

Application Acceleration with High-Level Synthesis

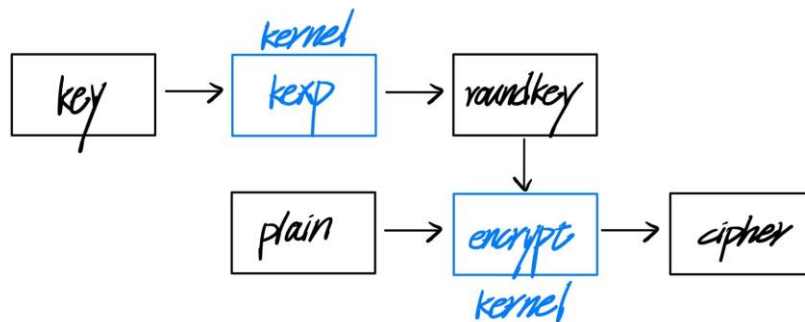
Final Report – AES-128 encryption

Team1 : 111061529 周子翔、111061545 陳揚哲

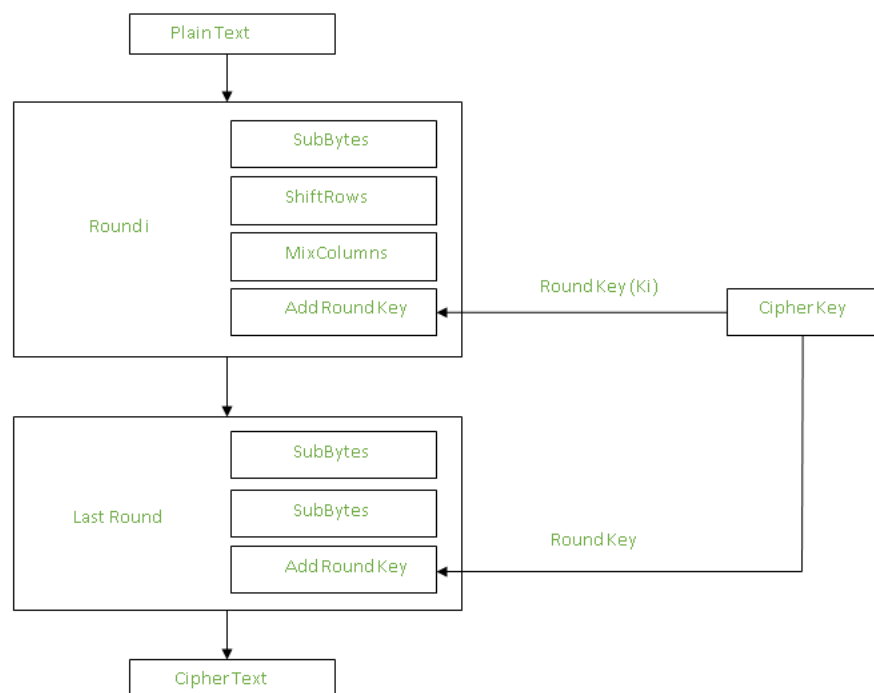
Github : https://github.com/ZheChen-Bill/HLS_FINAL

1. System block

Hardware block:



其中 Encrypt 會進行下圖中運算。



Software block:

我們的期末題目使用對稱式加密演算法(AES-128)套用至 FPGA 上進行執行，AES-128 的設計上包含 text 和 key 兩部分，原始的 text 會根據原始 key 的進行第一次加密，之後 key 會透過 key expansion 的過程計算出第二把 key，並將其對第一次加密後的 text 進行第二次加密。總共會進行 10 次的加密得到最後的 text 以及 key，因此最後總共使用到 11 把鑰匙(原始 + key expansion 後的 10 把)。我們將使用 16KB，16MB，16GB 的檔案對系統進行測試，透過 CPU 運算此 function 得到軟體的運算時間以及使用 u50 運算得到硬體的運算時間，比較軟體和硬體上的加速狀況。最後將加密完畢的檔案和標準答案比較，驗證加密是否正確。

2. How we design our system

i. Kernel

按上述 block 我們將系統分 Kexp(key expansion)、S-Box、ShiftRows、Mixcolumns、Cipher。

1. S-Box

我們原先想說 S-Box 僅僅只是查表法，期望他只是組合邏輯的輸出，8bits 輸入與 8bits 輸出，程式碼如下。

```
//#include "sbox.h"
#include "ap_int.h"

unsigned char sbox(unsigned char si){
    const unsigned char S_Box[] =
    {
        //0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F
        0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76, //0
        0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0, //1
        0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15, //2
        0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75, //3
        0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84, //4
        0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf, //5
        0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8, //6
        0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2, //7
        0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73, //8
        0xe6, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb, //9
        0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79, //A
        0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0x5d, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08, //B
        0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a, //C
        0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0xd8, 0x9e, //D
        0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0x9d, //E
        0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16, //F
    };

    // #pragma HLS ARRAY PARTITION variable=S_Box type=complete
    #pragma HLS INTERFACE ap_none port=si
    #pragma HLS INTERFACE ap_ctrl_none port=return

    return S_Box[si];
}
```

但實際合成結果如下。

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LU*	URAM
sbox				-	1	10.000		-	2	no	1	0	2	14	0

合成器會自動使用 BRAM 並使用到 FF，這結果並不是我們想要的，我們希望至少 sbox 在 kexp 裡面可以是組合邏輯，一個週期可以產生一把 roundkey。後來我們查到相關資料，多下這行 pragma。

```
#pragma HLS ARRAY_PARTITION variable=S_Box
type=complete
```

