ESE532 Project P1 Report

Ritika Gupta, Taylor Nelms, and Nishanth Shyamkumar

October 31, 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**:

```
skip input to minChunkSize − windowSize
buffer = input[minChunkSize − windowSize : minChunkSize]
curHash = 0
for byte in buffer:
    curHash += hash(byte)
if curHash == 0:
    markChunkBreak()
else:
    while (curHash != 0 and (notAtMaxChunkSize())):
        curHash −= hash(buffer[0])
        moveBufferWindow()
        readNextByte()
        curHash += hash(buffer[windowSize − 1])
    markChunkBreak()
```

   (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

# TAYLOR'S VERSION
h[0:7] = initializeHashValues()
k[0:63] = initializeRoundConstants()
padInitialMessage()#pads to a 512-bit boundary
for chunk512bitSection in chunk:
    w[0:15] = chunk512bitSection

    #Extend the first 16 words into the remaining 48 words w[16..63] of the message s
    for i from 16 to 63
        s0 := (w[i-15] rightrotate  7) xor (w[i-15] rightrotate 18) xor (w[i-15] righ
3)
        s1 := (w[i- 2] rightrotate 17) xor (w[i- 2] rightrotate 19) xor (w[i- 2] righ
        w[i] := w[i-16] + s0 + w[i-7] + s1

    a:h = h[0:7]
    #Compression function main loop:
```

```
    for i from 0 to 63
        S1 := (e rightrotate 6) xor (e rightrotate 11) xor (e rightrotate 25)
        ch := (e and f) xor ((not e) and g)
        temp1 := h + S1 + ch + k[i] + w[i]
        S0 := (a rightrotate 2) xor (a rightrotate 13) xor (a rightrotate 22)
        maj := (a and b) xor (a and c) xor (b and c)
        temp2 := S0 + maj

        h := g
        g := f
        f := e
        e := d + temp1
        d := c
        c := b
        b := a
        a := temp1 + temp2

    h[0:7] += [a:h]

digest = h0 append h1 append h2 append h3 append h4 append h5 append h6 append h7
```

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.

(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**:

  (iv) **LZW Encoding**:

(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**:

  (iv) **LZW Encoding**:

(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**:

(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**:

(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. (a)

  (b)

  (c)

  (d)

  (e)

  (f)