Lab 2: How to implement an AES in Software

1- Goal:

In this set of Labs, you are going to learn how to improve the performance of a program and how to accelerate it using special hardware available in DevCloud. You will learn what the main differences between software and hardware implementations of a solution are, and how you can overcome the traditional difficulties in speeding up a program on different platforms. By using DPC++, oneAPI toolkit, and various hardware available on DevCloud, you will learn how much it is easy to write a program in one language (DPC++) and run it on CPU, FPGA, and GPU without any special modification, and be able to offload compute intensive tasks on hardware accelerators.

For that purpose, the first step is to solve problem in software and then in hardware. After than we can move forward to solve the problem with DPC++ and compare those approaches. In this lab, we start with solving a problem in software.

In this lab, you will learn how to understand a problem from its high-level description, and how to convert the solution (algorithm) of the problem into a software workload for computer to solve.

2- Scenario:

Every program solves a problem. For example, a tax return program solves the problem of organizing and filing your taxes. A program is only as useful as the problem it solves. Most programs simplify and automate an existing problem. The goal of any program is to make a specific task faster, easier, and more convenient. The only way reaches that goal is to identify what task your program is trying to solve in the first place.

We select AES (Advanced Encryption Standard) as a main workload for this lab. AES is a popular, well-documented algorithm where you can identify its performance hotspot components to be completed in parallel and the part that must be completed serially. AES is based on a design principle known as a substitution–permutation network. It is efficient in both software and hardware, and can be implemented in both CPU, GPU, and FPGA. As AES is an important part of any cryptography application, it is an important example to be analyzed in terms of performance, throughput, area, and power. The detailed description, specification, and schematic of AES will be provided for you in this lab.

AES has four primary components that you must implement as single thread in C++. You are required to verify the functionality of the code and will be asked to optimize the algorithm and code as necessary for performance improvement.

3- Steps to Writing a Program

The general steps for writing a program include the following:

1- Understand the problem (Defining the Problem)
2- Design a solution
3- Draw a flow chart or Write pseudo-code
4- Write code
5- Test and debug
6- Release program
7- Iterate the steps for the next version

The task of defining the problem consists of identifying what it is you know (input-given data), and what it is you want to obtain (output-the result). For designing the solution, you need to figure out what functionality is required to map your input to the output. For each function, then you can plan a solution. Two common ways of planning the solution to a problem are to draw a flowchart and to write pseudocode, or possibly both. Pseudocode is an English-like nonstandard language that lets you state your solution with more precision than you can in plain English but with less precision than is required when using a formal programming language. Pseudocode permits you to focus on the program logic without having to be concerned just yet about the precise syntax of a particular programming language. However, pseudocode is not executable on the computer. As the programmer, your next step is to code the program-that is, to express your solution in a programming language. You will translate the logic from the flowchart or pseudocode-or some other tool-to a programming language. Some experts insist that a well-designed program can be written correctly the first time. In fact, they assert that there are mathematical ways to prove that a program is correct. However, the imperfections of the world are still with us, so most programmers get used to the idea that their newly written programs probably have a few errors. A term used extensively in programming, debugging means detecting, locating, and correcting bugs (mistakes), usually by running the program. These bugs are logic errors, such as telling a computer to repeat an operation but not telling it how to stop repeating. In this phase you run the program using test data that you devise. You must plan the test data carefully to make sure you test every part of the program.

Documentation is a written detailed description of the programming cycle and specific facts about the program. Typical program documentation materials include the origin and nature of the problem, a brief narrative description of the program, logic tools such as flowcharts and pseudocode, data-record descriptions, program listings, and testing results. Comments in the program itself are also considered an essential part of documentation.