程式碼如下。

```
#include "ap_int.h"

unsigned char sbbox(unsigned char si){
    const unsigned char S_Box[] =
    {
        //0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F
        0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76, //0
        0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0, //1
        0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15, //2
        0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75, //3
        0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84, //4
        0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf, //5
        0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8, //6
        0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2, //7
        0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73, //8
        0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb, //9
        0xe6, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xea, 0x79, //A
        0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0x5d, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08, //B
        0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a, //C
        0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e, //D
        0xe1, 0xf8, 0x98, 0x11, 0x00, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf, //E
        0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16 //F
    };
};

#pragma HLS ARRAY_PARTITION variable=S_Box type=complete
#pragma HLS INTERFACE ap_none port=si
#pragma HLS INTERFACE ap_ctrl_none port=return

return S_Box[si];
}
```

合成結果如下。

Performance & Resource Estimates													
▼ Performance & Resource Estimates ⓘ													
Modules & Loops													
Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSF	FF
sbox					0	0.0			1	no	0	0	0
												1362	0
													URAM

這樣就可以如同我們所預期的變成組合邏輯，僅用 LUT 即可完成。

2. Mixcolumn

Mixcolumn 運算我們希望可以用組合邏輯實現，所以在 for loop 中 pragma 使用 unroll、interface 的部分使用 ap_ctrl_none，以及輸出入的 array partition 使用 complete，程式碼如下。

```
#define xtime(x) ((x < 1) ^ (((x >> 7) & 0x01) * 0x1b))
void MixColumns(unsigned char state[4][4], unsigned char state_mixc[4][4])
{
    #pragma HLS ARRAY_PARTITION variable=state dim=0 complete
    #pragma HLS ARRAY_PARTITION variable=state_mixc dim=0 complete

    #pragma HLS INTERFACE ap_ctrl_none port=return

    unsigned char t[4];
    unsigned char Tmp[4], Tm1[4], Tm1_tmp[4];
    unsigned char Tm2[4], Tm2_tmp[4];
    unsigned char Tm3[4], Tm3_tmp[4];
    unsigned char Tm4[4], Tm4_tmp[4];

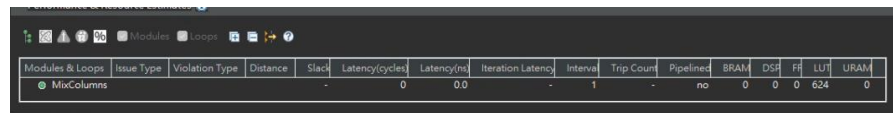
    int i, j;
    for( i = 0; i < 4; i++)
    {
        #pragma HLS UNROLL
        t[i] = state[0][i];
        Tmp[i] = state[0][i] ^ state[1][i] ^ state[2][i] ^ state[3][i];
        Tm1[i] = state[0][i] ^ state[1][i];
        Tm1_tmp[i] = xtime(Tm1[i]);
        state_mixc[0][i] = state[0][i] ^ Tm1_tmp[i] ^ Tmp[i];

        Tm2[i] = state[1][i] ^ state[2][i];
        Tm2_tmp[i] = xtime(Tm2[i]);
        state_mixc[1][i] = state[1][i] ^ Tm2_tmp[i] ^ Tmp[i];

        Tm3[i] = state[2][i] ^ state[3][i];
        Tm3_tmp[i] = xtime(Tm3[i]);
        state_mixc[2][i] = state[2][i] ^ Tm3_tmp[i] ^ Tmp[i];

        Tm4[i] = state[3][i] ^ t[i];
        Tm4_tmp[i] = xtime(Tm4[i]);
        state_mixc[3][i] = state[3][i] ^ Tm4_tmp[i] ^ Tmp[i];
    }
}
```

合成結果如下。



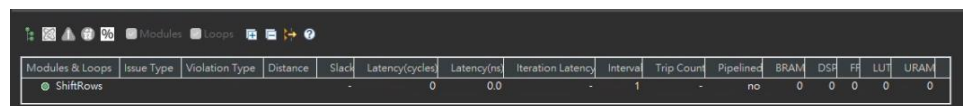
Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSF	FF	LU*	URAM
MixColumns					0	0.0		1		no	0	0	0	624	0

3. ShiftRow

ShiftRow 僅僅只是將矩陣從新排列，理論上不該有甚麼 LUT 的消耗，甚至不需要 FF，因此 interface 的部分一樣採用 ap_ctrl_none，array partition 依舊使用 complete。

```
void ShiftRows(unsigned char state[4][4], unsigned char state_sftr[4][4]){  
    #pragma HLS ARRAY_PARTITION variable=state dim=0 complete  
    #pragma HLS ARRAY_PARTITION variable=state_sftr dim=0 complete  
    #pragma HLS INTERFACE ap_ctrl_none port=return  
  
    //unsigned char tempByte1,tempByte2,tempByte3,tempByte4;  
    state_sftr[0][0] = state[0][0];  
    state_sftr[0][1] = state[0][1];  
    state_sftr[0][2] = state[0][2];  
    state_sftr[0][3] = state[0][3];  
  
    // 2nd row left Circular Shift 1 byte  
    //tempByte1 = state[1][0];  
    state_sftr[1][0] = state[1][1];  
    state_sftr[1][1] = state[1][2];  
    state_sftr[1][2] = state[1][3];  
    state_sftr[1][3] = state[1][0];  
  
    // 3th row left Circular Shift 2 byte  
    //tempByte2 = state[2][0];  
    state_sftr[2][0] = state[2][2];  
    state_sftr[2][2] = state[2][0];  
  
    //tempByte3 = state[2][1];  
    state_sftr[2][1] = state[2][3];  
    state_sftr[2][3] = state[2][1];  
  
    // 4th row left Circular Shift 3 byte  
    //tempByte4 = state[3][0];  
    state_sftr[3][0] = state[3][3];  
    state_sftr[3][3] = state[3][2];  
    state_sftr[3][2] = state[3][1];  
    state_sftr[3][1] = state[3][0];  
}
```

