Nolan Anderson | Project Checkpoint 1 | CPE 613 | March 29 2023

1. Describe the strategy you used and steps you took to develop a naive implementation of your semester project. Using code examples and providing examples of output, show me how to build it, run it, and test it.

The naive implementation for this project came from my last semester in CPE 512 (Parallel Programming). There is a serial and parallel (OpenMP) version of the 'naive' approach. Next, there is a 'generate.cpp' file that I use to generate the files to encrypt. All versions of the program will use the same data initially generated from this file. All of these files can be found in the appendix below. The header file included (aeslib.hpp) contains the common functions that are associated with AES encryption such as the add round key and shift rows.

These files can be compiled easily using:
g++ <filename>.cpp -o <outname> -std=c++11

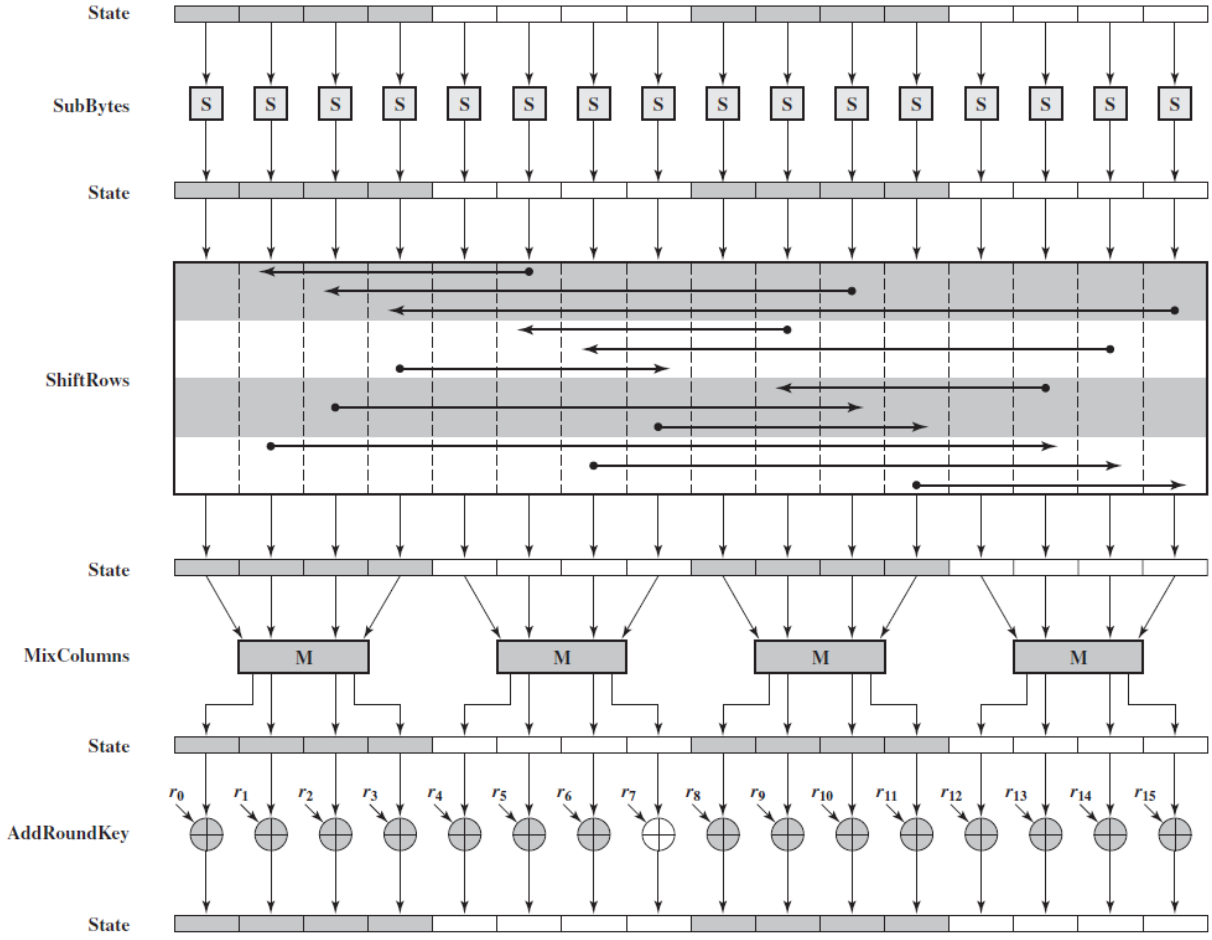
And can be executed using:
./<outname>

```
uahclsc0003@dmcvlogin3:src> ./parallel
========== STARTING AES ENCRYPTION PARALLEL ==========
Files encrypted: 100 |Batch #0| Time To Encrypt 159ms
Files encrypted: 100 |Batch #1| Time To Encrypt 235ms
Files encrypted: 100 |Batch #2| Time To Encrypt 211ms
Files encrypted: 100 |Batch #3| Time To Encrypt 179ms
Files encrypted: 100 |Batch #4| Time To Encrypt 187ms
Files encrypted: 200 |Batch #0| Time To Encrypt 266ms
Files encrypted: 200 |Batch #1| Time To Encrypt 258ms
Files encrypted: 200 |Batch #2| Time To Encrypt 293ms
Files encrypted: 200 |Batch #3| Time To Encrypt 332ms
Files encrypted: 200 |Batch #4| Time To Encrypt 382ms
Files encrypted: 300 |Batch #0| Time To Encrypt 519ms
Files encrypted: 300 |Batch #1| Time To Encrypt 486ms
Files encrypted: 300 |Batch #2| Time To Encrypt 583ms
Files encrypted: 300 |Batch #3| Time To Encrypt 589ms
Files encrypted: 300 |Batch #4| Time To Encrypt 387ms
========== ENDING AES ENCRYPTION PARALLEL ==========
uahclsc0003@dmcvlogin3:src> g++ serial.cpp -o serial -fopenmp -std=c++11
uahclsc0003@dmcvlogin3:src> ./serial
========== STARTING AES ENCRYPTION SERIALLY ==========
Files encrypted: 100 |Batch #0| Time To Encrypt 835ms
Files encrypted: 100 |Batch #1| Time To Encrypt 801ms
Files encrypted: 100 |Batch #2| Time To Encrypt 818ms
Files encrypted: 100 |Batch #3| Time To Encrypt 766ms
Files encrypted: 100 |Batch #4| Time To Encrypt 829ms
Files encrypted: 200 |Batch #0| Time To Encrypt 1665ms
Files encrypted: 200 |Batch #1| Time To Encrypt 1549ms
Files encrypted: 200 |Batch #2| Time To Encrypt 1580ms
Files encrypted: 200 |Batch #3| Time To Encrypt 1657ms
Files encrypted: 200 |Batch #4| Time To Encrypt 1594ms
Files encrypted: 300 |Batch #0| Time To Encrypt 2527ms
Files encrypted: 300 |Batch #1| Time To Encrypt 2496ms
Files encrypted: 300 |Batch #2| Time To Encrypt 2545ms
Files encrypted: 300 |Batch #3| Time To Encrypt 2459ms
Files encrypted: 300 |Batch #4| Time To Encrypt 2447ms
========== ENDING AES ENCRYPTION SERIALLY ==========
uahclsc0003@dmcvlogin3:src>
```

Right is the output for the serial and parallel versions and will show how I am gathering performance data. I have implemented this in a way where each program encrypts 100, 200, and 300 files. Each batch represents a different set of input files to encrypt. This gives the program a good set of randomized data to execute on. The time to encrypt each batch of files is also outputted.

