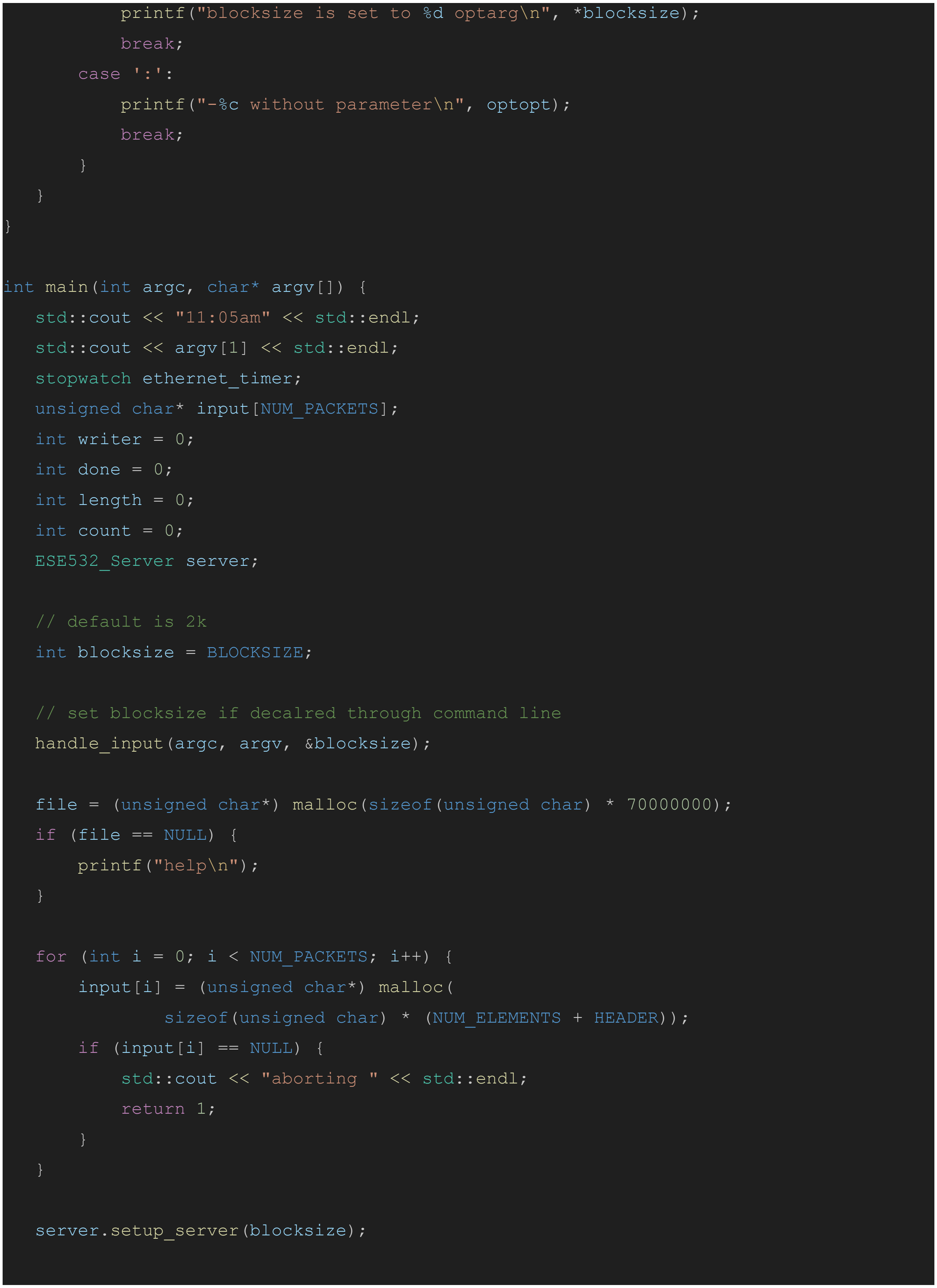
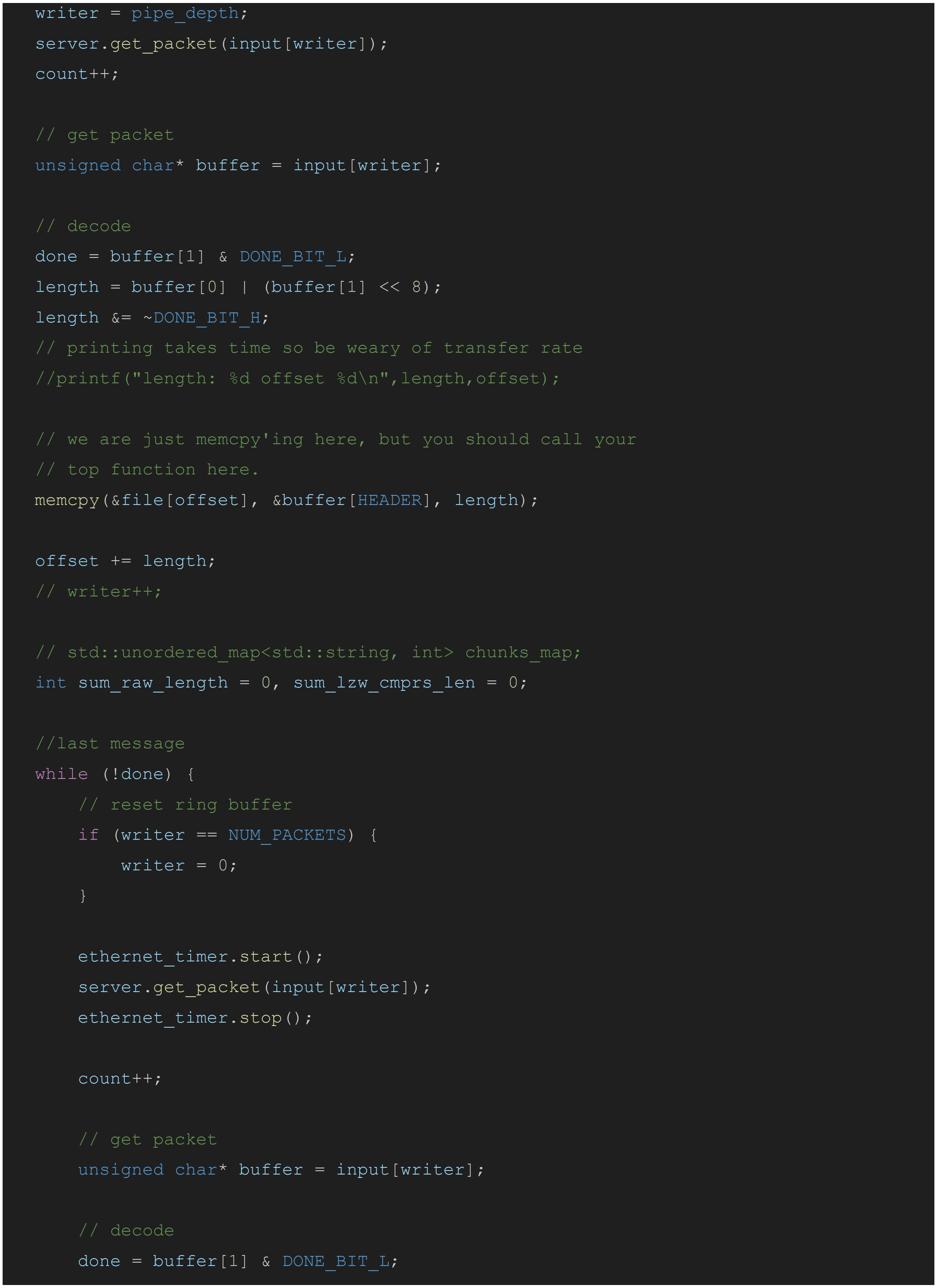
The current implementation of the whole encoder which almost achieves all the required functionalities. But some details might be refreshed and polished in the future:

Encoder.cpp:

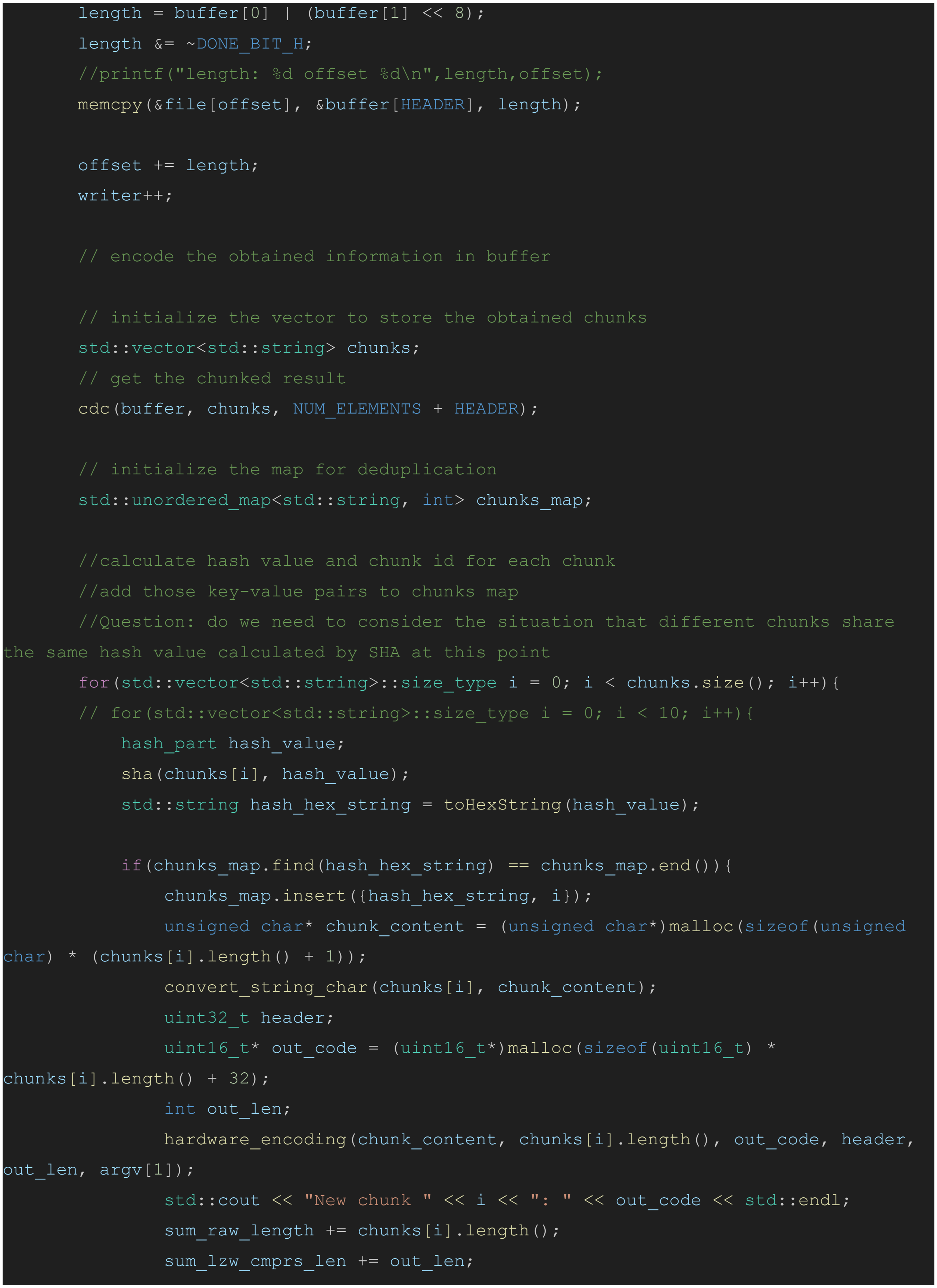
```
#include "encoder.h"
#include "cdc.h"
#include "sha.h"
#include "lzw.h"
#include "utils.h"
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
#include <fstream>
#include "server.h"
#include <unistd.h>
#include <fcntl.h>
#include <pthread.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unordered_map>
#include <vector>
#include "stopwatch.h"
#include "deduplication.h"

#define NUM_PACKETS 8
#define pipe_depth 4
#define DONE_BIT_L (1 << 7)
#define DONE_BIT_H (1 << 15)

int offset = 0;
unsigned char* file;