合成結果如同我們所預期的，並沒有用到任何資源，如下。



Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSF	FF	LU*	URAM
ShiftRows					0	0.0		1		no	0	0	0	0	0

4. Kexp

在 Kexp 我們希望可以每一個 cycle 產生一把 roundkey(128bits)，因此至少要執行 11 個 cycle，總計輸出 176 個 unsigned char 的資料，於是我們將輸出的 176 個 unsigned char 切成 11 等份每一份都是 16 個 unsigned char，輸出入的 pragma 如下。

```

void kexp(unsigned char key[16], unsigned char roundkey[176]){

    #pragma HLS ARRAY_PARTITION variable=key complete
    #pragma HLS ARRAY_PARTITION variable=roundkey cyclic factor=16

    unsigned char tempByte[16];
    unsigned char tempByte_tmp[4];
    unsigned char tempByte_sbox[4];
    unsigned char rk_tmp0[4], rk_tmp1[4], rk_tmp2[4], rk_tmp3[4];
    #pragma HLS INTERFACE ap_ctrl_hs port=return

    unsigned char roundkey_tmp[176];

    #pragma HLS ARRAY_PARTITION variable=roundkey_tmp cyclic factor=16

```

接著是執行 11 次的 for loop，程式碼如下。

```

for ( i = 0; i < 11; i++)
{

    // #pragma HLS UNROLL

    if(i==0){
        for ( j = 0; j < 16; j++){
            #pragma HLS UNROLL
            roundkey[j] = key[j];
            roundkey_tmp[j] = key[j];
        }
    }
    else{
        for ( j = 0; j < 16; j++){
            #pragma HLS UNROLL
            tempByte[j] = roundkey_tmp[(i - 1) * 16 + j];
            // tempByte[j] = roundkey_tmp[(i - 1) * 16 + j];
        }
        // RotWord function
        // a0 = tempByte[0];
        tempByte_tmp[0] = tempByte[13];
        tempByte_tmp[1] = tempByte[14];
        tempByte_tmp[2] = tempByte[15];
        tempByte_tmp[3] = tempByte[12];

        // SubWord function XOR Rcon[i]

        tempByte_sbox[0] = sbox(tempByte_tmp[0]) ^ Rcon[i];
        tempByte_sbox[1] = sbox(tempByte_tmp[1]);
        tempByte_sbox[2] = sbox(tempByte_tmp[2]);
        tempByte_sbox[3] = sbox(tempByte_tmp[3]);
    }
}

```



```
tempByte_sbox[0] = sbox(tempByte_tmp[0]) ^ Rcon[1];
tempByte_sbox[1] = sbox(tempByte_tmp[1]);
tempByte_sbox[2] = sbox(tempByte_tmp[2]);
tempByte_sbox[3] = sbox(tempByte_tmp[3]);

rk_tmp0[0] = tempByte[0] ^ tempByte_sbox[0];
rk_tmp0[1] = tempByte[1] ^ tempByte_sbox[1];
rk_tmp0[2] = tempByte[2] ^ tempByte_sbox[2];
rk_tmp0[3] = tempByte[3] ^ tempByte_sbox[3];

rk_tmp1[0] = tempByte[4] ^ rk_tmp0[0];
rk_tmp1[1] = tempByte[5] ^ rk_tmp0[1];
rk_tmp1[2] = tempByte[6] ^ rk_tmp0[2];
rk_tmp1[3] = tempByte[7] ^ rk_tmp0[3];

rk_tmp2[0] = tempByte[8] ^ rk_tmp1[0];
rk_tmp2[1] = tempByte[9] ^ rk_tmp1[1];
rk_tmp2[2] = tempByte[10] ^ rk_tmp1[2];
rk_tmp2[3] = tempByte[11] ^ rk_tmp1[3];

rk_tmp3[0] = tempByte[12] ^ rk_tmp2[0];
rk_tmp3[1] = tempByte[13] ^ rk_tmp2[1];
rk_tmp3[2] = tempByte[14] ^ rk_tmp2[2];
rk_tmp3[3] = tempByte[15] ^ rk_tmp2[3];

roundkey_tmp[i*16 + 0] = rk_tmp0[0];
roundkey_tmp[i*16 + 1] = rk_tmp0[1];
roundkey_tmp[i*16 + 2] = rk_tmp0[2];
roundkey_tmp[i*16 + 3] = rk_tmp0[3];

roundkey_tmp[i*16 + 4] = rk_tmp1[0];
roundkey_tmp[i*16 + 5] = rk_tmp1[1];
roundkey_tmp[i*16 + 6] = rk_tmp1[2];
roundkey_tmp[i*16 + 7] = rk_tmp1[3];
```

```

roundkey_tmp[i*16 + 8] = rk_tmp2[0];
roundkey_tmp[i*16 + 9] = rk_tmp2[1];
roundkey_tmp[i*16 + 10] = rk_tmp2[2];
roundkey_tmp[i*16 + 11] = rk_tmp2[3];

roundkey_tmp[i*16 + 12] = rk_tmp3[0];
roundkey_tmp[i*16 + 13] = rk_tmp3[1];
roundkey_tmp[i*16 + 14] = rk_tmp3[2];
roundkey_tmp[i*16 + 15] = rk_tmp3[3];

roundkey[i*16 + 0] = roundkey_tmp[i*16 + 0];
roundkey[i*16 + 1] = roundkey_tmp[i*16 + 1];
roundkey[i*16 + 2] = roundkey_tmp[i*16 + 2];
roundkey[i*16 + 3] = roundkey_tmp[i*16 + 3];

roundkey[i*16 + 4] = roundkey_tmp[i*16 + 4];
roundkey[i*16 + 5] = roundkey_tmp[i*16 + 5];
roundkey[i*16 + 6] = roundkey_tmp[i*16 + 6];
roundkey[i*16 + 7] = roundkey_tmp[i*16 + 7];

roundkey[i*16 + 8] = roundkey_tmp[i*16 + 8];
roundkey[i*16 + 9] = roundkey_tmp[i*16 + 9];
roundkey[i*16 + 10] = roundkey_tmp[i*16 + 10];
roundkey[i*16 + 11] = roundkey_tmp[i*16 + 11];

roundkey[i*16 + 12] = roundkey_tmp[i*16 + 12];
roundkey[i*16 + 13] = roundkey_tmp[i*16 + 13];
roundkey[i*16 + 14] = roundkey_tmp[i*16 + 14];
roundkey[i*16 + 15] = roundkey_tmp[i*16 + 15];
}
}

