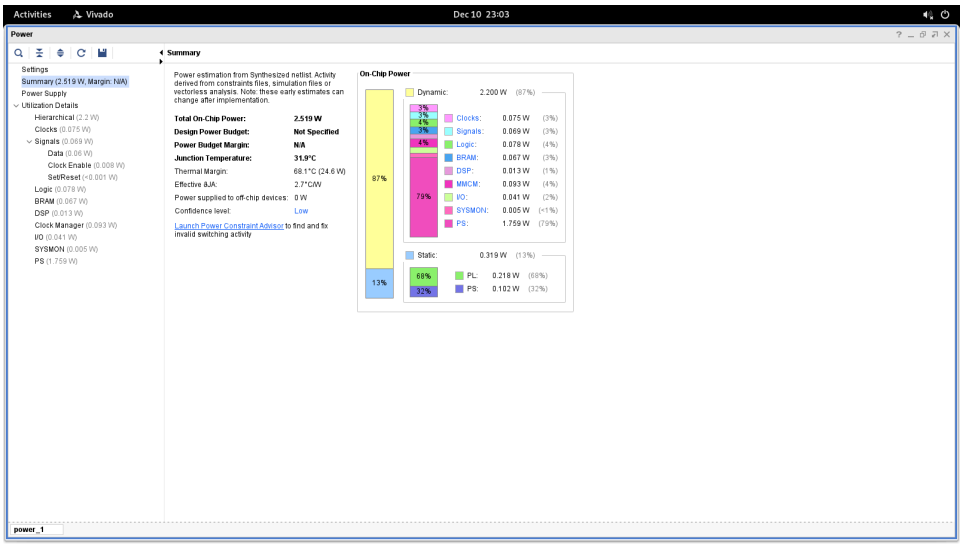

Figure 11: On-chip Power Consumption

The power consumption for BRAMs is 0.067W.

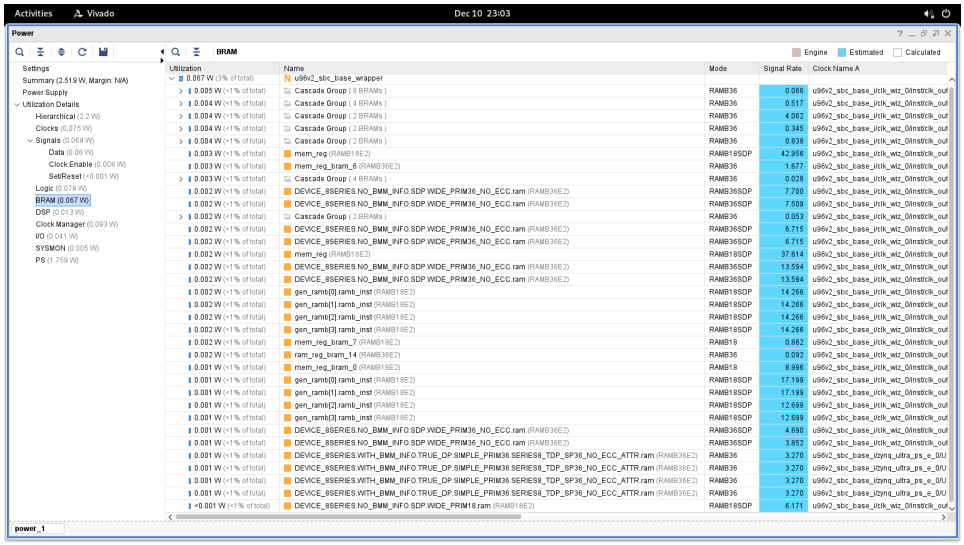

Figure 12: BRAM Power Consumption

2.11 Current bottleneck

Despite our efforts, LZW continues to be a bottleneck as the slowest operation in the pipeline. In the next section, we outline some optimizations that we had planned to work on but could not in the given timeframe.