The screenshot below shows the intent of aeslib.hpp and what it implements. Each 'state' represents what states the bytes are in while they are going through the encryption process. The first step in encryption we sub bytes which simply replaces the data in the state with the data in the header. Next, we shift the rows which 'scrambles' the data in the state and. Mix columns is by far the most expensive operation in AES and performs operations on the predefined state arrays. Lastly, we

do the addround key which gives us our final state for the data. Fortunately, we can parallelize all of these steps which is why we see so much of a performance difference. If we can run all of these at the same time, then it is going to go much faster.



For the CUDA side of things, I have generated some ‘skeleton’ code so that I can start putting code where it needs to go.

2. What specifically will your first step towards optimization be? Note that this shouldn't be a monumental effort, but one specific improvement that you could complete in the next day or two.

Right now I don’t have any code to optimize (at least on the CUDA side). I plan to leave the serial and openMP versions alone since they are my references. Due to the nature of AES, when I do have code to optimize I plan to load the sbox’s and state tables into constant / shared memory. This will allow the GPU direct access to the

data that it needs to calculate with. Unfortunately, because the state changes each round, loading the initial array is about all we can do. The biggest optimization roadblock is going to be optimizing the threads / blocks. Since AES only encrypts on 16 bytes at a time, I will need a way to load up a large block of data so I am not copying over little bits of data to the device every time. I am still brainstorming ways to do this, but getting it right will give big improvements in performance.

3. What questions do you have for me and how specifically can I help you succeed in your project work over the next few weeks?

Due to time constraints I have not been able to implement the decryption of AES. I plan to readjust my schedule to focus on implementing and optimizing encryption all the way before focusing on the decryption. Is it ok if I focus on encryption and optimization and add decryption as a stretch assignment? Other than that I don't have a lot of questions since I have not gotten very far on the CUDA side. Here is my revised schedule:

March 30th - April 13th
- Implement AES Encryption using CUDA
- Begin initial testing and comparing performance between the implementations.
- Begin implementing optimization techniques.
- Start writing report and compile performance data

April 14th - Due data
- Wrap up optimizations for CUDA
- Final testing and comparison
- Video assignment
- Complete report

An additional small assignment will be to output the data to ensure all the implementations match. This wont take very long since the state of the bytes can be seen at the end of each round. I can just run some simple output.