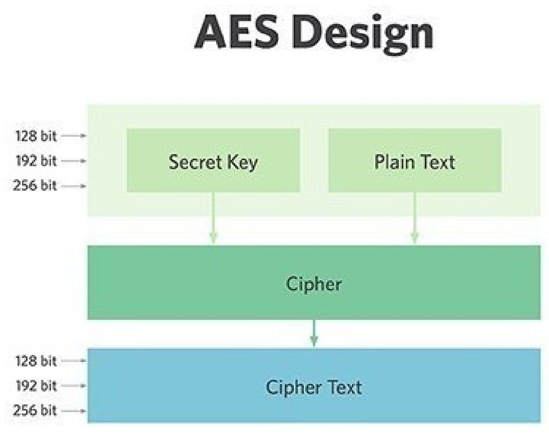
4- AES (Advanced Encryption Standard)

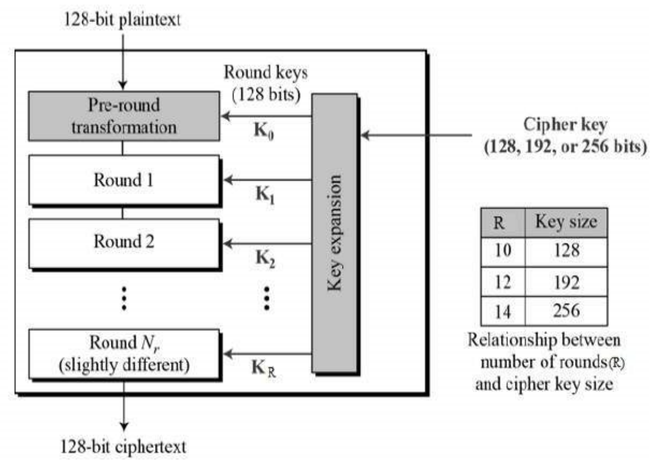
The Advanced Encryption Standard (AES) is a symmetric block cipher chosen by the U.S. government to protect classified information. AES is implemented in software and hardware throughout the world to encrypt sensitive data. It is essential for government computer security, cybersecurity and electronic data protection.

AES includes three block ciphers: AES-128, AES-192 and AES-256. AES-128 uses a 128-bit key length to encrypt and decrypt a block of messages, while AES-192 uses a 192-bit key length and AES-256 a 256-bit key length to encrypt and decrypt messages. Each cipher encrypts and decrypts data in blocks of 128 bits using cryptographic keys of 128, 192 and 256 bits, respectively. Symmetric, also known as secret key, ciphers use the same key for encrypting and decrypting, so the sender and the receiver must both know -- and use -- the same secret key.

In this lab, we look at AES-128. There are 10 rounds for 128-bit keys. A round consists of several processing steps that include substitution, transposition and mixing of the input plaintext to transform it into the final output of ciphertext.

The schematic of AES structure is given in the following:





R	Key size
10	128
12	192
14	256

5- Encryption Process

Here, we restrict to description of a typical round of AES encryption. Each round comprise of four sub-processes.

High-level description of the algorithm:

1. KeyExpansion – round keys are derived from the cipher key using the AES key schedule. AES requires a separate 128-bit round key block for each round plus one more.
2. Initial round key addition:
    1. AddRoundKey – each byte of the state is combined with a byte of the round key using bitwise xor.
3. 9, 11 or 13 rounds:
    1. SubBytes – a non-linear substitution step where each byte is replaced with another according to a lookup table.
    2. ShiftRows – a transposition step where the last three rows of the state are shifted cyclically a certain number of steps.
    3. MixColumns – a linear mixing operation which operates on the columns of the state, combining the four bytes in each column.
    4. AddRoundKey
4. Final round (making 10, 12 or 14 rounds in total):
    1. SubBytes
    2. ShiftRows
    3. AddRoundKey

Be aware that the following example is a simplification, but it gives you a general idea of how AES works.

Data division:

First, the data is divided into blocks. Under this method of encryption, the first thing that happens is that your plaintext (which is the information that you want to be encrypted) is separated into blocks. The block size of AES is 128-bits, so it separates the data into a four-by-four column of sixteen bytes (there are eight bits in a byte and 16 x 8 = 128). If your message was “buy me some potato chips please” the first block looks like this:

b	m	o	p
u	e	m	o
y		e	t
	s		a

The "…to chips please" would normally just be added to the next block.