```

最終合成結果也如同我們所預期的 11 個 cycle 就可以完成運算，如下。

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trips Count	Pipelined	BRAM	DSP	FF	LU*	URAM
keyp				-	13	130.000	-	14	-	no	0	0	396	6658	0
VITIS_LOOP_30_1				-	11	110.000	2	1	11	yes	-	-	-	-	-

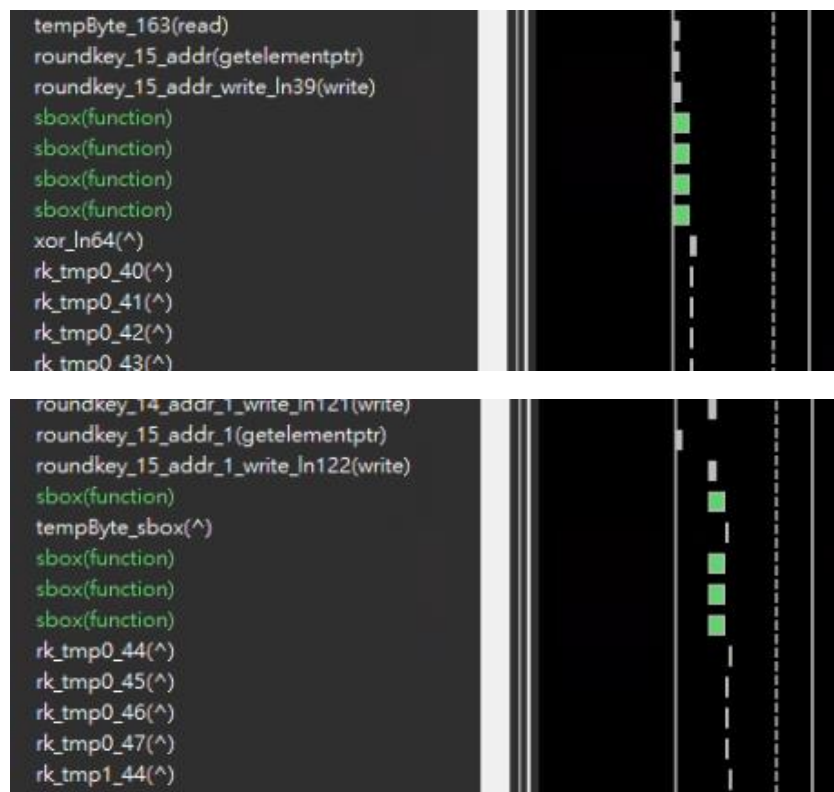
但是這樣寫 co-sim 是出來的數值是錯誤的，我們打開波形查看，僅有第一把 roundkey 是正常的，後面的 cycle 輸出全為 XX(unknown)，波形如下。



我們嘗試在最外層的 for loop 使用 UNROLL，結果僅使用 5 個 cycle 即可完成運算，但 sbox 使用量多了一倍，按上面程式碼，我們僅 call 了 4 個 sbox，合成結果如下。

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSR	#F	LU*	URAM
• kexp				-	5	50.000	-	-	6	-	0	0	470	14719	0
• sbox				-	0	0.0	-	0	-	no	0	0	0	1362	0
• sbox				-	0	0.0	-	0	-	no	0	0	0	1362	0
• sbox				-	0	0.0	-	0	-	no	0	0	0	1362	0
• sbox				-	0	0.0	-	0	-	no	0	0	0	1362	0
• sbox				-	0	0.0	-	0	-	no	0	0	0	1362	0
• sbox				-	0	0.0	-	0	-	no	0	0	0	1362	0
• sbox				-	0	0.0	-	0	-	no	0	0	0	1362	0
• sbox				-	0	0.0	-	0	-	no	0	0	0	1362	0
• sbox				-	0	0.0	-	0	-	no	0	0	0	1362	0