Appendix A - aeslib.hpp

```
#ifndef AESLIB_H
#define AESLIB_H
#include <iostream>
#include <string>
#include <stdlib.h>
#include <fstream>
#include <vector>
#include <omp.h>
```

```
#include <chrono>
typedef unsigned char byte;

#define N_ROUNDS 10

byte sbox[256] =
{
   0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5, 0x30, 0x01, 0x67, 0x2B, 0xFE, 0xD7, 0xAB, 0x76,
   0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0, 0xAD, 0xD4, 0xA2, 0xAF, 0x9C, 0xA4, 0x72, 0xC0,
   0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC, 0x34, 0xA5, 0xE5, 0xF1, 0x71, 0xD8, 0x31, 0x15,
   0x04, 0xC7, 0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A, 0x07, 0x12, 0x80, 0xE2, 0xEB, 0x27, 0xB2, 0x75,
   0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0, 0x52, 0x3B, 0xD6, 0xB3, 0x29, 0xE3, 0x2F, 0x84,
   0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B, 0x6A, 0xCB, 0xBE, 0x39, 0x4A, 0x4C, 0x58, 0xCF,
   0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85, 0x45, 0xF9, 0x02, 0x7F, 0x50, 0x3C, 0x9F, 0xA8,
   0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5, 0xBC, 0xB6, 0xDA, 0x21, 0x10, 0xFF, 0xF3, 0xD2,
   0xCD, 0x0C, 0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17, 0xC4, 0xA7, 0x7E, 0x3D, 0x64, 0x5D, 0x19, 0x73,
   0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A, 0x90, 0x88, 0x46, 0xEE, 0xB8, 0x14, 0xDE, 0x5E, 0x0B, 0xDB,
   0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C, 0xC2, 0xD3, 0xAC, 0x62, 0x91, 0x95, 0xE4, 0x79,
   0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5, 0x4E, 0xA9, 0x6C, 0x56, 0xF4, 0xEA, 0x65, 0x7A, 0xAE, 0x08,
   0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6, 0xE8, 0xDD, 0x74, 0x1F, 0x4B, 0xBD, 0x8B, 0x8A,
   0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E, 0x61, 0x35, 0x57, 0xB9, 0x86, 0xC1, 0x1D, 0x9E,
   0xE1, 0xF8, 0x98, 0x11, 0x69, 0xD9, 0x8E, 0x94, 0x9B, 0x1E, 0x87, 0xE9, 0xCE, 0x55, 0x28, 0xDF,
   0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68, 0x41, 0x99, 0x2D, 0x0F, 0xB0, 0x54, 0xBB, 0x16
};

byte mul2[256] =
{
   0x00, 0x02, 0x04, 0x06, 0x08, 0x0a, 0x0c, 0x0e, 0x10, 0x12, 0x14, 0x16, 0x18, 0x1a, 0x1c, 0x1e,
   0x20, 0x22, 0x24, 0x26, 0x28, 0x2a, 0x2c, 0x2e, 0x30, 0x32, 0x34, 0x36, 0x38, 0x3a, 0x3c, 0x3e,
   0x40, 0x42, 0x44, 0x46, 0x48, 0x4a, 0x4c, 0x4e, 0x50, 0x52, 0x54, 0x56, 0x58, 0x5a, 0x5c, 0x5e,
   0x60, 0x62, 0x64, 0x66, 0x68, 0x6a, 0x6c, 0x6e, 0x70, 0x72, 0x74, 0x76, 0x78, 0x7a, 0x7c, 0x7e,
   0x80, 0x82, 0x84, 0x86, 0x88, 0x8a, 0x8c, 0x8e, 0x90, 0x92, 0x94, 0x96, 0x98, 0x9a, 0x9c, 0x9e,
   0xa0, 0xa2, 0xa4, 0xa6, 0xa8, 0xaa, 0xac, 0xae, 0xb0, 0xb2, 0xb4, 0xb6, 0xb8, 0xba, 0xbc, 0xbe,
   0xc0, 0xc2, 0xc4, 0xc6, 0xc8, 0xca, 0xcc, 0xce, 0xd0, 0xd2, 0xd4, 0xd6, 0xd8, 0xda, 0xdc, 0xde,
   0xe0, 0xe2, 0xe4, 0xe6, 0xe8, 0xea, 0xec, 0xee, 0xf0, 0xf2, 0xf4, 0xf6, 0xf8, 0xfa, 0xfc, 0xfe,
   0x1b, 0x19, 0x1f, 0x1d, 0x13, 0x11, 0x17, 0x15, 0x0b, 0x09, 0x0f, 0x0d, 0x03, 0x01, 0x07, 0x05,
   0x3b, 0x39, 0x3f, 0x3d, 0x33, 0x31, 0x37, 0x35, 0x2b, 0x29, 0x2f, 0x2d, 0x23, 0x21, 0x27, 0x25,
   0x5b, 0x59, 0x5f, 0x5d, 0x53, 0x51, 0x57, 0x55, 0x4b, 0x49, 0x4f, 0x4d, 0x43, 0x41, 0x47, 0x45,
   0x7b, 0x79, 0x7f, 0x7d, 0x73, 0x71, 0x77, 0x75, 0x6b, 0x69, 0x6f, 0x6d, 0x63, 0x61, 0x67, 0x65,
   0x9b, 0x99, 0x9f, 0x9d, 0x93, 0x91, 0x97, 0x95, 0x8b, 0x89, 0x8f, 0x8d, 0x83, 0x81, 0x87, 0x85,
   0xbb, 0xb9, 0xbf, 0xbd, 0xb3, 0xb1, 0xb7, 0xb5, 0xab, 0xa9, 0xaf, 0xad, 0xa3, 0xa1, 0xa7, 0xa5,
   0xdb, 0xd9, 0xdf, 0xdd, 0xd3, 0xd1, 0xd7, 0xd5, 0xcb, 0xc9, 0xcf, 0xcd, 0xc3, 0xc1, 0xc7, 0xc5,
   0xfb, 0xf9, 0xff, 0xfd, 0xf3, 0xf1, 0xf7, 0xf5, 0xeb, 0xe9, 0xef, 0xed, 0xe3, 0xe1, 0xe7, 0xe5
};

byte mul3[256] =
{
   0x00, 0x03, 0x06, 0x05, 0x0c, 0x0f, 0x0a, 0x09, 0x18, 0x1b, 0x1e, 0x1d, 0x14, 0x17, 0x12, 0x11,
   0x30, 0x33, 0x36, 0x35, 0x3c, 0x3f, 0x3a, 0x39, 0x28, 0x2b, 0x2e, 0x2d, 0x24, 0x27, 0x22, 0x21,
   0x60, 0x63, 0x66, 0x65, 0x6c, 0x6f, 0x6a, 0x69, 0x78, 0x7b, 0x7e, 0x7d, 0x74, 0x77, 0x72, 0x71,
   0x50, 0x53, 0x56, 0x55, 0x5c, 0x5f, 0x5a, 0x59, 0x48, 0x4b, 0x4e, 0x4d, 0x44, 0x47, 0x42, 0x41,
   0xc0, 0xc3, 0xc6, 0xc5, 0xcc, 0xcf, 0xca, 0xc9, 0xd8, 0xdb, 0xde, 0xdd, 0xd4, 0xd7, 0xd2, 0xd1,
   0xf0, 0xf3, 0xf6, 0xf5, 0xfc, 0xff, 0xfa, 0xf9, 0xe8, 0xeb, 0xee, 0xed, 0xe4, 0xe7, 0xe2, 0xe1,
```

```
    0xa0, 0xa3, 0xa6, 0xa5, 0xac, 0xaf, 0xaa, 0xa9, 0xb8, 0xbb, 0xbe, 0xbd, 0xb4, 0xb7, 0xb2, 0xb1,
    0x90, 0x93, 0x96, 0x95, 0x9c, 0x9f, 0x9a, 0x99, 0x88, 0x8b, 0x8e, 0x8d, 0x84, 0x87, 0x82, 0x81,
    0x9b, 0x98, 0x9d, 0x9e, 0x97, 0x94, 0x91, 0x92, 0x83, 0x80, 0x85, 0x86, 0x8f, 0x8c, 0x89, 0x8a,
    0xab, 0xa8, 0xad, 0xae, 0xa7, 0xa4, 0xa1, 0xa2, 0xb3, 0xb0, 0xb5, 0xb6, 0xbf, 0xbc, 0xb9, 0xba,
    0xfb, 0xf8, 0xfd, 0xfe, 0xf7, 0xf4, 0xf1, 0xf2, 0xe3, 0xe0, 0xe5, 0xe6, 0xef, 0xec, 0xe9, 0xea,
    0xcb, 0xc8, 0xcd, 0xce, 0xc7, 0xc4, 0xc1, 0xc2, 0xd3, 0xd0, 0xd5, 0xd6, 0xdf, 0xdc, 0xd9, 0xda,
    0x5b, 0x58, 0x5d, 0x5e, 0x57, 0x54, 0x51, 0x52, 0x43, 0x40, 0x45, 0x46, 0x4f, 0x4c, 0x49, 0x4a,
    0x6b, 0x68, 0x6d, 0x6e, 0x67, 0x64, 0x61, 0x62, 0x73, 0x70, 0x75, 0x76, 0x7f, 0x7c, 0x79, 0x7a,
    0x3b, 0x38, 0x3d, 0x3e, 0x37, 0x34, 0x31, 0x32, 0x23, 0x20, 0x25, 0x26, 0x2f, 0x2c, 0x29, 0x2a,
    0x0b, 0x08, 0x0d, 0x0e, 0x07, 0x04, 0x01, 0x02, 0x13, 0x10, 0x15, 0x16, 0x1f, 0x1c, 0x19, 0x1a
};

byte rcon[256] = {
    0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a,
    0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39,
    0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a,
    0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8,
    0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef,
    0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc,
    0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b,
    0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3,
    0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94,
    0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20,
    0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35,
    0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f,
    0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04,
    0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63,
    0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd,
    0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d
};

byte inv_sbox[256] =
{
    0x52, 0x09, 0x6A, 0xD5, 0x30, 0x36, 0xA5, 0x38, 0xBF, 0x40, 0xA3, 0x9E, 0x81, 0xF3, 0xD7, 0xFB,
    0x7C, 0xE3, 0x39, 0x82, 0x9B, 0x2F, 0xFF, 0x87, 0x34, 0x8E, 0x43, 0x44, 0xC4, 0xDE, 0xE9, 0xCB,
    0x54, 0x7B, 0x94, 0x32, 0xA6, 0xC2, 0x23, 0x3D, 0xEE, 0x4C, 0x95, 0x0B, 0x42, 0xFA, 0xC3, 0x4E,
    0x08, 0x2E, 0xA1, 0x66, 0x28, 0xD9, 0x24, 0xB2, 0x76, 0x5B, 0xA2, 0x49, 0x6D, 0x8B, 0xD1, 0x25,
    0x72, 0xF8, 0xF6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xD4, 0xA4, 0x5C, 0xCC, 0x5D, 0x65, 0xB6, 0x92,
    0x6C, 0x70, 0x48, 0x50, 0xFD, 0xED, 0xB9, 0xDA, 0x5E, 0x15, 0x46, 0x57, 0xA7, 0x8D, 0x9D, 0x84,
    0x90, 0xD8, 0xAB, 0x00, 0x8C, 0xBC, 0xD3, 0x0A, 0xF7, 0xE4, 0x58, 0x05, 0xB8, 0xB3, 0x45, 0x06,
    0xD0, 0x2C, 0x1E, 0x8F, 0xCA, 0x3F, 0x0F, 0x02, 0xC1, 0xAF, 0xBD, 0x03, 0x01, 0x13, 0x8A, 0x6B,
    0x3A, 0x91, 0x11, 0x41, 0x4F, 0x67, 0xDC, 0xEA, 0x97, 0xF2, 0xCF, 0xCE, 0xF0, 0xB4, 0xE6, 0x73,
    0x96, 0xAC, 0x74, 0x22, 0xE7, 0xAD, 0x35, 0x85, 0xE2, 0xF9, 0x37, 0xE8, 0x1C, 0x75, 0xDF, 0x6E,
    0x47, 0xF1, 0x1A, 0x71, 0x1D, 0x29, 0xC5, 0x89, 0x6F, 0xB7, 0x62, 0x0E, 0xAA, 0x18, 0xBE, 0x1B,
    0xFC, 0x56, 0x3E, 0x4B, 0xC6, 0xD2, 0x79, 0x20, 0x9A, 0xDB, 0xC0, 0xFE, 0x78, 0xCD, 0x5A, 0xF4,
    0x1F, 0xDD, 0xA8, 0x33, 0x88, 0x07, 0xC7, 0x31, 0xB1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xEC, 0x5F,
    0x60, 0x51, 0x7F, 0xA9, 0x19, 0xB5, 0x4A, 0x0D, 0x2D, 0xE5, 0x7A, 0x9F, 0x93, 0xC9, 0x9C, 0xEF,
    0xA0, 0xE0, 0x3B, 0x4D, 0xAE, 0x2A, 0xF5, 0xB0, 0xC8, 0xEB, 0xBB, 0x3C, 0x83, 0x53, 0x99, 0x61,
    0x17, 0x2B, 0x04, 0x7E, 0xBA, 0x77, 0xD6, 0x26, 0xE1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0C, 0x7D
};

byte mul9[256] =
{
```

```
    0x00, 0x09, 0x12, 0x1b, 0x24, 0x2d, 0x36, 0x3f, 0x48, 0x41, 0x5a, 0x53, 0x6c, 0x65, 0x7e, 0x77,
    0x90, 0x99, 0x82, 0x8b, 0xb4, 0xbd, 0xa6, 0xaf, 0xd8, 0xd1, 0xca, 0xc3, 0xfc, 0xf5, 0xee, 0xe7,
    0x3b, 0x32, 0x29, 0x20, 0x1f, 0x16, 0x0d, 0x04, 0x73, 0x7a, 0x61, 0x68, 0x57, 0x5e, 0x45, 0x4c,
    0xab, 0xa2, 0xb9, 0xb0, 0x8f, 0x86, 0x9d, 0x94, 0xe3, 0xea, 0xf1, 0xf8, 0xc7, 0xce, 0xd5, 0xdc,
    0x76, 0x7f, 0x64, 0x6d, 0x52, 0x5b, 0x40, 0x49, 0x3e, 0x37, 0x2c, 0x25, 0x1a, 0x13, 0x08, 0x01,
    0xe6, 0xef, 0xf4, 0xfd, 0xc2, 0xcb, 0xd0, 0xd9, 0xae, 0xa7, 0xbc, 0xb5, 0x8a, 0x83, 0x98, 0x91,
    0x4d, 0x44, 0x5f, 0x56, 0x69, 0x60, 0x7b, 0x72, 0x05, 0x0c, 0x17, 0x1e, 0x21, 0x28, 0x33, 0x3a,
    0xdd, 0xd4, 0xcf, 0xc6, 0xf9, 0xf0, 0xeb, 0xe2, 0x95, 0x9c, 0x87, 0x8e, 0xb1, 0xb8, 0xa3, 0xaa,
    0xec, 0xe5, 0xfe, 0xf7, 0xc8, 0xc1, 0xda, 0xd3, 0xa4, 0xad, 0xb6, 0xbf, 0x80, 0x89, 0x92, 0x9b,
    0x7c, 0x75, 0x6e, 0x67, 0x58, 0x51, 0x4a, 0x43, 0x34, 0x3d, 0x26, 0x2f, 0x10, 0x19, 0x02, 0x0b,
    0xd7, 0xde, 0xc5, 0xcc, 0xf3, 0xfa, 0xe1, 0xe8, 0x9f, 0x96, 0x8d, 0x84, 0xbb, 0xb2, 0xa9, 0xa0,