Key expansion:

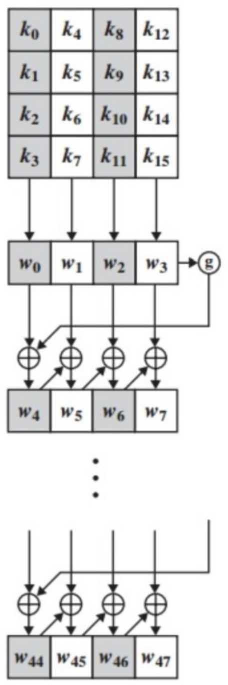
Key expansion involves taking the initial key and using it to come up with a series of other keys for each round of the encryption process. These new 128-bit round keys are derived with Rijndael's key schedule, which is essentially a simple and fast way to produce new key ciphers. Key expansion is a critical step, because it gives us our keys for the later rounds. Otherwise, the same key would be added in each round, which would make AES easier to crack. In the first round, the initial key is added in order to begin the alteration of the plain text.

Here we need to know three symbols: Nb: the number of columns (32-bit words) contained in the State (Nb=4), Nk: The number of 32-bit words contained in the key (Nk=4), Nr: the number of rounds encrypted (Nr=10).

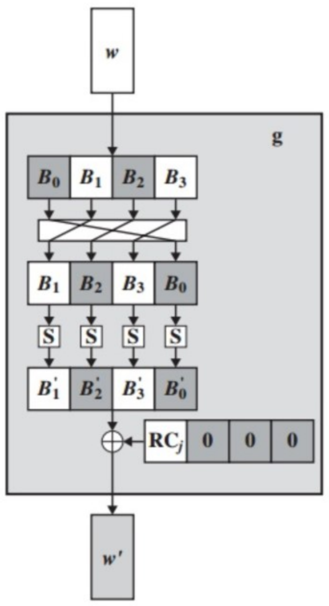
The AES key expansion algorithm takes as input a four-word (16-byte) key and produces a linear array of 44 words (176 bytes). This is sufficient to provide a four-word round key for the initial AddRoundKey stage and each of the 10 rounds of the cipher. The pseudocode describes the expansion.

```
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk)
begin
   word  temp

   i = 0

   while (i < Nk)
      w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
      i = i+1
   end while

   i = Nk

   while (i < Nb * (Nr+1)]
      temp = w[i-1]
      if (i mod Nk = 0)
         temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
      else if (Nk > 6 and i mod Nk = 4)
         temp = SubWord(temp)
      end if
      w[i] = w[i-Nk] xor temp
      i = i + 1
   end while
end
```

The key is copied into the first four words of the expanded key. The remainder of the expanded key is filled in four words at a time. Each added word w[i] depends on the immediately preceding word, w[i - 1], and the word four positions back, w[i - 4]. In three out of four cases, a simple XOR is used. For a word whose position in the *w* array is a multiple of 4, a more complex function is used.



(a) Overall algorithm



(b) Function g

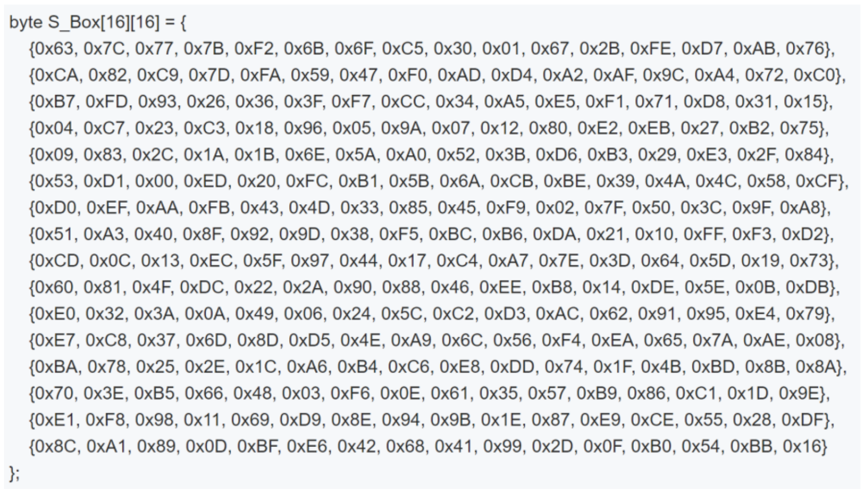
Above illustrates the generation of the expanded key, using the symbol g to represent that complex function. The function g consists of the following subfunctions.

1. RotWord() performs a one-byte circular left shift on a word. This means that an input word [B0, B1, B2, B3] is transformed into [B1, B2, B3, B0].