接著跑 co-sim 跟看 timeline，co-sim 的模擬結果是對的，而且查看 timeline 發現在一個 cycle 內，它運行了 2 次的 for loop 內運算，因此需要多一倍的 sbox，我們也藉此發現，若一運算無法在一個周期內完成，合成器會自動幫我們加上 register 隔開，並用一些周期來完成，部分 timeline(同一個週期)如下。



5. Cipher(加密)

最後我們需要將資料和上面 kexp 產生的 roundkey 進行 11 回合的 add roundkey。我們想將其應用在大型的資料上面，並希望每一個 cycle 能輸出 1 筆 128bits 資料(16 個 bytes)，於是我們用 stream 的方式讀取跟寫出，深度定義為 1024 筆，也就是我們一個 kernel 可以處理 1024 個 16B，也就是 16KB，top module 如下。


```

void Cipher_top(unsigned char roundkey_in[176],hls::vector<uint8_t, NUM_WORDS>* plain, hls::vector<uint8_t, NUM_WORDS>* cipher){
    unsigned char roundkey[176];
    #pragma HLS ARRAY_PARTITION variable=roundkey_in cyclic factor=16
    #pragma HLS ARRAY_PARTITION variable=roundkey cyclic factor=16
    load_key(roundkey_in,roundkey);

    #pragma HLS INTERFACE m_axi port=plain bundle=gmem1 depth=1024
    #pragma HLS INTERFACE m_axi port=cipher bundle=gmem0 depth=1024
    static hls::stream<hls::vector<uint8_t, NUM_WORDS> > plain_stream("input_plain_stream");
    static hls::stream<hls::vector<uint8_t, NUM_WORDS> > cipher_stream("output_cipher_stream");

    #pragma HLS dataflow
    load_input(plain,plain_stream);
    Cipher(roundkey,plain_stream,cipher_stream);
    store_result(cipher,cipher_stream);
}

```

(定義 NUM_WORDS = 16、DATASIZE=1024)

讀取與寫入程式碼如下。

```

}

static void load_input(hls::vector<uint8_t, NUM_WORDS>* in,
                      hls::stream<hls::vector<uint8_t, NUM_WORDS> >& inStream) {
    mem_rd:
    for (int i = 0; i < DATA_SIZE; i++) {
        #pragma HLS LOOP_TRIPCOUNT min = c_size max = c_size
        inStream << in[i];
    }
}

static void store_result(hls::vector<uint8_t, NUM_WORDS>* out,
                        hls::stream<hls::vector<uint8_t, NUM_WORDS> >& out_stream) {
    mem_wf:
    for (int i = 0; i < DATA_SIZE; i++) {
        #pragma HLS LOOP_TRIPCOUNT min = c_size max = c_size
        out[i] = out_stream.read();
    }
}

```

最後 Cipher 如同上述，需要進行 11 次的 roundkey，也就是要跑 11 次的迴圈，再以 stream 的方式讀進與寫出，程式碼如下。

```

static void Cipher(unsigned char roundkey[176], hls::stream<hls::vector<uint8_t, NUM_WORDS> >& plain_stream, hls::stream<hls::vector<uint8_t, NUM_WORDS> >& cipher_stream)
{
    for (int r=0;r<DATA_SIZE;r++)
    {
        #pragma HLS LOOP_TRIPCOUNT min = c_size max = c_size

        hls::vector<uint8_t, NUM_WORDS> plain;
        hls::vector<uint8_t, NUM_WORDS> cipher;
        plain = plain_stream.read();

        #pragma HLS ARRAY_PARTITION variable=roundkey dim=0 complete

        int round;
        int i,j;
        unsigned char state[4][4];
        #pragma HLS ARRAY_PARTITION variable=state dim=0 complete
        unsigned char state_sftr[4][4];
        #pragma HLS ARRAY_PARTITION variable=state_sftr dim=0 complete
        unsigned char state_mixc[4][4];
        #pragma HLS ARRAY_PARTITION variable=state_mixc dim=0 complete
        /**
        * [b0 b1 ... b15] -> [b0 b4 b8 b12
        *                   b1 b5 b9 b13
        *                   b2 b6 b10 b14
        *                   b3 b7 b11 b15]
        */

        for ( i = 0; i < 4; i++){
            #pragma HLS UNROLL
            for ( j = 0; j < 4; j++){
                #pragma HLS UNROLL
                state[j][i] = plain[i * 4 + j];
            }
        }
    }
}

```

```

// round 0 : add round key
for ( i = 0; i < 4; i++){
    #pragma HLS UNROLL
    for ( j = 0; j < 4; j++){
        #pragma HLS UNROLL
        state[j][i] ^= roundkey[(i * Nb + j)];
    }
}

// Round 1 ~ Nc-1
for (round = 1; round < Nr; round++){

    for ( i = 0; i < 4; i++){
        #pragma HLS UNROLL
        for ( j = 0; j < 4; j++){
            #pragma HLS UNROLL
            state[j][i] = sbox(state[j][i]);
        }
    }

    ShiftRows(state, state_sftr);
    MixColumns(state_sftr, state_mxc);

    for ( i = 0; i < 4; i++){
        #pragma HLS UNROLL
        for ( j = 0; j < 4; j++){
            #pragma HLS UNROLL
            state[j][i] = state_mxc[j][i] ^ roundkey[(i * Nb + j) + (round * Nb * 4)];
        }
    }
}