    0x47, 0x4e, 0x55, 0x5c, 0x63, 0x6a, 0x71, 0x78, 0x0f, 0x06, 0x1d, 0x14, 0x2b, 0x22, 0x39, 0x30,
    0x9a, 0x93, 0x88, 0x81, 0xbe, 0xb7, 0xac, 0xa5, 0xd2, 0xdb, 0xc0, 0xc9, 0xf6, 0xff, 0xe4, 0xed,
    0x0a, 0x03, 0x18, 0x11, 0x2e, 0x27, 0x3c, 0x35, 0x42, 0x4b, 0x50, 0x59, 0x66, 0x6f, 0x74, 0x7d,
    0xa1, 0xa8, 0xb3, 0xba, 0x85, 0x8c, 0x97, 0x9e, 0xe9, 0xe0, 0xfb, 0xf2, 0xcd, 0xc4, 0xdf, 0xd6,
    0x31, 0x38, 0x23, 0x2a, 0x15, 0x1c, 0x07, 0x0e, 0x79, 0x70, 0x6b, 0x62, 0x5d, 0x54, 0x4f, 0x46
};

byte mul11[256] =
{
    0x00, 0x0b, 0x16, 0x1d, 0x2c, 0x27, 0x3a, 0x31, 0x58, 0x53, 0x4e, 0x45, 0x74, 0x7f, 0x62, 0x69,
    0xb0, 0xbb, 0xa6, 0xad, 0x9c, 0x97, 0x8a, 0x81, 0xe8, 0xe3, 0xfe, 0xf5, 0xc4, 0xcf, 0xd2, 0xd9,
    0x7b, 0x70, 0x6d, 0x66, 0x57, 0x5c, 0x41, 0x4a, 0x23, 0x28, 0x35, 0x3e, 0x0f, 0x04, 0x19, 0x12,
    0xcb, 0xc0, 0xdd, 0xd6, 0xe7, 0xec, 0xf1, 0xfa, 0x93, 0x98, 0x85, 0x8e, 0xbf, 0xb4, 0xa9, 0xa2,
    0xf6, 0xfd, 0xe0, 0xeb, 0xda, 0xd1, 0xcc, 0xc7, 0xae, 0xa5, 0xb8, 0xb3, 0x82, 0x89, 0x94, 0x9f,
    0x46, 0x4d, 0x50, 0x5b, 0x6a, 0x61, 0x7c, 0x77, 0x1e, 0x15, 0x08, 0x03, 0x32, 0x39, 0x24, 0x2f,
    0x8d, 0x86, 0x9b, 0x90, 0xa1, 0xaa, 0xb7, 0xbc, 0xd5, 0xde, 0xc3, 0xc8, 0xf9, 0xf2, 0xef, 0xe4,
    0x3d, 0x36, 0x2b, 0x20, 0x11, 0x1a, 0x07, 0x0c, 0x65, 0x6e, 0x73, 0x78, 0x49, 0x42, 0x5f, 0x54,
    0xf7, 0xfc, 0xe1, 0xea, 0xdb, 0xd0, 0xcd, 0xc6, 0xaf, 0xa4, 0xb9, 0xb2, 0x83, 0x88, 0x95, 0x9e,
    0x47, 0x4c, 0x51, 0x5a, 0x6b, 0x60, 0x7d, 0x76, 0x1f, 0x14, 0x09, 0x02, 0x33, 0x38, 0x25, 0x2e,
    0x8c, 0x87, 0x9a, 0x91, 0xa0, 0xab, 0xb6, 0xbd, 0xd4, 0xdf, 0xc2, 0xc9, 0xf8, 0xf3, 0xee, 0xe5,
    0x3c, 0x37, 0x2a, 0x21, 0x10, 0x1b, 0x06, 0x0d, 0x64, 0x6f, 0x72, 0x79, 0x48, 0x43, 0x5e, 0x55,
    0x01, 0x0a, 0x17, 0x1c, 0x2d, 0x26, 0x3b, 0x30, 0x59, 0x52, 0x4f, 0x44, 0x75, 0x7e, 0x63, 0x68,
    0xb1, 0xba, 0xa7, 0xac, 0x9d, 0x96, 0x8b, 0x80, 0xe9, 0xe2, 0xff, 0xf4, 0xc5, 0xce, 0xd3, 0xd8,
    0x7a, 0x71, 0x6c, 0x67, 0x56, 0x5d, 0x40, 0x4b, 0x22, 0x29, 0x34, 0x3f, 0x0e, 0x05, 0x18, 0x13,
    0xca, 0xc1, 0xdc, 0xd7, 0xe6, 0xed, 0xf0, 0xfb, 0x92, 0x99, 0x84, 0x8f, 0xbe, 0xb5, 0xa8, 0xa3
};

byte mul13[256] =
{
    0x00, 0x0d, 0x1a, 0x17, 0x34, 0x39, 0x2e, 0x23, 0x68, 0x65, 0x72, 0x7f, 0x5c, 0x51, 0x46, 0x4b,
    0xd0, 0xdd, 0xca, 0xc7, 0xe4, 0xe9, 0xfe, 0xf3, 0xb8, 0xb5, 0xa2, 0xaf, 0x8c, 0x81, 0x96, 0x9b,
    0xbb, 0xb6, 0xa1, 0xac, 0x8f, 0x82, 0x95, 0x98, 0xd3, 0xde, 0xc9, 0xc4, 0xe7, 0xea, 0xfd, 0xf0,
    0x6b, 0x66, 0x71, 0x7c, 0x5f, 0x52, 0x45, 0x48, 0x03, 0x0e, 0x19, 0x14, 0x37, 0x3a, 0x2d, 0x20,
    0x6d, 0x60, 0x77, 0x7a, 0x59, 0x54, 0x43, 0x4e, 0x05, 0x08, 0x1f, 0x12, 0x31, 0x3c, 0x2b, 0x26,
    0xbd, 0xb0, 0xa7, 0xaa, 0x89, 0x84, 0x93, 0x9e, 0xd5, 0xd8, 0xcf, 0xc2, 0xe1, 0xec, 0xfb, 0xf6,
    0xd6, 0xdb, 0xcc, 0xc1, 0xe2, 0xef, 0xf8, 0xf5, 0xbe, 0xb3, 0xa4, 0xa9, 0x8a, 0x87, 0x90, 0x9d,
    0x06, 0x0b, 0x1c, 0x11, 0x32, 0x3f, 0x28, 0x25, 0x6e, 0x63, 0x74, 0x79, 0x5a, 0x57, 0x40, 0x4d,
    0xda, 0xd7, 0xc0, 0xcd, 0xee, 0xe3, 0xf4, 0xf9, 0xb2, 0xbf, 0xa8, 0xa5, 0x86, 0x8b, 0x9c, 0x91,
    0x0a, 0x07, 0x10, 0x1d, 0x3e, 0x33, 0x24, 0x29, 0x62, 0x6f, 0x78, 0x75, 0x56, 0x5b, 0x4c, 0x41,
    0x61, 0x6c, 0x7b, 0x76, 0x55, 0x58, 0x4f, 0x42, 0x09, 0x04, 0x13, 0x1e, 0x3d, 0x30, 0x27, 0x2a,
    0xb1, 0xbc, 0xab, 0xa6, 0x85, 0x88, 0x9f, 0x92, 0xd9, 0xd4, 0xc3, 0xce, 0xed, 0xe0, 0xf7, 0xfa,
    0xb7, 0xba, 0xad, 0xa0, 0x83, 0x8e, 0x99, 0x94, 0xdf, 0xd2, 0xc5, 0xc8, 0xeb, 0xe6, 0xf1, 0xfc,
```

```
    0x67, 0x6a, 0x7d, 0x70, 0x53, 0x5e, 0x49, 0x44, 0x0f, 0x02, 0x15, 0x18, 0x3b, 0x36, 0x21, 0x2c,
    0x0c, 0x01, 0x16, 0x1b, 0x38, 0x35, 0x22, 0x2f, 0x64, 0x69, 0x7e, 0x73, 0x50, 0x5d, 0x4a, 0x47,
    0xdc, 0xd1, 0xc6, 0xcb, 0xe8, 0xe5, 0xf2, 0xff, 0xb4, 0xb9, 0xae, 0xa3, 0x80, 0x8d, 0x9a, 0x97
};

byte mul14[256] =
{
    0x00, 0x0e, 0x1c, 0x12, 0x38, 0x36, 0x24, 0x2a, 0x70, 0x7e, 0x6c, 0x62, 0x48, 0x46, 0x54, 0x5a,
    0xe0, 0xee, 0xfc, 0xf2, 0xd8, 0xd6, 0xc4, 0xca, 0x90, 0x9e, 0x8c, 0x82, 0xa8, 0xa6, 0xb4, 0xba,
    0xdb, 0xd5, 0xc7, 0xc9, 0xe3, 0xed, 0xff, 0xf1, 0xab, 0xa5, 0xb7, 0xb9, 0x93, 0x9d, 0x8f, 0x81,
    0x3b, 0x35, 0x27, 0x29, 0x03, 0x0d, 0x1f, 0x11, 0x4b, 0x45, 0x57, 0x59, 0x73, 0x7d, 0x6f, 0x61,
    0xad, 0xa3, 0xb1, 0xbf, 0x95, 0x9b, 0x89, 0x87, 0xdd, 0xd3, 0xc1, 0xcf, 0xe5, 0xeb, 0xf9, 0xf7,
    0x4d, 0x43, 0x51, 0x5f, 0x75, 0x7b, 0x69, 0x67, 0x3d, 0x33, 0x21, 0x2f, 0x05, 0x0b, 0x19, 0x17,
    0x76, 0x78, 0x6a, 0x64, 0x4e, 0x40, 0x52, 0x5c, 0x06, 0x08, 0x1a, 0x14, 0x3e, 0x30, 0x22, 0x2c,
    0x96, 0x98, 0x8a, 0x84, 0xae, 0xa0, 0xb2, 0xbc, 0xe6, 0xe8, 0xfa, 0xf4, 0xde, 0xd0, 0xc2, 0xcc,
    0x41, 0x4f, 0x5d, 0x53, 0x79, 0x77, 0x65, 0x6b, 0x31, 0x3f, 0x2d, 0x23, 0x09, 0x07, 0x15, 0x1b,
    0xa1, 0xaf, 0xbd, 0xb3, 0x99, 0x97, 0x85, 0x8b, 0xd1, 0xdf, 0xcd, 0xc3, 0xe9, 0xe7, 0xf5, 0xfb,
    0x9a, 0x94, 0x86, 0x88, 0xa2, 0xac, 0xbe, 0xb0, 0xea, 0xe4, 0xf6, 0xf8, 0xd2, 0xdc, 0xce, 0xc0,
    0x7a, 0x74, 0x66, 0x68, 0x42, 0x4c, 0x5e, 0x50, 0x0a, 0x04, 0x16, 0x18, 0x32, 0x3c, 0x2e, 0x20,
    0xec, 0xe2, 0xf0, 0xfe, 0xd4, 0xda, 0xc8, 0xc6, 0x9c, 0x92, 0x80, 0x8e, 0xa4, 0xaa, 0xb8, 0xb6,
    0x0c, 0x02, 0x10, 0x1e, 0x34, 0x3a, 0x28, 0x26, 0x7c, 0x72, 0x60, 0x6e, 0x44, 0x4a, 0x58, 0x56,
    0x37, 0x39, 0x2b, 0x25, 0x0f, 0x01, 0x13, 0x1d, 0x47, 0x49, 0x5b, 0x55, 0x7f, 0x71, 0x63, 0x6d,
    0xd7, 0xd9, 0xcb, 0xc5, 0xef, 0xe1, 0xf3, 0xfd, 0xa7, 0xa9, 0xbb, 0xb5, 0x9f, 0x91, 0x83, 0x8d
};

void shift_sub_rcon(byte *in, byte i) {
    byte t = in[0];
    in[0] = in[1];
    in[1] = in[2];
    in[2] = in[3];
    in[3] = t;

    in[0] = sbox[in[0]];
    in[1] = sbox[in[1]];
    in[2] = sbox[in[2]];
    in[3] = sbox[in[3]];

    in[0] ^= rcon[i];
}

void KeyExpansion(byte inputKey[16], byte expandedKeys[176]) {

    for (int i = 0; i < 16; i++) {
        expandedKeys[i] = inputKey[i];
    }

    int bytesGenerated = 16;
    int rconIteration = 1;
    byte tmpCore[4];

    while (bytesGenerated < 176) {
        for (int i = 0; i < 4; i++) {
            tmpCore[i] = expandedKeys[i + bytesGenerated - 4];
```

```
        }

        if (bytesGenerated % 16 == 0) {
            shift_sub_rcon(tmpCore, rconIteration++);
        }

        for (int a = 0; a < 4; a++) {
            expandedKeys[bytesGenerated] = expandedKeys[bytesGenerated - 16] ^ tmpCore[a];
            bytesGenerated++;
        }
    }
}

/*****************************************************************
* Pretty Printing Hex Values
*****************************************************************/
byte hexmap[] = {'0', '1', '2', '3', '4', '5', '6', '7',
                 '8', '9', 'a', 'b', 'c', 'd', 'e', 'f'};

std::string hex(byte *data, int len)
{
    std::string s(len * 3, ' ');
    for (int i = 0; i < len; ++i) {
        s[3 * i]     = hexmap[(data[i] & 0xF0) >> 4];
        s[3 * i + 1] = hexmap[data[i] & 0x0F];
        s[3 * i + 2] = ' ';
    }
    return s;
}

void AddRoundKey(byte *state, byte *RoundKey) {
    for(int i = 0; i < 16; i++) {
        state[i] ^= RoundKey[i];
    }
}

void SubBytes(byte *state) {
    for(int i = 0; i < 16; i++) {
        state[i] = sbox[state[i]];
    }
}

void ShiftRows(byte *state) {
    byte temp;

    temp = state[1];
    state[1] = state[5];
    state[5] = state[9];
    state[9] = state[13];
    state[13] = temp;

    temp = state[2];
    state[2] = state[10];
```

```
    state[10] = temp;
    temp = state[6];
    state[6] = state[14];
    state[14] = temp;

    temp = state[15];
    state[15] = state[11];
    state[11] = state[7];
    state[7] = state[3];
    state[3] = temp;
}

void MixColumns(byte *state) {
    byte temp[16];

    temp[0]  = (byte) mul2[state[0]] ^ mul3[state[1]] ^ state[2] ^ state[3];
    temp[1]  = (byte) state[0] ^ mul2[state[1]] ^ mul3[state[2]] ^ state[3];
    temp[2]  = (byte) state[0] ^ state[1] ^ mul2[state[2]] ^ mul3[state[3]];
    temp[3]  = (byte) mul3[state[0]] ^ state[1] ^ state[2] ^ mul2[state[3]];

    temp[4]  = (byte) mul2[state[4]] ^ mul3[state[5]] ^ state[6] ^ state[7];
    temp[5]  = (byte) state[4] ^ mul2[state[5]] ^ mul3[state[6]] ^ state[7];
    temp[6]  = (byte) state[4] ^ state[5] ^ mul2[state[6]] ^ mul3[state[7]];
    temp[7]  = (byte) mul3[state[4]] ^ state[5] ^ state[6] ^ mul2[state[7]];

    temp[8]  = (byte) mul2[state[8]] ^ mul3[state[9]] ^ state[10] ^ state[11];
    temp[9]  = (byte) state[8] ^ mul2[state[9]] ^ mul3[state[10]] ^ state[11];
    temp[10] = (byte) state[8] ^ state[9] ^ mul2[state[10]] ^ mul3[state[11]];
    temp[11] = (byte) mul3[state[8]] ^ state[9] ^ state[10] ^ mul2[state[11]];

    temp[12] = (byte) mul2[state[12]] ^ mul3[state[13]] ^ state[14] ^ state[15];
    temp[13] = (byte) state[12] ^ mul2[state[13]] ^ mul3[state[14]] ^ state[15];
    temp[14] = (byte) state[12] ^ state[13] ^ mul2[state[14]] ^ mul3[state[15]];
    temp[15] = (byte) mul3[state[12]] ^ state[13] ^ state[14] ^ mul2[state[15]];

    for (int i = 0; i < 16; i++) {
        state[i] = temp[i];
    }
}

void Round(byte *state, byte *RoundKey, bool isFinal=false) {
    SubBytes(state);
    ShiftRows(state);
    if(!isFinal) MixColumns(state);
    AddRoundKey(state, RoundKey);
}

#endif
```