2. SubWord() performs a byte substitution on each byte of its input word, using the S-box table.

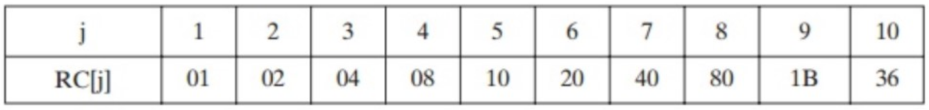
3. The result of steps 1 and 2 is XORed with a round constant, Rcon[j]. let alone directly regard it as a constant array

S-box transformation function SubWord(), accepts a word [a0, a1, a2, a3] as input. The S box is a 16x16 table, with each element being a byte. For each byte input, the first four bits constitute the hexadecimal number x as the line number, and the last four bits constitute the hexadecimal number y as the column number to find the corresponding values in the table. Finally, the function outputs a 32-bit word consisting of four new bytes.



```
byte S_Box[16][16] = {
    {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}
};
```

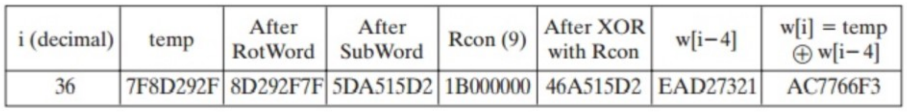
The round constant is a word in which the three rightmost bytes are always 0. Thus, the effect of an XOR of a word with Rcon is to only perform an XOR on the left-most byte of the word. The round constant is different for each round and is defined as Rcon[j] = (RC[j], 0, 0, 0), with RC[1] = 1, RC[j] = 2 RC[j -1] and with multiplication defined over the field GF(2^8). The values of RC[j] in hexadecimal are:



j	1	2	3	4	5	6	7	8	9	10
RC[j]	01	02	04	08	10	20	40	80	1B	36

For example, suppose that the round key for round 8 is:
EA D2 73 21 B5 8D BA D2 31 2B F5 60 7F 8D 29 2F

Then the first 4 bytes (first column) of the round key for round 9 are calculated as follows:



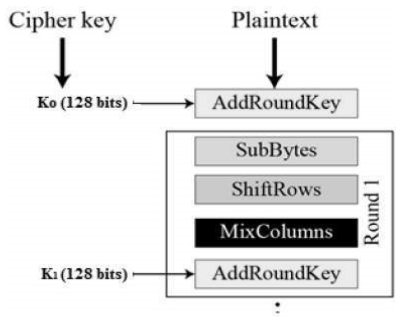
i (decimal)	temp	After RotWord	After SubWord	Rcon (9)	After XOR with Rcon	w[i−4]	w[i] = temp ⊕ w[i−4]
36	7F8D292F	8D292F7F	5DA515D2	1B000000	46A515D2	EAD27321	AC7766F3

Encryption:

The pseudocode of encryption is as follows:

```
Cipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
   byte  state[4,Nb]

   state = in

   AddRoundKey(state, w[0, Nb-1])

   for round = 1 step 1 to Nr-1
      SubBytes(state)
      ShiftRows(state)
      MixColumns(state)
      AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
   end for

   SubBytes(state)
   ShiftRows(state)
   AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])

   out = state
end
```

From the pseudocode description, we can see that each round comprise of four sub-processes such as SubBytes(), ShiftRows(), MixColumns(), and AdRoundKey(). The first-round process is depicted below:



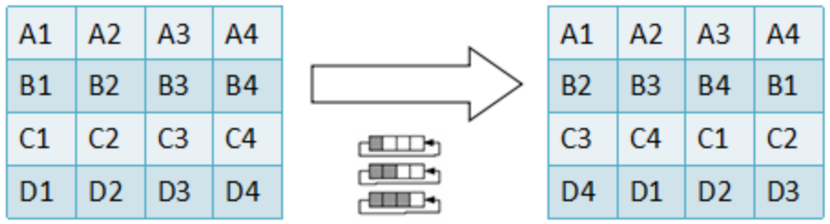
1- Byte Substitution (SubBytes):

The 16 input bytes are substituted by looking up a fixed table (S-box) given in design. The result is in a matrix of four rows and four columns.