```

```

// Round Nc, no MixColumns()
for ( i = 0; i < 4; i++){
    #pragma HLS UNROLL
    for ( j = 0; j < 4; j++){
        #pragma HLS UNROLL
        state[j][i] = sbox(state[j][i]);
    }
}

ShiftRows(state, state_sftr);
for ( i = 0; i < 4; i++){
    #pragma HLS UNROLL
    for ( j = 0; j < 4; j++){
        #pragma HLS UNROLL
        state[j][i] = state_sftr[j][i] ^ roundkey[(i * Nb + j) + (round * Nb * 4)];
    }
}

/**
 * [c0 c4 c8 c12
 * c1 c5 c9 c13 --> [c0 c1 c2 ... c15]
 * c2 c6 c10 c14
 * c3 c7 c11 c15]
 */
for ( i = 0; i < 4; i++) {
    #pragma HLS UNROLL
    for ( j = 0; j < 4; j++){
        #pragma HLS UNROLL
        cipher[i * 4 + j] = state[j][i];
    }
}
cipher_stream.write(cipher);
}

```

合成結果出乎我們意料，它會自動將 1~Nc-1 round 的 for loop 進行 unroll，就如同 kexp，每一個 cycle 都做兩次 for loop 的運算，然而這邊需要做 dataflow 的 pipeline 需要至少 5 層的 flip flop 隔開，因此需要的硬體資源會是 x2x5 倍，按上述預計的總 cycle 會是(1024+6-1)=1029 個 cycle，合成結果如下。

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency/cycles	Latency/ns	Iteration Latency	Interval	Trip Count	Pipelined	BRAMs	DSPs	FFs	LU ^T	URAMs
⊟ Cipher_top				-	1047	1.047E4	-	1036	-	dataflow	0	0	31428	249594	0
⊟ load_key				-	0	0.0	-	0	-	no	0	0	3	29	0
⊟ load_input				-	13	130.000	-	13	-	no	0	0	2831	3241	0
⊟ Cipher				-	1035	1.035E4	-	1035	-	no	0	0	216	305	0
⊟ store_result				-	1030	1.030E4	-	1030	-	no	0	0	8470	228106	0
				-	1035	1.035E4	-	1035	-	no	0	0	216	339	0

最後執行 cosim 也與合成的預計時間差不多，如下。


```

size_t array_size_bytes_k = key_length * sizeof(unsigned char);
size_t array_size_bytes_rk = roundkey_length * sizeof(unsigned char);
OCL_CHECK(err,
            cl::Buffer buffer_key(context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY, array_size_bytes_k, key.data(), &err));
OCL_CHECK(err,
            cl::Buffer buffer_roundkey1(context, CL_MEM_USE_HOST_PTR | CL_MEM_WRITE_ONLY, array_size_bytes_rk, roundkey1.data(), &err));
OCL_CHECK(err,
            cl::Buffer buffer_roundkey2(context, CL_MEM_USE_HOST_PTR | CL_MEM_WRITE_ONLY, array_size_bytes_rk, roundkey2.data(), &err));
OCL_CHECK(err,
            cl::Buffer buffer_roundkey3(context, CL_MEM_USE_HOST_PTR | CL_MEM_WRITE_ONLY, array_size_bytes_rk, roundkey3.data(), &err));
OCL_CHECK(err,
            cl::Buffer buffer_roundkey4(context, CL_MEM_USE_HOST_PTR | CL_MEM_WRITE_ONLY, array_size_bytes_rk, roundkey4.data(), &err));
std::cout << " |-----|\n"
            << " | Kernel           | Wall-Clock Time (ns) |\n"
            << " |-----|\n";

OCL_CHECK(err, kexp_kernel1 = cl::Kernel(program, "kexp", &err));
OCL_CHECK(err, kexp_kernel2 = cl::Kernel(program, "kexp", &err));
OCL_CHECK(err, kexp_kernel3 = cl::Kernel(program, "kexp", &err));
OCL_CHECK(err, kexp_kernel4 = cl::Kernel(program, "kexp", &err));

OCL_CHECK(err, err = kexp_kernel1.setArg(0, buffer_key));
OCL_CHECK(err, err = kexp_kernel1.setArg(1, buffer_roundkey1));
OCL_CHECK(err, err = kexp_kernel2.setArg(0, buffer_key));
OCL_CHECK(err, err = kexp_kernel2.setArg(1, buffer_roundkey2));
OCL_CHECK(err, err = kexp_kernel3.setArg(0, buffer_key));
OCL_CHECK(err, err = kexp_kernel3.setArg(1, buffer_roundkey3));
OCL_CHECK(err, err = kexp_kernel4.setArg(0, buffer_key));
OCL_CHECK(err, err = kexp_kernel4.setArg(1, buffer_roundkey4));

OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_key}, 0));

```

```

cl::Event event1,event2,event3,event4;

OCL_CHECK(err, err = q.enqueueTask(kexp_kernel1, nullptr, &event1));
OCL_CHECK(err, err = q.enqueueTask(kexp_kernel2, nullptr, &event2));
OCL_CHECK(err, err = q.enqueueTask(kexp_kernel3, nullptr, &event3));
OCL_CHECK(err, err = q.enqueueTask(kexp_kernel4, nullptr, &event4));

OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_roundkey1}, CL_MIGRATE_MEM_OBJECT_HOST));
OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_roundkey2}, CL_MIGRATE_MEM_OBJECT_HOST));
OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_roundkey3}, CL_MIGRATE_MEM_OBJECT_HOST));
OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_roundkey4}, CL_MIGRATE_MEM_OBJECT_HOST));

q.finish();

verify(roundkey1);
verify(roundkey2);
verify(roundkey3);
verify(roundkey4);

return EXIT_SUCCESS;