Appendix B - generate.cpp (generates the datasets to encrypt)

```
// Generate files.
```

```cpp
// Compilation: g++ generate.cpp -o generate -std=c++11
// Run: ./generate
// Run in the same directory as your parallel and serial files.

#include <iostream>
#include <fstream>
#include <vector>
#include <stdlib.h>
#include <string>
#include <errno.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <dirent.h>

#define SYSERROR()  errno
#define KEYLENGTH 16

using namespace std;

typedef unsigned char byte;

void make_dir(string path) {
   const int dir_err = mkdir(path.c_str() , S_IRWXU | S_IRWXG | S_IROTH | S_IXOTH);
   if (dir_err == -1)
   {
       cerr << "Error creating directory:" << path;
       exit(1);
   }
}

void check_make_dir(string path) {
   DIR* dir = opendir(path.c_str());

   if (dir) {
       closedir(dir);
   }
   else if(SYSERROR() = ENOENT) {
       make_dir(path);
   }
   else {
       cerr << "opendir failed";
       exit(1);
   }
}

void generate_file(string path, int n_bytes) {
   ofstream fout;
   fout.open(path, ios::binary);

   if(fout.is_open()){
       int i;
       byte b;
       for(i = 0; i < n_bytes; i++) {
```

```
            b = rand()%256;
            fout << b;
        }
        fout.close();
    }
    else {
        cerr<<"Failed to open file : "<< SYSERROR() << std::endl;
        exit(1);
    }

    fout.open(path + "_key", ios::binary);
    if(fout.is_open()){
        int i;
        byte b;
        for(i = 0; i < KEYLENGTH; i++) {
            b = rand()%256;
            fout << b;
        }
        fout.close();
    }
    else {
        cerr<<"Failed to open file : "<< SYSERROR() << std::endl;
        exit(1);
    }
}

void generate(int start, int end, int step, int runs, int minlength, int maxlength, string data_path) {
    int i, j, k, n_bytes;
    string path;
    for(i = start; i <= end; i += step) {
        for (j = 0; j < runs; j++) {
            for(k = 0; k < i; k++) {
                path = data_path + "/" + to_string(i);
                check_make_dir(path);
                path = path + "/" + to_string(j);
                check_make_dir(path);
                path = path + "/" + to_string(k);
                cout << path << endl;
                n_bytes = minlength + rand()%(maxlength - minlength + 1);
                n_bytes -= n_bytes%16;
                generate_file(path, n_bytes);
            }
        }
    }
}

int main(int argc, char const *argv[])
{
    int Snumfiles = 100;    // Number of files to start encrypting
    int Enumfiles = 300;    // Number of files to stop encrypting
    int Stepfiles = 100;    // Number of files to step
    int runs = 5;           // Number of runs to do each time
    std::string data_path = "dataset";  // Data path
```

```
    int minlength = 1024;               // Minimum size of file
    int maxlength = 1024*32;            // Maximum size of file

    srand(time(NULL));
    check_make_dir(data_path);
    generate(Snumfiles, Enumfiles, Stepfiles, runs, minlength, maxlength, data_path);

    return 0;
}
```

