ESE532 Project P3 Report - Group

Ritika Gupta, Taylor Nelms, and Nishanth Shyamkumar

November 15, 2019

1 I/O

We were unable to fully get a working integration of I/O functionality together within the alotted time. Cryptic errors and long build times exacerbated the problem. As such, we have no idea what the current throughput we are achieving is. That said, large portions of our code **are** coming together, so that's encouraging.

2 FPGA Acceleration

We were able to accelerate some functions, but unable to sensibly measure throughput in time. Further development should fix this.

2.1 Rabin

We attempted to put the Rabin fingerprinting onto hardware in this iteration, but ran into issues and decided to stick with a software implementation instead.

2.2 SHA

We have not yet moved this step onto the FPGA.

2.3 Deduplication

We have not yet moved this step onto the FPGA. Our current development version plans to hash the SHA fingerprint to a smaller number of bits (we're currently looking at 16), and divide a portion of DRAM into "buckets" for it to use. For all of the roughly 2^{17} potential "rows" that pair a SHA256 digest with an "output chunk" index, the hashed digest will point to some section of them. We're currently leaning towards a potential hash depth of 2 (allowing for a factor-of-two hash overflow), which would point to needing to look at four potential digest-index pairs for a digest match.

In this way, for each digest that comes in, we will need to pull 144 bytes from DRAM to check against our incoming digest. Ideally, this kind of shared memory load is feasible to enact in a reasonable amount of time; once our implementation becomes sensibly integrated, we will know bette

2.4 LZW Compression

We were able to put the LZW compression algorithm entirely onto the FPGA. For development, this involved making a wrapper function in hardware that took in a buffer filled with the chunk to compress, fed that into a stream (to simulate what would come out of the other hardware components), read from that stream to compress the contents, and fed that output stream into a different buffer to return back to the CPU. (The connections between them were acheived via the `dataflow` pragma.)

Validation was achieved against the working CPU version of the code we completed previously. There was significant tuning involved, given the memory limitations of the FPGA; after all, there is not enough onboard BRAM space to hold 2^{21} 13-bit data entries. Instead, we implemented a hash table internally that hashed the addresses of our conceptual LZW-table (21-bit values) into 10-bit values with some slight degree of uniformity (more on that in a second). We elected (arbitrarily) to use 1024 hash table rows, and allow for 24 of the 34-bit key-value pairs to be stored in each row. This allowed for each "hash bucket" to store up to three times what its "fair" share would be.

This hash memory was distributed to enough BRAM's to allow for all the values in a hash row to be read in one operation. For lookups, a key is then checked against each key-value pair to find the corresponding value. This approach allowed

for a functional table implementaton with just 52 BRAM units dedicated to the LZW process (so far).

Notably, hash collisions past our planned depth may occur; as such, we plan to look into implementing a small associative memory to handle these outliers, and see if there is some good tuning that we can do to waste as little memory as possible.

The hash function was developed highly arbitrarily, but from empirical methods. We logged the software-produced LZW key-value pairs offline, and then took those 21-bit key values and attempted to develop a function that would distribute them among 1024 hash buckets reasonably evenly. In the end, the implementation is a combination of a lot of `XOR` operations, with one additional lookup in the Rijndael substitution box, in various mixing of key bits arranged to, hopefully, distribute the higher-entropy bits of the input across the bits of the output. Upon testing this approach with both a text and binary data source, we found that no hash bucket was seeing more than a factor-of-three load more than a completely even share. While this could probably be refined further to be more efficient, the relative efficiency of the current approach is remarkable in its own right.

In terms of clearing the table after each iteration, we are using an array of "valid bits" to keep track of the "validity" of each entry in the hash table; in this fashion, the process of clearing the data should involve significantly fewer operations than trying to set the entirety of the hash table memory.

The primary loop of our hardware function was able to be pipelined down to an `II` of 6; if we were to cut down on the maximum chunk size, we're optimistic that we could reduce that further if need be.

3 Code

Included in different turn-in location.

4 Binaries

Included in different turn-in location.