ESE 532 Analysis Milestone

Sheil Sarda, Anthony Stewart, Shaokang Xia

1. Report the partners you have agreed to work with for the project duration.

a.

Sheil Sarda, Anthony Stewart, Shaokang Xia

2. Specifically considering processing the input stream at 1 Gb/s:

a. $64b/(1Gb/s) * 1.2GHz = 72\ cycles$

b. $64b/(1Gb/s) * 200MHz = 12\ cycles$

3. Working from the high-level description of the computations involved, assess the computational and memory requirements for each of the coarse-grained operations (ContentDefined Chunking, SHA-256, chunk matching for deduplication, LZW encoding).

a.

Coarse-grained operation	Overview of the algorithm
Content-Defined Chunking	Chunks are split up based on patterns in the data so they are of variable length with average size close to the desired length. Since the chunk boundaries shift along with data patterns, duplicates are still found. To find duplicates, the common technique is to compute a hash value of a few consecutive bytes at every byte position in the data stream. If the hash value matches a predefined pattern we can declare a chunk boundary at that position. A rolling hash uses a sliding window that scans over the data bytes and provides a hash value at each point. The hash value at position I can be computed using a recurrence relation of the has at position I - 1. Hashing requires an addition, subtraction, multiplication and conditionally an XOR (modulus) for each byte position.
SHA-256	Key for hash table
Chunk matching for deduplication	SHA-256 hashes to screen for duplicate chunks using a Hash map.
LZW encoding	Used to compress non-duplicate chunks

Pseudocode

Content-Defined Chunking (Rabin Fingerprint)

```
content_defined_chunking(string s[n], string pattern[m]){    //m window size
      hash_pattern=hash(pattern);
      for (i from 0 to n-m){
            hash_temp=hash(s[i, i+m-1]);
            if (hash_temp==hash_pattern){
                  if (s[i, i+m-1]==pattern)     return i; //return the
position of match
            }
      }
      return False;
}


/* Definition of the hash function */
hash(string input[m]){        //m window size
      prime = 101;
      if currently at the beginning of the original string
            sum=0;
            for (i from 0 to m-1){
                  sum += ascii(input[i])*prime^(i);
            }

      else
            sum= (old_sum-ascii(old_input[i]))/prime
      +ascii(input[m-1])*prime^(m-1);
      return sum;
}
```

- Credit source:
  https://en.wikipedia.org/wiki/Rabin%E2%80%93Karp_algorithm
  https://en.wikipedia.org/wiki/Rolling_hash

SHA-256

```
sha256 (string message[L]){
      /* initial hash values */
      h0 := 0x6a09e667
      h1 := 0xbb67ae85
      h2 := 0x3c6ef372
      h3 := 0xa54ff53a
      h4 := 0x510e527f
      h5 := 0x9b05688c
      h6 := 0x1f83d9ab
      h7 := 0x5be0cd19
```

```
    /* Initialize array of round constants:
    (first 32 bits of the fractional parts of the cube roots of the first 64 primes 2..311): */
    k[0..63] :=
      0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
      0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
      0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
      0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
      0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
      0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
      0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
      0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2

    /* preprocessing */
    message.append(1);
    Append K '0's to message, so that (L+1+K+64)mod (512) = 0;
    Append the value L as a 64bit integer, thus making (L+1+K+64)mod (512) = 0;

    /* Process the message in successive 512-bit chunks: */
    break message into 512-bit chunks
    for each chunk
            create a 64-entry message schedule array w[0..63] of 32-bit words
            (The initial values in w[0..63] don't matter, so many implementations zero them here)
            copy chunk into first 16 words w[0..15] of the message schedule array

            Extend the first 16 words into the remaining 48 words w[16..63] of the message schedule array:
            for i from 16 to 63
                    s0 := (w[i-15] right_rotate  7) xor (w[i-15] right_rotate 18) xor (w[i-15] right_shift  3)
                    s1 := (w[i- 2] right_rotate 17) xor (w[i- 2] right_rotate 19) xor (w[i- 2] right_shift 10)
                    w[i] := w[i-16] + s0 + w[i-7] + s1

            Initialize working variables to current hash value:
            a := h0
            b := h1
            c := h2
            d := h3
            e := h4
            f := h5
            g := h6
            h := h7

/* Compression function main loop: */
            for i from 0 to 63
                    S1 := (e right_rotate 6) xor (e right_rotate 11) xor (e right_rotate 25)
                    ch := (e and f) xor ((not e) and g)
                    temp1 := h + S1 + ch + k[i] + w[i]
                    S0 := (a right_rotate 2) xor (a right_rotate 13) xor (a right_rotate 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

            Add the compressed chunk to the current hash value:
            h0 = h0 + a
            h1 = h1 + b
```

```
            h2 = h2 + c
            h3 = h3 + d
            h4 = h4 + e
            h5 = h5 + f
            h6 = h6 + g
            h7 = h7 + h

        /* Produce the final hash value (big-endian): */
        digest := hash := h0 append h1 append h2 append h3 append h4 append h5 append h6 append h7;
        return digest;
}
```