Appendix C - parallel.cpp (contains the code for encrypting with OpenMP)

```
// Parallel version of AES.
// Compilation: g++ parallel.cpp -o parallel -fopenmp -std=c++11
// Run: ./parallel

#include "/home/uahclsc0003/source/Project/AES/AES/include/aeslib.hpp"
using namespace std;
#define N_ROUNDS 10

void encrypt(vector<byte *> &data, vector<int> &length, vector<byte *> &key, vector<byte *> &ciphers);
void get_data(string path, vector<byte*> &msgs, vector<int> &lens, vector<byte*> &keys, int i, int j);

int main() {
    int Snumfiles = 100;    // Number of files to start encrypting
    int Enumfiles = 300;    // Number of files to stop encrypting
    int Stepfiles = 100;    // Number of files to step
    int runs = 5;           // Number of runs to do each time
    std::string data_path = "dataset";
    cout << "========== STARTING AES ENCRYPTION PARALLEL ==========" << endl;
    for(int i = Snumfiles; i <= Enumfiles; i += Stepfiles) {    // Loop through start and end number of files
        for(int j = 0; j < runs; j++) {                         // Number of runs to do on each file
            vector<byte*> data, key, ciphers;                   // Allocate space for data, key, and ciphers.
            vector<int> length;                                 // Length of message

            get_data(data_path, data, length, key, i, j);   // Get the data
            ciphers.reserve(i);     // Reserve space for the cipher

            auto s = chrono::high_resolution_clock::now();  // Start clock
            encrypt(data, length, key, ciphers);            // Encrypt the data
            auto e = chrono::high_resolution_clock::now();  // End clock

            string out_path;
            ofstream fout;
            // Write the ciphers to memory. This is important for moving onto the next run.
            for(int k = 0; k < i; k++) {
                out_path = data_path + "/" + to_string(i) + "/" + to_string(j) + "/" + to_string(k) +
"_cipher_encrypt";
                fout.open(out_path, ios::binary);
                fout.write(reinterpret_cast<char *> (ciphers[k]), length[k]);
                fout.close();
                delete[] data[k];   // Remove for extra space
```

```
            delete[] key[k];
        }
        auto _time = chrono::duration_cast<chrono::milliseconds>(e - s);    // Time calculation
        cout << "Files encrypted: " << i << " |Batch #" << j << "| Time To Encrypt " << _time.count() <<
"ms " << endl; // Output results.
      }
   }
   cout << "=========== ENDING AES ENCRYPTION PARALLEL ===========" << endl;

   return 0;
}

void encrypt(vector<byte *> &data, vector<int> &length, vector<byte *> &key, vector<byte *> &ciphers){
   int n;                      // Length of message in loop
   byte expandedKey[176];      // Allocate space for the expanded key...
   for(int i = 0; i < data.size(); i++) {
       n = length[i];
       byte *cipher = new byte[n];             // space for cipher
       KeyExpansion(key[i], expandedKey);      // run key expansion

       omp_set_num_threads(4);                 // number of threads
       #pragma omp parallel for                // open mp for loop
       for(int index = 0 ; index<length[i] ; index += 16){ // Loop through index
           AddRoundKey(data[i] + index , expandedKey);
           for(int n_rounds = 1 ; n_rounds<=10 ; ++n_rounds)
               Round(data[i] + index, expandedKey + (n_rounds*16), (n_rounds==10));
       }

       cipher = data[i];
       ciphers.push_back(move(cipher));
   }
}

void get_data(string path, vector<byte*> &msgs, vector<int> &lens, vector<byte*> &keys, int i, int j) {
   string mpath, kpath;                // message and key path
   ifstream input_message, input_key;  // input message and key
   int k, n;

   for(k = 0; k < i; k++) {
       mpath = path + "/" + to_string(i) + "/" + to_string(j) + "/" + to_string(k);
       kpath = mpath+"_key";

       input_message.open(mpath, ios::binary);
       input_key.open(kpath, ios::binary);

       if(input_message && input_key) {

           input_message.seekg(0, input_message.end);  // Look at the message
           n = input_message.tellg();                  // get length
           input_message.seekg(0, input_message.beg);
           byte *message = new byte[n];                // allocate space for message
           byte *key = new byte[16];                   // allocate space for key
```

```
            input_message.read( reinterpret_cast<char *> (message), n); // Read in message
            input_key.read( reinterpret_cast<char *> (key), 16);        // Read in key

            msgs.push_back(move(message));
            lens.push_back(n);
            keys.push_back(move(key));

            input_message.close();
            input_key.close();
        }
    }
}
```