void handle_input(int argc, char* argv[], int* blocksize) {
   int x;
   extern char *optarg;

   while ((x = getopt(argc, argv, ":b:")) != -1) {
       switch (x) {
       case 'b':
           *blocksize = atoi(optarg);
```



```
           printf("blocksize is set to %d optarg\n", *blocksize);
           break;
       case ':':
           printf("-%c without parameter\n", optopt);
           break;
       }
   }
}

int main(int argc, char* argv[]) {
   std::cout << "11:05am" << std::endl;
   std::cout << argv[1] << std::endl;
   stopwatch ethernet_timer;
   unsigned char* input[NUM_PACKETS];
   int writer = 0;
   int done = 0;
   int length = 0;
   int count = 0;
   ESE532_Server server;

   // default is 2k
   int blocksize = BLOCKSIZE;

   // set blocksize if decalred through command line
   handle_input(argc, argv, &blocksize);

   file = (unsigned char*) malloc(sizeof(unsigned char) * 70000000);
   if (file == NULL) {
       printf("help\n");
   }

   for (int i = 0; i < NUM_PACKETS; i++) {
       input[i] = (unsigned char*) malloc(
               sizeof(unsigned char) * (NUM_ELEMENTS + HEADER));
       if (input[i] == NULL) {
           std::cout << "aborting " << std::endl;
           return 1;
       }
   }

   server.setup_server(blocksize);
```



```
writer = pipe_depth;
server.get_packet(input[writer]);
count++;

// get packet
unsigned char* buffer = input[writer];

// decode
done = buffer[1] & DONE_BIT_L;
length = buffer[0] | (buffer[1] << 8);
length &= ~DONE_BIT_H;
// printing takes time so be weary of transfer rate
//printf("length: %d offset %d\n",length,offset);

// we are just memcpy'ing here, but you should call your
// top function here.