2- Shiftrows

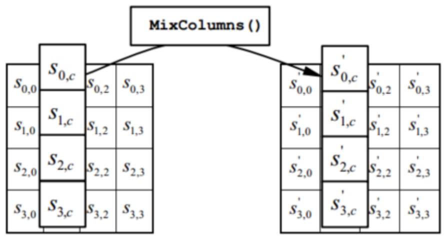
Each of the four rows of the matrix is shifted to the left. Any entries that ‘fall off’ are re-inserted on the right side of row. Shift is carried out as follows:

- First row is not shifted.
- Second row is shifted one (byte) position to the left.
- Third row is shifted two positions to the left.
- Fourth row is shifted three positions to the left.
- The result is a new matrix consisting of the same 16 bytes but shifted with respect to each other.



3- MixColumns

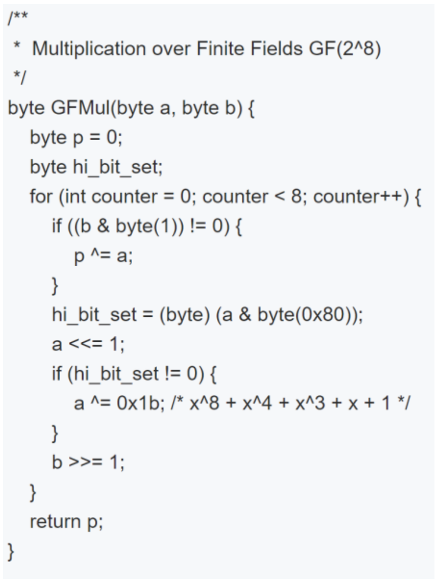
Each column of four bytes is now transformed using a special mathematical function. This function takes as input the four bytes of one column and outputs four completely new bytes, which replace the original column. The result is another new matrix consisting of 16 new bytes. It should be noted that this step is not performed in the last round.



$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad 0 \le c < 4$$

$$\begin{aligned} s'_{0,c} &= (\{02\} \bullet s_{0,c}) \oplus (\{03\} \bullet s_{1,c}) \oplus s_{2,c} \oplus s_{3,c} \\ s'_{1,c} &= s_{0,c} \oplus (\{02\} \bullet s_{1,c}) \oplus (\{03\} \bullet s_{2,c}) \oplus s_{3,c} \\ s'_{2,c} &= s_{0,c} \oplus s_{1,c} \oplus (\{02\} \bullet s_{2,c}) \oplus (\{03\} \bullet s_{3,c}) \\ s'_{3,c} &= (\{03\} \bullet s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \bullet s_{3,c}) \end{aligned}$$

Note that the multiplication used in the formula is Multiplication over Galois Fields (GF, Finite Fields) that we provide it here:



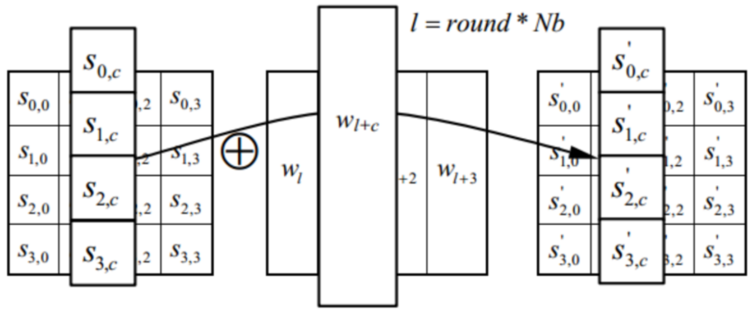
```
/**
 *  Multiplication over Finite Fields GF(2^8)
 */
byte GFMul(byte a, byte b) {
    byte p = 0;
    byte hi_bit_set;
    for (int counter = 0; counter < 8; counter++) {
        if ((b & byte(1)) != 0) {
            p ^= a;
        }
        hi_bit_set = (byte) (a & byte(0x80));
        a <<= 1;
        if (hi_bit_set != 0) {
            a ^= 0x1b; /* x^8 + x^4 + x^3 + x + 1 */
        }
        b >>= 1;
    }
    return p;
}
```

Multiplication over finite field GF(2^8) can be implemented by lookup table, and the encryption speed of AES will increase by more than 80%.