```

先定義所需要的 buffer size，將寫好的 kernel 寫入 u50 並定義 kernel function arguments，最後將 data 從 host 端搬進 u50 的 4 個 buffer 中，執行運算後再將 output 從 4 個 buffer 搬回 host 端，最後再驗證 4 個 roundkey。

3. System(two kernel, 16KB)

由於我們使用 16KB 一開始一樣讀取 key 進行 key expansion 得出 roundkey。接著使用此 roundkey 進行 encrypt。我們每執行一次加密可以處理 16KB 大小的資料，因此 16KB 只需執行一次，不需要計算 iteration 的數量。因此總系統類似於上述只使用 key expansion kernel 的程式，只需使用 2 個 kernel(kexp 和 cipher 兩個 kernel) 按順序執行

```

OCL_CHECK(err,
            cl::Buffer buffer_key(context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY, array_size_bytes_k, key.data(), &err));
OCL_CHECK(err,
            cl::Buffer buffer_roundkey(context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_WRITE, array_size_bytes_rk, roundkey.data(), &err));

OCL_CHECK(err,
            cl::Buffer buffer_plain(context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY, array_size_bytes_plain, plain.data(), &err));
OCL_CHECK(err,
            cl::Buffer buffer_cipher(context, CL_MEM_USE_HOST_PTR | CL_MEM_WRITE_ONLY, array_size_bytes_cipher, cipher.data(), &err));

OCL_CHECK(err, kexp_kernel = cl::Kernel(program, "kexp", &err));
OCL_CHECK(err, cipher_kernel = cl::Kernel(program, "Cipher_top", &err));

OCL_CHECK(err, err = kexp_kernel.setArg(0, buffer_key));
OCL_CHECK(err, err = kexp_kernel.setArg(1, buffer_roundkey));

OCL_CHECK(err, err = cipher_kernel.setArg(0, buffer_roundkey));
OCL_CHECK(err, err = cipher_kernel.setArg(1, buffer_plain));
OCL_CHECK(err, err = cipher_kernel.setArg(2, buffer_cipher));

OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_key}, 0 /* 0 means from host */));
OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_plain}, 0 /* 0 means from host */));
cl::Event event_rk, event_cipher;

vector<cl::Event> event;
OCL_CHECK(err, err = q.enqueueTask(kexp_kernel, nullptr, &event_rk));
event.push_back(event_rk);

OCL_CHECK(err, err = q.enqueueTask(cipher_kernel, &event, &event_cipher));

OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_cipher}, CL_MIGRATE_MEM_OBJECT_HOST));

q.finish();

```

4. System(two kernel, 16MB、16G)

由於此時必須重新使用 buffer 和執行 encrypt kernel，因此必須計算總共跑幾次 iterations。因為每次執行都能加密 16KB 的檔案，16MB 和 16GB 總共需要 1024、1024*1024 次 iterations。

```

size_t ARRAY_SIZE = DATA_SIZE*NUM_WORDS*(1024LL*1024);
size_t elements_per_iteration = DATA_SIZE*NUM_WORDS;
size_t num_iterations = ARRAY_SIZE / elements_per_iteration;

```

接著執行 key expansion 的部分，但僅需執行一次即可，因此 key expansion 的程式與上述內容相同。

```

cl::Kernel kexp_kernel;
cl::Event kexp_events;

cl::Buffer buffer_key;
cl::Buffer buffer_roundkey1;

std::cout << "Creating Kexp Buffers..." << std::endl;
OCL_CHECK(err,
            buffer_key = cl::Buffer(context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY, array_size_bytes_k, key.data(), &err));
OCL_CHECK(err,
            buffer_roundkey1 = cl::Buffer(context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_WRITE, array_size_bytes_rk, roundkey1.data(), &err));

cl::Event kexp_transfer;
vector<cl::Event> wait_kexp;

std::cout << "Copying Kexp data (Host to Device)..." << std::endl;
OCL_CHECK(err, kexp_kernel = cl::Kernel(program, "kexp", &err));
OCL_CHECK(err, err = kexp_kernel.setArg(0, buffer_key));
OCL_CHECK(err, err = kexp_kernel.setArg(1, buffer_roundkey1));

OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_key}, 0 /* 0 means from host */, &wait_kexp, &kexp_transfer));
wait_kexp.push_back(kexp_transfer);
set_callback(kexp_transfer, "write_event_queue");

auto kernel_begin = std::chrono::high_resolution_clock::now();
auto kexp_kernel_begin = std::chrono::high_resolution_clock::now();

printf("Enqueueing kernel.\n");
std::cout << "Running kexp_kernel" << std::endl;
OCL_CHECK(err, err = q.enqueueTask(kexp_kernel, &wait_kexp, &kexp_events));
wait_kexp.push_back(kexp_events);
set_callback(kexp_events, "kexp_kernel_queue");
auto kexp_kernel_end = std::chrono::high_resolution_clock::now();