- Credit source:
  https://en.wikipedia.org/wiki/SHA-2
  https://www.youtube.com/watch?v=PMOEdd4yzyU

Chunk matching for deduplication

Given the chunks & hashes from Rabin Fingerprint and SHA-256:
- store the chunks and hashes into a hashmap, with SHA-256 hashes as the key;
- store the hashes into arrays (for indexing & retrieving the corresponding chunk)

- Credit source:
  https://www.youtube.com/watch?v=8fT8hCn5HyM

LZW encoding

1. List the initial input;
2. Select the longest substring in current string that matches the data to send;
3. Encode the substring as <x,y>, where x is the position of the initial character, y is the length of the substring
4. Repeat 2 and 3 until the end of the message

- Credit source:
  Lecture 17

b.

Coarse-grained operation	Memory needed to support this task
Content-Defined Chunking	O(input_size)
SHA-256	O(input_size*chunk_size)

Chunk matching for deduplication	O(input_size/chunk_size)
LZW encoding	O(chunk_size*input_size)

c.

Coarse-grained operation	**Computational work required per byte / chunk**
Content-Defined Chunking	1 multiplication, 1 addition, 2 comparisons per byte
SHA-256	576 right_rotates; 96 right_shifts 576 XORs; 599 additions; 320 ANDs; 64 negations (2231 operations in total) per chunk
Chunk matching for deduplication	0
LZW encoding	1 comparison per byte

d.

Coarse-grained operation	**Memory operations required per byte / chunk**
Content-Defined Chunking	1 read from/write to memory per byte
SHA-256	Chunk_size memory operations per chunk
Chunk matching for deduplication	2*input_size in total
LZW encoding	1 memory lookup per byte

e.

Coarse-grained operation pair	**Data communicated**
Content-Defined Chunking	resulting chunks -> SHA256 and deduplication
SHA-256	Hashes -> deduplication
Chunk matching for deduplication	Deduplicated results (dictionary)-> LZW encoding

LZW encoding	N/A

f.

Coarse-grained operation	Data Rates (assuming input arrives at 1Gb / s)
Content-Defined Chunking -> SHA-256	5 operations per byte in CDC, 1Gb=1.25*10^8B => 6.25*10^8 operations, 200MHz => 3.125 s => data rate: 1Gb/3.125s=320 Mb/s
SHA-256 -> Deduplication	(2231+2048)/2048=2.1 operation/byte; 1.92Gb=2.4*10^8B => 5.04*10^8 operations, 200MHz => 2.52s => data rate: 320Mb/2.52s=127 Mb/s
Deduplication -> LZW encoding	2*input_size=2*324M/8=81*10^6 operations; 200MHz => 0.405 s => data rate: 127Mb/0.405s=313.5Mb/s

g.

Considering the results in 3f, and a possible parallelism with a factor of 4, the throughput would be 4*127Mb/s=508Mb/s.

4. Identify and characterize parallelism available.

a.

Coarse-grained operation	Parallelism opportunities
Content-Defined Chunking	N/A
SHA-256	Dependent on the results from Content-Defined-Chunking in order to compute the SHA-256 hash keys
Chunk matching for deduplication	Dependent on the results from SHA-256 to have hash keys and values for this stage
LZW encoding	Dependent on the hash map created at the earlier stage to first identify unique chunks which need to be LZW encoded

b.

Coarse-grained operation	Opportunities for thread-level parallelism
Content-Defined Chunking	(1) After chunk boundaries have been identified, the rolling hashes for each chunk can be computed on independent threads since there are no dependencies between chunks
SHA-256	(1) The SHA-256 for each chunk's rolling hash can be parallelized using multiple threads
Chunk matching for deduplication	(1) The SHA-256 fingerprints from the previous step must be inserted into the hash map in a thread-safe manner (using mutexes or semaphores) because we want to avoid storing duplicate chunks into the hash map
LZW encoding	(1) Similar to the previous step, The LZW encoding process also needs to occur in a thread-safe manner because all the encode operations need to refer to the same LZW tree to determine their encoding

c.

