ESE532 Project P1 Report

Ritika Gupta, Taylor Nelms, and Nishanth Shyamkumar

November 1, 2019

1. Our group makeup is Ritika Gupta, Taylor Nelms, and Nishanth Shyamkumar.

2. (a) We end up with $64ns$ to process each $64b$ word of input, which comes out to 76.8 (so, 76) cycles for a 1.2GHz processor.

   (b) By similar logic as the last question, with a 200MHz clock, we end up with 12.8 (so, 12) cycles to process all of the input.

3. (a) (i) **Content-Defined Chunking**:

```
read input from secondary storage to a local buffer
hash arg is used to update window size chunk and store rabin fingerprint

  while input present:
    fread(buf, buf_len, 1, fp)
    len = buf_len
    while len >= 0:
      generatechunk(hash, buf, len)
      len -= hash.length
      sha256_hash(chunk)

generatechunk(hash, buf,len):
  for b in buf:
    rabin_slide(hash, b)  //Calculates rabin fingerprint on hash.window.

    if(hash.length >= MIN_SIZE && (hash.digest & MASK) == 0 || hash.length >= MAX_SIZE
      chunk is generated

      chunk.start = initial pointer
      chunk.length = hash.length

      reset_hash(hash)
      return

    hash.length++;
```

   (ii) **SHA-256**:

```
# RITIKA'S VERSION
hash[0:7] = initializeHashValues()
k[0:63] = initializeRoundConstants()

for each chunk
        for each 512_bit_subchunk m[]    # 64 byte sunchunk m[]
                for ( ; i < 64; ++i)     # For eevry byte in subchunk
                        m[i] = SIG1(m[i - 2]) + m[i - 7] + SIG0(m[i - 15])
                                              + m[i - 16];

                # call update function
                sha_update(data, len)
```

```
        # Compute hash for last incomplete chunk, if any
        if (last_subchunk < 448 bits)
                append 1
                append 0s to make 448 bits
        else
                append 1
                append 0s to make 512 bits
                sha_update(last_subchunk, 64)
                last_chunk[] = {0}

        append_0s_till_448_bit_subchunk()
        append_msg_len_in_last_64_bits()
        sha_update(last_subchunk)
        convert_hash_values_big_endian()

sha_update(data, len)
        # initialize message schedule m[]
        for (i = 0, j = 0; i < 16; ++i, j += 4)
                m[i] = (data[j] << 24) | (data[j + 1] << 16) | (data[j + 2] << 8) | (
        for ( ; i < 64; ++i)
                m[i] = SIG1(m[i - 2]) + m[i - 7] + SIG0(m[i - 15]) + m[i - 16];

        # initialize working variables with previous hash values
        a = hash[0]
        b = hash[1]
        c = hash[2]
        d = hash[3]
        e = hash[4]
        f = hash[5]
        g = hash[6]
        h = hash[7]

        # update_working_variables()
        for ( ; i < 64; ++i)    # For every byte in subchunk
                t1 = h + EP1(e) + CH(e,f,g) + k[i] + m[i];
                t2 = EP0(a) + MAJ(a,b,c);
                h = g;
                g = f;
                f = e;
                e = d + t1;
                d = c;
                c = b;
                b = a;
                a = t1 + t2;

        # increment hash values by corresponding working variable
        hash[0] += a
        hash[1] += b
        hash[2] += c
        hash[3] += d
        hash[4] += e
        hash[5] += f
        hash[6] += g
        hash[7] += h

digest = hash0 append hash1 append hash2 append hash3 append hash4 append hash5 appen
```

Credit: Wikipedia

(iii) **Chunk Matching**:

```
if shaResult in chunkDictionary:
    send(shaResult)
else:
    send(LZW(rawChunk))
```

(iv) **LZW Encoding**:

```
table = {}
for i in range(256):
    table[i] = i
curPos = 256
STRING = Input.read()
while(True):
    CHAR = Input.read()
    if STRING + CHAR in table.values():
        STRING += CHAR
    else:
        Output.write(table[STRING])
        table[STRING + CHAR] = curPos
        curPos += 1
        STRING = CHAR
    if Input.isDone():
        break
```

Credit: https://www.dspguide.com/ch27/5.htm

(b) **Memory Requirements**

(i) **Content-Defined Chunking**:
We'll need a rolling hash window's worth of working memory, spanning 16ish bytes. Structure to store rabin fingerprint, chunk size that has been slid over. 2 2KB tables that is used by the Rabin algorithm.

(ii) **SHA-256**:
Eight 32-bit span of memory to hold hash values, sixty four 32-bit span of memory for storing message schedule, sixty four 32-bit span of memory for storing round constants, ten 32-bit span of memory for working variables, and 64-byte SHA-block span of memory.

(iii) **Chunk Matching**:
We'll want a table to store hash values for index purposes, which would require at least 8 bytes times the maximum number of chunks to be processed.

(iv) **LZW Encoding**:
This is a somewhat tricky question given the associative memory involved, but it will be on the scale of roughly MAX_CHUNK_SIZE entries times 12 bits.