Appendix D - serial.cpp (contains the code for encrypting serially)

```
// Serial version of AES.
// Compilation: g++ serial.cpp -o serial -fopenmp -std=c++11
// Run: ./serial

#include "/home/uahclsc0003/source/Project/AES/AES/include/aeslib.hpp"
using namespace std;
#define N_ROUNDS 10

void Cipher(byte *message, int msg_length, byte expandedKey[176], byte *cipher);
void encrypt(string mpath, string kpath, string opath);

int main(int argc, char const *argv[])
{
    int Snumfiles = 100;    // Number of files to start encrypting
    int Enumfiles = 300;    // Number of files to stop encrypting
    int Stepfiles = 100;    // Number of files to step
    int runs = 5;           // Number of runs to do each time
    std::string data_path = "dataset";
    string path;
    cout << "========== STARTING AES ENCRYPTION SERIALLY ==========" << endl;
    for(int i = Snumfiles; i <= Enumfiles; i += Stepfiles) {    // Loop through number of files
        for (int j = 0; j < runs; j++) {                        // Number of runs to do each time
            auto s = chrono::high_resolution_clock::now();      // Start the clock
            for(int k = 0; k < i; k++) {                        // Grab data paths and start calculation
                path = data_path + "/" + to_string(i) + "/" + to_string(j) + "/" + to_string(k);
                encrypt(path, path+"_key", path+"_cipher_seq");
            }
            auto e = chrono::high_resolution_clock::now();      // Stop the clock
            auto _time = chrono::duration_cast<chrono::milliseconds>(e - s);    // Quick time calculation
            cout << "Files encrypted: " << i << " |Batch #" << j << "| Time To Encrypt " << _time.count() <<
"ms " << endl; // Output results.
        }
    }
    cout << "========== ENDING AES ENCRYPTION SERIALLY ==========" << endl;
    return 0;
}
```

```
void Cipher(byte *message, int msg_length, byte expandedKey[176], byte *cipher) {
   for(int i = 0; i < msg_length; i++) cipher[i] = message[i]; // cipher is now equal to message
   for(int i = 0; i < msg_length; i += 16) {
       AddRoundKey(cipher + i, expandedKey);
       for(int n = 1; n <= N_ROUNDS; n++) Round(cipher + i, expandedKey + (n)*16, n == 10);
   }
}

void encrypt(string mpath, string kpath, string opath) {
   ifstream f_msg(mpath, ios::binary); // Incoming message
   ifstream f_key(kpath, ios::binary); // Incoming key
   ofstream fout(opath, ios::binary);  // Outgoing message (i.e. the cipherthex)

   if(f_msg && f_key) {
       f_msg.seekg(0, f_msg.end);  // Seek through the message
       int n = f_msg.tellg();      // Get size of incoming message
       f_msg.seekg(0, f_msg.beg);

       byte message[n], cipher[n]; // Allocate the message and cipher of size n
       byte key[16];               // Allocate the key
       byte expandedKey[176];      // Allocate for the expanded key

       f_msg.read(reinterpret_cast<char *> (message), n);  // Read in message
       f_key.read(reinterpret_cast<char *> (key), 16);     // Read in key

       KeyExpansion(key, expandedKey);                     // Perform the AES key expansion
       Cipher(message, n, expandedKey, cipher);            // Compute the cipher portion.
       fout.write(reinterpret_cast<char *> (cipher), n);   // Write to the cipher for next round.
   }
}
```