memcpy(&file[offset], &buffer[HEADER], length);

offset += length;
// writer++;

// std::unordered_map<std::string, int> chunks_map;
int sum_raw_length = 0, sum_lzw_cmprs_len = 0;

//last message
while (!done) {
    // reset ring buffer
    if (writer == NUM_PACKETS) {
        writer = 0;
    }

    ethernet_timer.start();
    server.get_packet(input[writer]);
    ethernet_timer.stop();

    count++;

    // get packet
    unsigned char* buffer = input[writer];

    // decode
    done = buffer[1] & DONE_BIT_L;
```



```
        length = buffer[0] | (buffer[1] << 8);
        length &= ~DONE_BIT_H;
        //printf("length: %d offset %d\n",length,offset);
        memcpy(&file[offset], &buffer[HEADER], length);

        offset += length;
        writer++;

        // encode the obtained information in buffer

        // initialize the vector to store the obtained chunks
        std::vector<std::string> chunks;
        // get the chunked result
        cdc(buffer, chunks, NUM_ELEMENTS + HEADER);

        // initialize the map for deduplication
        std::unordered_map<std::string, int> chunks_map;

        //calculate hash value and chunk id for each chunk
        //add those key-value pairs to chunks map
        //Question: do we need to consider the situation that different chunks share
the same hash value calculated by SHA at this point
        for(std::vector<std::string>::size_type i = 0; i < chunks.size(); i++){
        // for(std::vector<std::string>::size_type i = 0; i < 10; i++){
            hash_part hash_value;
            sha(chunks[i], hash_value);
            std::string hash_hex_string = toHexString(hash_value);

            if(chunks_map.find(hash_hex_string) == chunks_map.end()){
                chunks_map.insert({hash_hex_string, i});
                unsigned char* chunk_content = (unsigned char*)malloc(sizeof(unsigned
char) * (chunks[i].length() + 1));
                convert_string_char(chunks[i], chunk_content);
                uint32_t header;
                uint16_t* out_code = (uint16_t*)malloc(sizeof(uint16_t) *
chunks[i].length() + 32);
                int out_len;
                hardware_encoding(chunk_content, chunks[i].length(), out_code, header,
out_len, argv[1]);
                std::cout << "New chunk " << i << ": " << out_code << std::endl;
                sum_raw_length += chunks[i].length();
                sum_lzw_cmprs_len += out_len;
                free(out_code);
                free(chunk_content);
            }

            else{
                uint32_t out_code;
                duplicate_encoding(chunks_map.at(hash_hex_string), out_code, argv[1]);
                std::cout << "Duplicate chunk " << i << ": " << out_code << std::endl;
            }
        }
    }
    float lzw_compress_ratio = sum_raw_length / sum_lzw_cmprs_len;
    std::cout << "LZW compress ratio: " << lzw_compress_ratio << std::endl;


    // write file to root and you can use diff tool on board
    FILE *outfd = fopen("output_cpu.bin", "wb");
    int bytes_written = fwrite(&file[0], 1, offset, outfd);
    printf("write file with %d\n", bytes_written);
    fclose(outfd);

    for (int i = 0; i < NUM_PACKETS; i++) {
        free(input[i]);
    }

    free(file);
    std::cout << "--------------- Key Throughputs ---------------" << std::endl;
    float ethernet_latency = ethernet_timer.latency() / 1000.0;
    float input_throughput = (bytes_written * 8 / 1000000.0) / ethernet_latency; // Mb/s
    std::cout << "Input Throughput to Encoder: " << input_throughput << " Mb/s."