4- Addroundkey

The 16 bytes of the matrix are now considered as 128 bits and are XORed to the 128 bits of the round key. If this is the last round then the output is the ciphertext. Otherwise, the resulting 128 bits are interpreted as 16 bytes and we begin another similar round.

Extended keys are only involved in this step. According to the number of rounds currently encrypted, four extended keys in w [] are bitwise exclusive or with four columns of the matrix.



Okay, AES decryption is over here. It is easy to implement AES decryption algorithm based on pseudo-code after writing three functions of inverse transformation. But decryption is not necessary for this lab.

Assignment:

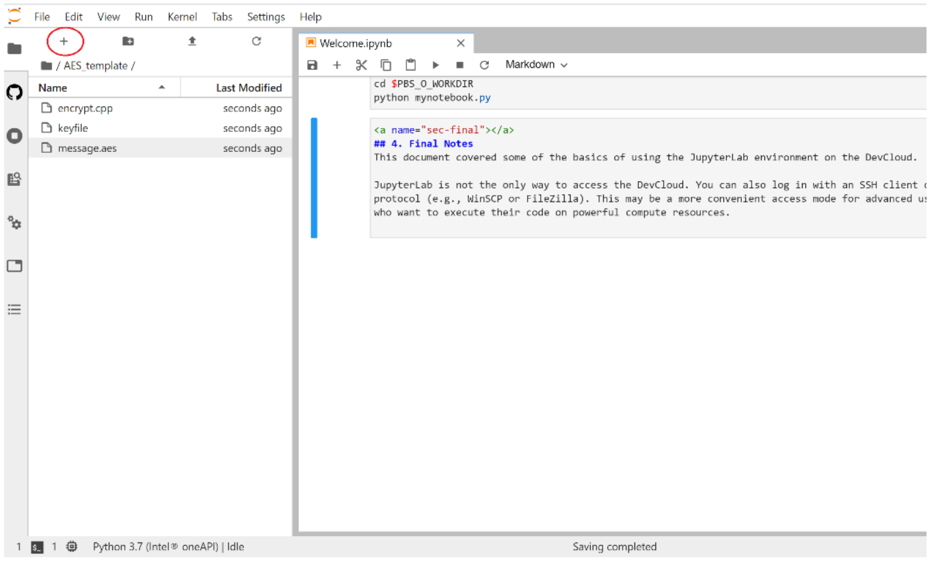
Please write a code that can perform AES-128 on an input string and write the result in a file.

The template file is provided in the lab_2 directory. Template file can read the key from a file.

Implement the Mixcolumns() in two ways (lookup table, and function) and compare the speedup of the AES.

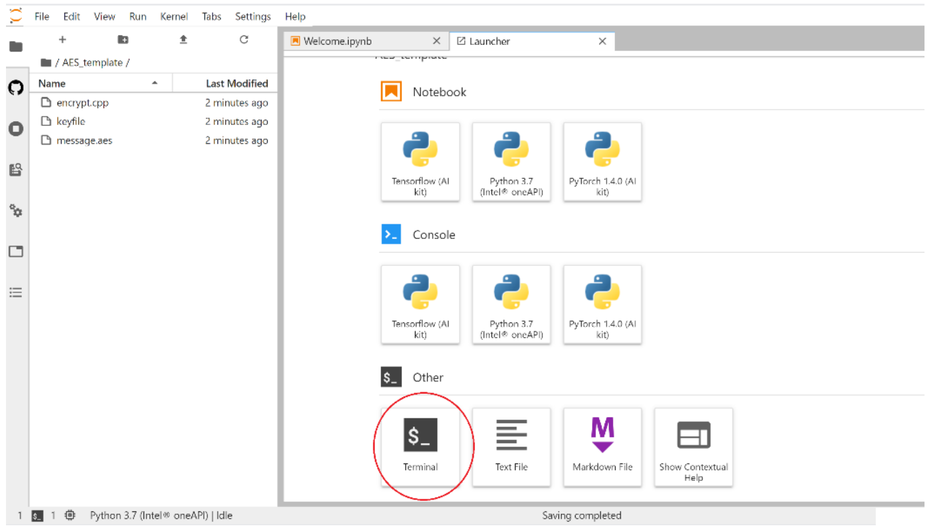
Follow Lab_1 instruction to upload the template files in Jupyter environment and then follow these instructions to compile and run your program:

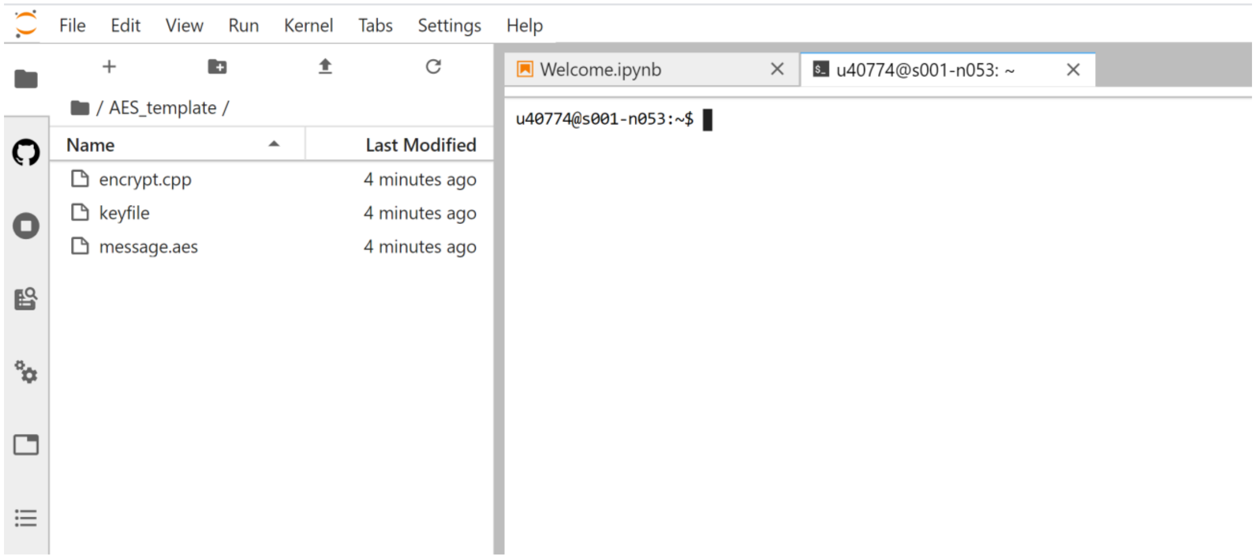
1- Open launcher (in the red circle below) to open a terminal window:



```
cd $PBS_O_WORKDIR
python mynotebook.py
```

```
<a name="sec-final"></a>
## 4. Final Notes
This document covered some of the basics of using the JupyterLab environment on the DevCloud.

JupyterLab is not the only way to access the DevCloud. You can also log in with an SSH client o
protocol (e.g., WinSCP or FileZilla). This may be a more convenient access mode for advanced us
who want to execute their code on powerful compute resources.
```



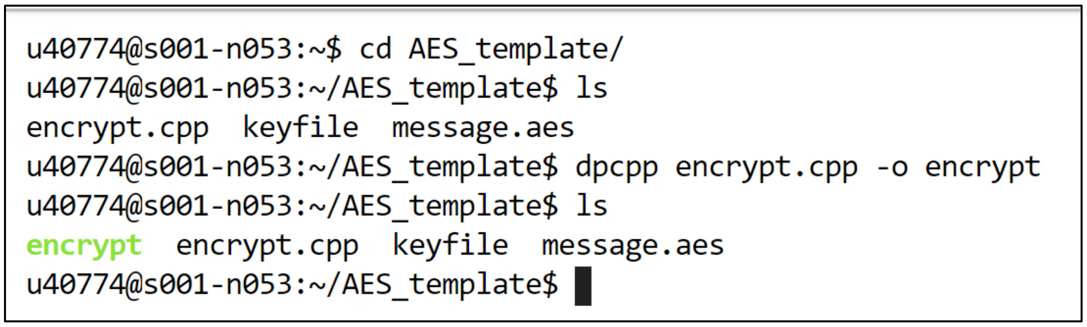


```
encrypt.cpp     4 minutes ago
keyfile         4 minutes ago
message.aes     4 minutes ago
```

```
u40774@s001-n053:~$
```