2.12 Possible future optimizations

- Larger chunk sizes could theoretically improve the computation-to-overhead ratio in LZW implementation on the FPGA. However, this would lead to further hash collision issues that we were already facing at this chunk size – perhaps we could expand the hash table or associative memory, or rewrite the hashing algorithm to make it less sparse in filling out the hash table.
- Packing multiple chunks at once and sending it to the kernel would also improve this ratio. However, that would require us to change the pipelined framework that we had developed in a non-trivial way.
- Implementing multiple kernels would also allow for more parallelization, however it suffers from the same issue mentioned above of mod-

ifying the pipelined system framework. It would also require some effort to synchronize the activity of these two compute units so they write data to the output stream in the correct order.

- We considered using free running kernels to minimize data transfer between host and FPGA. Free running kernels reduce the overhead of re-initializing the FPGA kernels every time we want to perform an LZW operation, since we noticed that `cl::commandEnqueueTask` and `cl::memMigrateObjects` took a significant amount of time in the application timeline.

- Increasing the level of multithreading. Though we used two threads, we could possibly add more to handle writing to the output stream, and perhaps others within the main pipeline as well. This requires a much more careful analysis of data dependencies and drastically increases program complexity in coordinating the multiple threads.

- Removing branching from LZW implementation. This makes pipelining difficult, as it forces sequentialization, but we couldn't come up with a way to avoid it for the bitpacking implementation.

- Improving the prefix code lookup speeds in the table could also lead to improvement – despite the fact that it was highly optimized for speed at the expense of hardware resources, it is still somewhat of a bottleneck and we couldn't think of better ways to do it.

- There are some parts of the framework that have to be written in a blocking way, due to data dependencies. But there are also some parts that are blocking when they strictly do not need to be, and were only written that way as an initial implementation. For example, we need not wait for the LZW encoding to return data before starting the CDC, SHA, and deduplication process for the next chunk. However, implementing this is tricky, because it could lead to problems where there was too much data input (that is, input to LZW encoding kernel) backed up and possible risks of losing data or overwriting previously buffered data.

3 Validation

3.1 Unit testing

During the development process, each component was developed by first specifying the interfaces and implementing placeholder code to achieve that minimum functionality. That validation was a straightforward manual examination of the outputs and ensuring that the pattern of outputs was according to the interfaces specified. Then we rewrote the components individually to expand functionality as required.

The input of the CDC algorithm is the packet itself. The algorithm iterates over all characters, maintaining a rolling window for which a hash is computed. The output of the content defined chunking algorithm is a set of indices to mark the first character of the new chunk. The input of the SHA256 algorithm is a pointer that points to the input packet, except that it starts from the index found by CDC and terminates at the end of the chunk. The output of the SHA256 algorithm is a table that contains all the IDs, hash values, and seen indicators for a particular chunk. The input to the deduplication stage is this table, where the previously added chunk is compared against all previous ones to look for a match. The output is a modified version of that table with chunk IDs changed if the chunk has been encountered before. The input to the LZW compression, which is called if the chunk hasn't been encountered before, is the chunk itself. The output is the bitpacked version of the output code. Bitpacking is done within the LZW function itself so that it can be accelerated on the Zynq processor.

3.2 Content Defined Chunking

The validation for content defined chunking was broken down to meet the following requirements:

- The index output shall be the first value of each new chunk.
- There shall be no overlaps between chunks.
- The average chunk size is 2048 bytes.

3.3 SHA-256

The validation criterion for this was that different chunks shall be given different hash values. We validated by logging both the chunks themselves and the corresponding hash values in a file, and ensuring that all duplicates were given the same hash values. However, since this is a standard algorithm, we did not have to do much to ensure that it was working correctly.

3.4 Deduplication

The validation for deduplication involved examining the state of the SHA256 table before and after the deduplication operation. Since the table also stored hash values, it would be evident if there was a duplicate hash value that wasn't caught. We also created custom input with multiple repetitions – for example, we duplicated a paragraph of the LittlePrince.txt test case 10 times and then looked for a sequence of chunk IDs that was repeated 10 times to verify that the repeated sequences had the same chunk IDs as the original chunks.

3.5 LZW compression

The initial validation of the LZW compression algorithm was done against the golden version taken from an online source. We compared each output value of our algorithm to the output value of the golden version until they were similar. For the bitpacked version, we used the decoder and ran the diff command on a linux shell to verify that the output matched the source input.