            << " (Latency: " << ethernet_latency << "s)." << std::endl;

    return 0;
}
```

The sample code for OpenCL in previous assignment, might be used as a reference in the OpenCL implementation of my own project:

```
#include "Utilities.h"
```

```
//
------------------------------------------------------------------------------------
// Main program
//
------------------------------------------------------------------------------------
int main(int argc, char** argv)
{
// Initialize an event timer we'll use for monitoring the application
   EventTimer timer;
//
------------------------------------------------------------------------------------
// Step 1: Initialize the OpenCL environment
//
------------------------------------------------------------------------------------
   timer.add("OpenCL Initialization");
   cl_int err;
   std::string binaryFile = argv[1];
   unsigned fileBufSize;
   std::vector<cl::Device> devices = get_xilinx_devices();
   devices.resize(1);
   cl::Device device = devices[0];
   cl::Context context(device, NULL, NULL, NULL, &err);
   char* fileBuf = read_binary_file(binaryFile, fileBufSize);
   cl::Program::Binaries bins{{fileBuf, fileBufSize}};
   cl::Program program(context, devices, bins, NULL, &err);
   cl::CommandQueue q(context, device, CL_QUEUE_PROFILING_ENABLE, &err);
   cl::Kernel krnl_mmult(program,"mmult", &err);

//
------------------------------------------------------------------------------------
// Step 2: Create buffers and initialize test values
//
------------------------------------------------------------------------------------
   timer.add("Allocate contiguous OpenCL buffers");
   // Create the buffers and allocate memory
   cl::Buffer in1_buf(context, CL_MEM_ALLOC_HOST_PTR | CL_MEM_READ_ONLY,
sizeof(matrix_type) * MATRIX_SIZE, NULL, &err);
   cl::Buffer in2_buf(context, CL_MEM_ALLOC_HOST_PTR | CL_MEM_READ_ONLY,
sizeof(matrix_type) * MATRIX_SIZE, NULL, &err);
   cl::Buffer out_buf_hw(context, CL_MEM_ALLOC_HOST_PTR | CL_MEM_WRITE_ONLY,
sizeof(matrix_type) * MATRIX_SIZE, NULL, &err);
```

```
   timer.add("Set kernel arguments");
   // Map buffers to kernel arguments, thereby assigning them to specific device
memory banks
   krnl_mmult.setArg(0, in1_buf);
   krnl_mmult.setArg(1, in2_buf);
   krnl_mmult.setArg(2, out_buf_hw);

   timer.add("Map buffers to userspace pointers");
   // Map host-side buffer memory to user-space pointers
   matrix_type *in1 = (matrix_type *)q.enqueueMapBuffer(in1_buf, CL_TRUE,
CL_MAP_WRITE, 0, sizeof(matrix_type) * MATRIX_SIZE);
   matrix_type *in2 = (matrix_type *)q.enqueueMapBuffer(in2_buf, CL_TRUE,
CL_MAP_WRITE, 0, sizeof(matrix_type) * MATRIX_SIZE);
   matrix_type *out_sw = Create_matrix();

   timer.add("Populating buffer inputs");
   // Initialize the vectors used in the test
   Randomize_matrix(in1);
   Randomize_matrix(in2);

//
-------------------------------------------------------------------------------------
// Step 3: Run the kernel
//
-------------------------------------------------------------------------------------
   timer.add("Set kernel arguments");
   // Set kernel arguments
   krnl_mmult.setArg(0, in1_buf);
   krnl_mmult.setArg(1, in2_buf);
   krnl_mmult.setArg(2, out_buf_hw);

   // Schedule transfer of inputs to device memory, execution of kernel, and transfer
of outputs back to host memory
   timer.add("Memory object migration enqueue host->device");
   cl::Event event_sp;
   q.enqueueMigrateMemObjects({in1_buf, in2_buf}, 0 /* 0 means from host*/, NULL,
&event_sp);