2- Similar to Linux environment, got to AES directory. Then run following command to compile the code and create the executable output:

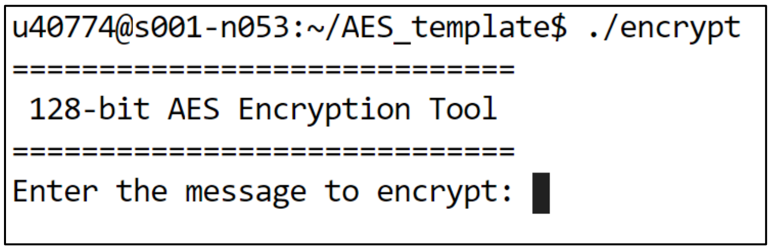
- $ dpcpp <name of file to be compiled> -o <name of output file>



```
u40774@s001-n053:~$ cd AES_template/
u40774@s001-n053:~/AES_template$ ls
encrypt.cpp  keyfile  message.aes
u40774@s001-n053:~/AES_template$ dpcpp encrypt.cpp -o encrypt
u40774@s001-n053:~/AES_template$ ls
encrypt  encrypt.cpp  keyfile  message.aes
u40774@s001-n053:~/AES_template$
```

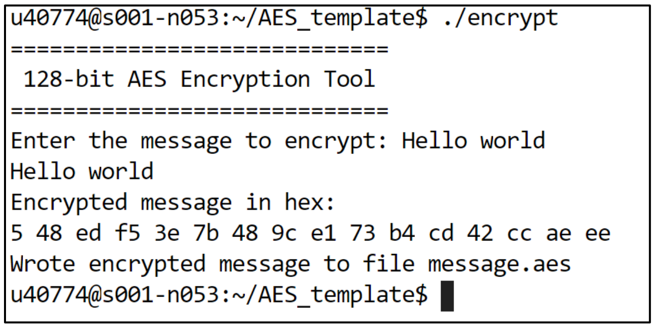
3- after compiling the code, you can see the output. Now, run the executable output with following command:

- $ ./encrypt



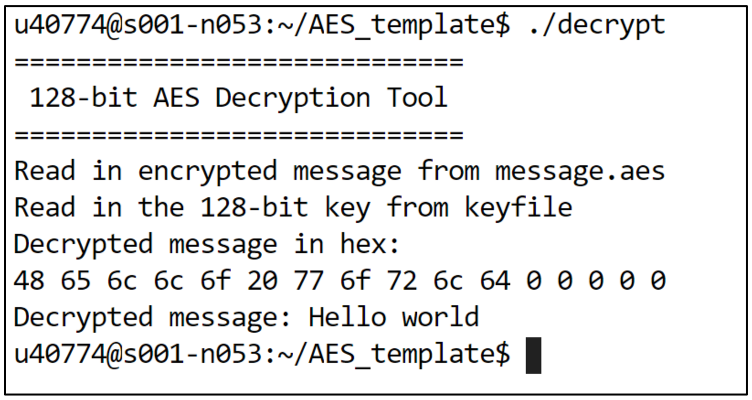
```
u40774@s001-n053:~/AES_template$ ./encrypt
==============================
 128-bit AES Encryption Tool
==============================
Enter the message to encrypt:
```

4- Type a message to encrypt:



```
u40774@s001-n053:~/AES_template$ ./encrypt
=============================
 128-bit AES Encryption Tool
=============================
Enter the message to encrypt: Hello world
Hello world
Encrypted message in hex:
5 48 ed f5 3e 7b 48 9c e1 73 b4 cd 42 cc ae ee
Wrote encrypted message to file message.aes
u40774@s001-n053:~/AES_template$ 
```

We will provide and executable decrypt file to check if your encryption is correct!



```
u40774@s001-n053:~/AES_template$ ./decrypt
=============================
 128-bit AES Decryption Tool
=============================
Read in encrypted message from message.aes
Read in the 128-bit key from keyfile
Decrypted message in hex:
48 65 6c 6c 6f 20 77 6f 72 6c 64 0 0 0 0 0
Decrypted message: Hello world
u40774@s001-n053:~/AES_template$ 
```