Within an operation thread, the following opportunities exist for data-level parallelism:

Coarse-grained operation	Opportunities for data-parallelism
Content-Defined Chunking	Within each thread, assuming each thread only handles one chunk, there are not many opportunities for data-parallelism because the rolling hash for each chunk needs to be computed in a sequential fashion using following recurrence relation `rollhash = (rollhash * PRIME + inbyte - outbyte * POW) % MODULUS`
SHA-256	Assuming each thread handles multiple rolling hashes (i.e. multiple content-defined chunks), the SHA-256 calculation within each thread is data parallel since there are no dependencies between their hash calculations.
Chunk matching for deduplication	Assuming all SHA-256 key value pairs get inserted into the hash map in a thread-safe manner, not many opportunities exist for data level parallelism, because the hash map is updated after every operation and each operation needs to check for duplicates from the map before inserting.
LZW encoding	Assuming each thread handles LZW encoding for multiple entries in the hashmap, no data-parallel opportunities occur for the same reason as hashing. The LZW tree is updated at each iteration.

d.

Within an operation thread, the following opportunities exist for pipelined computation:

Coarse-grained operation	Opportunities for data-parallelism
Content-Defined Chunking	We can pipeline the computation of the rolling hash, which is recursively updated as follows: `rollhash = (rollhash * PRIME + inbyte - outbyte * POW) % MODULUS`
SHA-256	No need to pipeline, since all the operations can be parallelized
Chunk matching for deduplication	All SHA-256 key value pairs can get inserted in a pipelined manner.
LZW encoding	Encoding entries in the hashmap can occur in a pipelined manner, with the LZW tree getting updated at each iteration. As soon as a key value pair gets inserted into the hash map, we can start encoding it via LZW

GIven the above opportunities, we can compute the Initiation Interval as follows:

<u>Dependencies determining the Initiation Interval:</u> the number of clock cycles it takes to perform one iteration of the pipeline for each coarse-grained operation, resource constraints, cross-iteration dependencies (recurrences), period of the clock, etc.

<u>Depth of the pipelines:</u>

Coarse-grained operation	Depth of pipeline
Content-Defined Chunking	If we increase the clock period to accommodate the entire C-D rolling hash computation for a chunk in one cycle, than we can reduce the depth of the pipeline to 1 stages
SHA-256	No need to pipeline, since all the operations can be parallelized. So, 1-deep pipeline assuming that we can compute the SHA-256 for a C-D chunk.
Chunk matching for deduplication	This operation can occur right after the SHA-256 hash gets computed, so both of them can get pipelined together. This is also a 1-deep pipeline.
LZW encoding	In the case that a new key value pair gets inserted into the hash map, it can also get passed along to the next stage to get LZW encoded. Thus, we can append this stage to the pipeline after chunk matching using hash maps

Talking specifically about the pipeline of coarse-grained operations:

- SHA-256

- Chunk matching for deduplication
- LZW encoding

We can achieve an Initiation interval equal to the max number of cycles taken by each of the individual operations, since that is what determines the throughput of the pipeline.

e.

Latency bound for the entire deduplication / compression flow for a single chunk, given

- Maximum chunk size of 8KB
- 1.2Ghz

Coarse-grained operation	Data Rates (assuming input arrives at 1Gb / s)
Content-Defined Chunking -> SHA-256	5 operations per byte in CDC 8KB=>8e3 * 5 = 4e4 operations 1.2e9 hz => 3.33e-5 seconds
SHA-256 -> Deduplication	(2231+2048)/2048 = 2.1 operation/byte 8KB=> 8e3 * 2.1 = 1.68e4 operations 1.2e9 hz => 1.4e-5 seconds
Deduplication -> LZW encoding	2*input_size=2*8KB= 1.6e4 operations 1.2Ghz => 1.33e-5

To compute the critical path for the dedup / compression flow for a chunk, we use the above data rates from Q3:

Latency bound = 3.33e-5 + 1.4e-5 + 1.33e-5 seconds = 6.06e-5 seconds

5. Develop placeholder encoder solution in C.

c. While designing the individual components for the encoder, we took into consideration the data formats, data rates, data communication, and synchronization previously discussed in question 3 and 4.

e.

- Component interfaces:
  - `static chunks Chunk(std::string input, char pattern)`
  - `static uint64_t* Hash(chunks chunkList)`
  - `static hashMap ChunkMatch(uint64_t* hashKeys)`
  - `static std::vector<int> Encode(hashMap chunkMap)`
- We partitioned our code into 4 functions to perform the corresponding operations on the input data.
  - Chunk(): performs the content-defined chunking, simplified by chunking based on an input character pattern.
  - Hash(): performs SHA256 hashing on input chunks, simplified by using addition modulo 2^{64} as a placeholder hash.
  - ChunkMatch(): generates a hashmap using the given chunks and SHA256 hashes, simplified by ignoring duplicate hash values for the time being.
  - Encode(): performs LZW encoding using the previously generated hashmap, simplified by doing a simple mapping between chunks and their respective hashes with no actual encoding done on the chunks.
- The code is currently able to generate plaintext encodings of the inputted text with the correct header value.