   clWaitForEvents(1, (const cl_event *)&event_sp);

   timer.add("Launch mmult kernel");
   q.enqueueTask(krnl_mmult, NULL, &event_sp);
   timer.add("Wait for mmult kernel to finish running");
   clWaitForEvents(1, (const cl_event *)&event_sp);

   timer.add("Read back computation results (implicit device->host migration)");
   matrix_type *out_hw = (matrix_type *)q.enqueueMapBuffer(out_buf_hw, CL_TRUE,
CL_MAP_READ, 0, sizeof(matrix_type) * MATRIX_SIZE);
   timer.finish();

//
------------------------------------------------------------------------------------
// Step 4: Check Results and Release Allocated Resources
//
------------------------------------------------------------------------------------
   multiply_gold(in1, in2, out_sw);
   bool match = Compare_matrices(out_sw, out_hw);
   Destroy_matrix(out_sw);
   delete[] fileBuf;
   q.enqueueUnmapMemObject(in1_buf, in1);
   q.enqueueUnmapMemObject(in2_buf, in2);
   q.enqueueUnmapMemObject(out_buf_hw, out_hw);
   q.finish();

   std::cout << "--------------- Key execution times ---------------" << std::endl;
   timer.print();

   std::cout << "TEST " << (match ? "PASSED" : "FAILED") << std::endl;
   return (match ? EXIT_SUCCESS : EXIT_FAILURE);
}
```

The Makefile for the sample code, which might be used as a reference in my own project:

```
HOST_CXX ?= aarch64-linux-gnu-g++
VPP ?= ${XILINX_VITIS}/bin/v++
RM = rm -f
RMDIR = rm -rf

VITIS_PLATFORM = u96v2_sbc_base
VITIS_PLATFORM_DIR = ${PLATFORM_REPO_PATHS}
VITIS_PLATFORM_PATH = $(VITIS_PLATFORM_DIR)/u96v2_sbc_base.xpfm

# host compiler global settings
CXXFLAGS += -march=armv8-a+simd -mtune=cortex-a53 -std=c++11
-DVITIS_PLATFORM=$(VITIS_PLATFORM) -D__USE_XOPEN2K8 -I$(XILINX_VIVADO)/include/
-I$(VITIS_PLATFORM_DIR)/sw/$(VITIS_PLATFORM)/PetaLinux/sysroot/aarch64-xilinx-linux/us
```

```
r/include/xrt/ -O3 -g -Wall -c -fmessage-length=0
--sysroot=$(VITIS_PLATFORM_DIR)/sw/$(VITIS_PLATFORM)/PetaLinux/sysroot/aarch64-xilinx-linux
LDFLAGS += -lxilinxopencl -lpthread -lrt -ldl -lcrypt -lstdc++
-L$(VITIS_PLATFORM_DIR)/sw/$(VITIS_PLATFORM)/PetaLinux/sysroot/aarch64-xilinx-linux/usr/lib/
--sysroot=$(VITIS_PLATFORM_DIR)/sw/$(VITIS_PLATFORM)/PetaLinux/sysroot/aarch64-xilinx-linux

# hardware compiler shared settings
VPP_OPTS = --target hw


#
# OpenCL kernel files
#
XO := mmult.xo
XCLBIN := mmult.xclbin

#
# host files
#
HOST_SOURCES = ./Host.cpp ./common/EventTimer.cpp ./common/Utilities.cpp
HOST_OBJECTS =$(HOST_SOURCES:.cpp=.o)
HOST_EXE = host


all: package/sd_card.img

.cpp.o:
   $(HOST_CXX) $(CXXFLAGS) -I./hls/ -I./common -o "$@" "$<"

$(HOST_EXE): $(HOST_OBJECTS)
   $(HOST_CXX) -o "$@" $(+) $(LDFLAGS)

# $(XO): ./hls/MatrixMultiplication.cpp
#   $(VPP) $(VPP_OPTS) -k mmult --compile -I"$(<D)" --config --config ./u96_v2.cfg -o"$@" "$<"

$(XCLBIN): $(XO)
   $(VPP) $(VPP_OPTS) --link --config ./u96_v2.cfg -o"$@" $(+)
```

```
package/sd_card.img: $(HOST_EXE) $(XCLBIN) xrt.ini
   $(VPP) --package $(VPP_OPTS) --config ./u96_v2.cfg $(XCLBIN) \
          --package.out_dir package \
          --package.sd_file $(HOST_EXE)\
          --package.kernel_image $(PLATFORM_REPO_PATHS)/sw/u96v2_sbc_base/PetaLinux/image/image.ub \
          --package.rootfs ${PLATFORM_REPO_PATHS}/sw/u96v2_sbc_base/PetaLinux/rootfs/rootfs.ext4 \
          --package.sd_file $(XCLBIN) \
          --package.sd_file xrt.ini

clean:
   -$(RM) $(HOST_EXE) *.o *.xclbin *.xclbin.sh *.xclbin.info *.xclbin.link_summary* *.compile_summary *.str
   -$(RM) *json *csv *log *summary *.json *.jou
   -$(RMDIR) _x
   -$(RMDIR) .Xil
   -$(RMDIR) .ipcache
   -$(RMDIR) package
```