(c) **Computational Requirements**

(i) **Content-Defined Chunking**:

(ii) **SHA-256**:
Computation work per chunk: (Ignoring the index and loop iterator computations) Prepare the message schedule m[i]: 16 * (3 ORs + 3 Shifts) = 96 operations (64 - 16) * (3 ADDS + 11 for SIG0 + 11 for SIG1) = 1200 operations Update the working variables: 64 * (7 adds + 14 for EP0 + 14 for EP1 + 4 for CH + 5 for MAJ) = 2496 operations Update hash values: 8 adds Total no of operations = 96 + 1200 + 2496 + 8 = 3800 computations

(iii) **Chunk Matching**:
The only real operation here is a dictionary lookup, so computation can sensibly be considered negligible.

(iv) **LZW Encoding**:
With sensible dictionary lookup, there should be on the scale of one comparison operation per incoming byte of input, plus possibly a couple of additions as we loop through the incoming data.

(d) **Memory Access Requirements**

(i) **Content-Defined Chunking**:

(ii) **SHA-256**:
Memory operations per chunk(Ignoring local - BRAM reads and writes): Prepare the message schedule m[i]: 16 * (4 reads) = 64 Initiate the working variables: 8 reads Update hash values : 8 writes Total operations: 64 + 8 + 8 = 80

(iii) **Chunk Matching**:
One dictionary lookup should be required for each incoming hashed value; as such, we're looking at roughly 32 bytes of memory read (for reading the hash), and whatever memory costs are required after that for a dictionary lookup on that value.

(iv) **LZW Encoding**:
With efficient encoding, there should be roughly one memory read and one memory write involved in devising the code for each incoming byte as part of LZW. This will, of course, be very dependent on the specific dictionary implementation.

(e)

4. (a) The **LZW** and **SHA-256** operations can feasibly be done in parallel, as neither depends on the other. Once, SHA256 completes and tells whether the chunk already exists or not, the decision can be made whether to continue with LZW or discard its output.

(b) **Task-Level Parallelism**

(i) **Content-Defined Chunking**:

(ii) **SHA-256**:
Hash computation of each input chunk is independent of each other. But this is limited by the input coming from CDC, which will send input chunk by chunk. For each input chunk, SHA256 works by dividing the input chunk into 512 bit subchunks and padding the last subchunk if it is less than 512. The computation of hash values for each subchunk is dependent on the previous hash computation. So, it inhibits parallelization of computation for each subchunk. Each subchunk has to go sequentially. But it does not need to wait for the entire input chunk to start computation. It can start as soon as it receives first 512 bits because padding only happens in the last subchunk, that's too only if it is less than 512 bits.

(iii) **Chunk Matching**:
There aren't many tasks here, so task-level parallelism seems a rather useless thing to pursue.

(iv) **LZW Encoding**:
The overall task graph for LZW encoding is close enough to linear that there is not much reasonable task-level parallelism to be captured.

(c) **Data-Level Parallelism**

(i) **Content-Defined Chunking**:
Any particular window of data could, feasibly, be rabin-fingerprinted at the same time; however, given the computational efficiency of doing a rolling hash sequentially, this seems ill-advised.

(ii) **SHA-256**:
There is no data-level parallelism within the computation for each subchunk. There is data-level parallelism for each sub-chunk as different sub-chunks are to be given to each thread, but since one subchunk computation is dependent on previous subchunk computation, this parallelism can't be exploited.

(iii) **Chunk Matching**:
With each table lookup result depending (potentially) on the last table entry, engaging in any parallelism here seems foolish.

(iv) **LZW Encoding**:
With every incoming byte's code potentially dependent on the previous byte's code lookup results, there is no sensible data-level parallelism to be leveraged for LZW.

(d) **Pipeline Parallelism**

(i) **Content-Defined Chunking**:
Any particular window of data could, feasibly, be rabin-fingerprinted at the same time; however, given the computational efficiency of doing a rolling hash sequentially, this seems ill-advised.

(ii) **SHA-256**:
The entire SHA main loop operating on input chunks coming from CDC could be feasibly pipelined. II in this case would be equal to the no. of cycles it takes to complete hash computation for 1 chunk. The II is restrcited by dependencies of internal variables within the subchunk computation. The depth of pipeline is no. of subchunks in an input chunk.

(iii) **Chunk Matching**:
There aren't really enough tasks here to pipeline sensibly.

(iv) **LZW Encoding**:
If the end goal is to fill some kind of input/output buffers while dealing with streaming I/O data, any processes like that could be pipelined pretty well. For any of the internals, though, the logic should be atomic enough that the pipelining approach may not work optimally.

(e)

5. Our CPU implementation may end up looking rather different than our eventual hardware implementation, as our current plan is to put as many components as we can onto the FPGA for performance purposes. For instance, our current plan for inter-process communciation involves significantly more streaming interfaces than we currently use.

   That said, our current implementation is functional; we have a CPU version of Rabin Fingerprinting, SHA-256, and LZW Compression all implemented. There are some significant gaps; for instance, the LZW implementation just `mallocs` a whole table space that would be frustratingly sparse, and the hash table for chunk deduplication is just an array which we search through on each iteration.

   Currently, the interfaces between components boil down to shared memory buffers and pointers within them. The program will only LZW-compress the chunks which are not duplicates, which is likely not how we would operate in hardware. Additionally, there are certain tunable parameters that we will likely ship to different locations; for instance, throwing `CHUNK_SIZE`-related parameters into a single header file.

   In all though, in terms of getting some of the algorithms together, it's a feasible substitute.