Appendix E - CUDA stuff

```
#include <aes.h>
#include <helper_cuda.h>
#define TILE_WIDTH 16
#define COARSE_FACTOR 4

__global__ void AESEncrypt_Kernel(){

}

__global__ void AESDecrypt_Kernel(){

}
```

```
void AES_Encrypt_CUDA(){
 int blockSize;
 int minGridSize;
 checkCudaErrors(cudaOccupancyMaxPotentialBlockSize(&minGridSize, &blockSize,
MatrixMulKernel, 0, 0));

 dim3 numThreadsPerBlock(16, 16);
 dim3 numBlocks((Size + numThreadsPerBlock.x - 1)/numThreadsPerBlock.x,
                (Size + numThreadsPerBlock.y - 1)/numThreadsPerBlock.y);

 AESEncrypt_Kernel<<<numThreadsPerBlock, numBlocks>>>();
}

void AES_Decrypt_CUDA(){
 int blockSize;
 int minGridSize;
 checkCudaErrors(cudaOccupancyMaxPotentialBlockSize(&minGridSize, &blockSize,
MatrixMulKernel, 0, 0));

 dim3 numThreadsPerBlock(16, 16);
 dim3 numBlocks((Size + numThreadsPerBlock.x - 1)/numThreadsPerBlock.x,
                (Size + numThreadsPerBlock.y - 1)/numThreadsPerBlock.y);

 AESDecrypt_Kernel<<<numThreadsPerBlock, numBlocks>>>();
}
/*
   Nolan Anderson
   CPE 613 Final Project
   This program compares the the run times and performance metrics between serial,
openMP, and CUDA implementations
   of AES encryption / decryption.

   To run / compile:
   module load cuda && module load blas
   make run
*/

#include <aes.h>
#include <Timer.hpp>
#include <cuda_runtime.h>
#include <helper_cuda.h>
```

```
#include <cublas_v2.h>
#include <cmath>
#include <cstdio>
#include <vector>
#include <stdio.h>
#include <typeinfo>
#include <iostream>

using namespace std;

Timer AES_Serial_Encryption();
Timer AES_Serial_Decryption();
Timer AES_Parallel_Encryption();
Timer AES_Parallel_Decryption();
Timer AES_CUDA_Encryption();
Timer AES_CUDA_Decryption();

int main (int argc, char ** argv){
 Timer time;
 cout << "========== STARTING AES ENCRYPTION SERIAL ==========" << endl;
 time = AES_Serial_Encryption();
 cout << "========== ENDING AES ENCRYPTION SERIAL ==========" << endl;

 cout << "========== STARTING AES DECRYPTION SERIAL ==========" << endl;
 time = AES_Serial_Decryption();
 cout << "========== ENDING AES DECRYPTION SERIAL ==========" << endl;
  cout << "========== STARTING AES ENCRYPTION OPENMP ==========" << endl;
 time = ES_Parallel_Encryption();
 cout << "========== ENDING AES ENCRYPTION OPENMP ==========" << endl;

 cout << "========== STARTING AES DECRYPTION OPENMP ==========" << endl;
 time = AES_Parallel_Decryption();
 cout << "========== ENDING AES DECRYPTION OPENMP ==========" << endl;

 cout << "========== STARTING AES ENCRYPTION CUDA ==========" << endl;
 time = AES_CUDA_Encryption();
 cout << "========== ENDING AES ENCRYPTION CUDA ==========" << endl;

 cout << "========== STARTING AES DECRYPTION CUDA ==========" << endl;
 time = AES_CUDA_Decryption();
 cout << "========== ENDING AES DECRYPTION CUDA ==========" << endl;
```

```
 return 0;
}

Timer AES_Serial_Encryption(){
 Timer time;
 return time;
}

Timer AES_Serial_Decryption(){
 Timer time;
 return time;
}

Timer AES_Parallel_Encryption(){
 Timer time;
 return time;
}

Timer AES_Parallel_Decryption(){
 Timer time;
 return time;
}

Timer AES_CUDA_Encryption(){
 Timer time;
 return time;
}

Timer AES_CUDA_Decryption(){
 Timer time;
 return time;
}
```