4 Key lessons

The key lessons from this project were all about managing complex tasks effectively. We learned the importance of breaking it into smaller, more manageable parts. The first step is to clearly define the interfaces that our components would have – the inputs and outputs. After those are established, the next crucial step was to make a basic version. It didn't have

to be functional, but it had to follow interface patterns that had been defined earlier. This minimal version acted as a starting point that we could incrementally improve module-by-module and make better until it worked completely.

Also, the project taught us a lot about the challenges of putting together different parts in bigger software projects. Even though each part was ready early on, putting them all together had its own set of challenges. We realized we made a mistake early in the project. We thought it would be easy to make each part (CDC, SHA, Deduplication, LZW) work on entire files and then smoothly switch to a streaming setup. But that turned out to be quite difficult. Unexpected problems came up because we didn't think about some extra constraints that come with making a program run with input data being streamed. This realization led us to a big task – we had to redo almost the whole code to deal with these unexpected issues and make the overall project structure better. It also slowed us down and prevented us from having the time to explore more hardware optimizations later on into the project.

We also learned a lot about how to write code while thinking about performance. We learned how to think in greater levels of details – thinking about all our buffers, loops, and wondering whether some operations were necessary. We improved the II of LZW encoding by realizing we didn't need an intermediate buffer to hold the outputs, and directly performed a bitpacking operation for each output prefix code before writing it to the output buffer. We thought of storing values that were constantly recomputed in local memory, as well as thinking about how we can make loop bounds deterministic so that they can be better implemented by hardware. All in all, this project helped refine our thought process for writing low-level code that runs in a way that is suitable for direct hardware implementation, as well as improving our overall debugging skills and problem-solving ability.

5 Design Space Exploration

6 Who did what

The development of the baseline version was a collaborative effort involving all three. The initial phase of the project was characterized by a series of brainstorming sessions, during which the team exchanged ideas and provided feedback. The focus was on exploring different implementation options, evaluating their advantages and disadvantages, and planning the necessary interfaces. Subsequently, a prototype version was created, featuring the correct interfaces but with a basic and somewhat inaccurate implementation (for example, the initial placeholder for LZW compression was an identity operation that returned the same output as the input).

Within the team, specific responsibilities were divided. Vishnu and Tarush took charge of the core algorithm running on the host processor and the software for the LZW compression technique with the hybrid memory model. Pratyush was primarily responsible for the FPGA implementation and hardware optimization. In the initial phases, the team wrote and tested the code on their personal computers, leading to several iterations and modifications. The original code operated on entire files, and improvements were gradually introduced over time. Additionally, intermittent testing on the FPGA was conducted to ensure the functionality of the code.

Tarush and Pratyush were also involved in the development of the Open Computing Language (OpenCL) code. Vishnu dedicated a significant amount of time to software debugging, rewriting or revising parts of the project to align with the current implementation. This was a challenge because data corruption was a frequent issue, and sometimes caused by something difficult to spot that might have root cause as hidden as a `<=` instead of a `<` in a loop comparison, or an unintentional implicit typecast when passing a value to a function. As a result, fine-tuning the implementation was a constant effort every time a new feature was added. Vishnu and Tarush also managed to implement multithreading on the host processor, having ethernet data streaming on one thread and the remaining CDC, SHA and Deduplication on the other thread. The second thread was also responsible for queueing the hardware kernel and triggering data transfers, and it was a challenge to synchronize the work of these two threads.

Overall, the collaborative efforts of Vishnu, Tarush, and Pratyush in the initial stages involved rigorous planning, testing, and debugging, with each team member contributing to different aspects of the project, from algorithms and software to hardware implementation and optimization. Each member made sure to learn from teammates along every step of the way, not only so that their contributions were usable by other team members but also to ensure that each person would understand the parts they didn't work on – to the degree that they could recreate it again if they had to, alone.

7 Academic Integrity

We, Vishnu Venkatesh, Tarush Sharma, and Pratyush Mallick certify that we have complied with the University of Pennsylvania's Code of Academic Integrity in completing this final exercise.