```

接著，我們必須不斷讀取新的 data 進入 encrypt 中的 buffer 並使用 roundkey 去進行加密，並重新指定 kernel

的 argument。另外，在此處遇到了 Host mem bad use 的問題，經過測試發現可能是因為多次使用同一個 buffer 和 host arguments，因此將此處使用的 buffer 和 host 中的 array 數量提高。

```
cl::Kernel cipher_kernel;
cl::Buffer buffer_roundkey;
vector<cl::Buffer> buffer_plain(20);
vector<cl::Buffer> buffer_cipher(20);
cl::Event cipher_events;

int i=0;
int z=0;
int j;
for (size_t iteration_idx = 0; iteration_idx < num_iterations; iteration_idx++){
    j=z%20;
    printf("Read Plain...\n");
    fread(plain[j].data(),1,NUM_WORDS*DATA_SIZE,fp1);
    std::cout << "Creating Cipher Buffers..." << std::endl;
    //    buffer_roundkey = buffer_roundkey1;
    OCL_CHECK(err,
               buffer_plain[j] = cl::Buffer(context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY, array_size_bytes_plain, plain[j].data(), &err));
    OCL_CHECK(err,
               buffer_cipher[j] = cl::Buffer(context, CL_MEM_USE_HOST_PTR | CL_MEM_WRITE_ONLY, array_size_bytes_cipher, cipher[j].data(), &err));

    cl::Event write_event;
    std::cout << "Copying Cipher data (Host to Device)..." << std::endl;
    OCL_CHECK(err, cipher_kernel = cl::Kernel(program, "Cipher_top", &err));

    OCL_CHECK(err, err = cipher_kernel.setArg(0, buffer_roundkey1));
    OCL_CHECK(err, err = cipher_kernel.setArg(1, buffer_plain[j]));
    OCL_CHECK(err, err = cipher_kernel.setArg(2, buffer_cipher[j]));

    OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_plain[j]}, 0 /* 0 means from host */, nullptr, &write_event));
    write_event.wait();
    set_callback(write_event, "write_event_queue");
}
```

執行 kernel 後，必須確定 buffer 的內容已經搬進 host，清除目前使用中的 buffer 並將搬出的 data 寫入檔案後，再進行下個 iteration。

```
auto cipher_kernel_begin = std::chrono::high_resolution_clock::now();

vector<cl::Event> eventList;
printf("Enqueueing kernel...\n");
std::cout << "Running cipher_kernel: " << std::endl;
OCL_CHECK(err, err = q.enqueueTask(cipher_kernel, nullptr, &cipher_events));
eventList.push_back(cipher_events);
OCL_CHECK(err, err = cl::Event::waitForEvents(eventList));
set_callback(cipher_events, "cipher_kernel_queue");

auto cipher_kernel_end = std::chrono::high_resolution_clock::now();
auto kernel_end = std::chrono::high_resolution_clock::now();
cipher_kernel_duration += cipher_kernel_end - cipher_kernel_begin;
kernel_duration = kernel_end - kernel_begin;

vector<cl::Event> flagWait;
cl::Event read_events;
std::cout << "Getting Cipher Results (Device to Host)..." << std::endl;
OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_cipher[j]}, CL_MIGRATE_MEM_OBJECT_HOST, &eventList, &read_events));
flagWait.push_back(read_events);
OCL_CHECK(err, err = cl::Event::waitForEvents(flagWait));
set_callback(read_events, "total_queue");

printf("Releasing Cipher Buffer...\n");
buffer_roundkey = nullptr;
buffer_plain[j] = nullptr;
buffer_cipher[j] = nullptr;
cipher_events = nullptr;

printf("Write Cipher...\n");
fwrite(cipher.data(),1,NUM_WORDS*DATA_SIZE,fp2);
```

5. System(two kernel, 16MB、16G, multiple compute unit)
修改 number of compute units 數量去進行計算
與第 4 部分內容接近，但是此時我們需要使用更多的 buffer 以及 kernel 執行 encrypt，因此我們使用等同於 compute unit 數量的 buffer 和 kernel。

```
vector<cl::Kernel> cipher_kernel(num_CU);
vector<cl::Buffer> buffer_roundkey(num_CU);
vector<cl::Buffer> buffer_plain(num_CU);
vector<cl::Buffer> buffer_cipher(num_CU);
vector<cl::Event> cipher_events(num_CU);
```

接下來的概念與上述相同，把檔案搬入每個 buffer 中，把這些 buffer 指定給各個 kernel。

```
for (size_t iteration_idx = 0; iteration_idx < num_iterations; iteration_idx++){
    int i=0;
    printf("Read Plain...\n");
    for(int i=0;i<num_CU;i++){
        fread(plain[i].data(),1,NUM_WORDS*DATA_SIZE,fp1);
    }

    std::cout << "Creating Cipher Buffers..." << std::endl;
    for(int i=0;i<num_CU;i++){
        buffer_roundkey[i] = buffer_roundkey1;
        OCL_CHECK(err,
            buffer_roundkey[i] = cl::Buffer(context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_WRITE, array_size_bytes_plain, roundkey1.data(), &err));
        OCL_CHECK(err,
            buffer_plain[i] = cl::Buffer(context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY, array_size_bytes_plain, plain[i].data(), &err));
        OCL_CHECK(err,
            buffer_cipher[i] = cl::Buffer(context, CL_MEM_USE_HOST_PTR | CL_MEM_WRITE_ONLY, array_size_bytes_cipher, cipher[i].data(), &err));
    }

    vector<cl::Event> write_event(num_CU);
    std::cout << "Copying Cipher data (Host to Device)..." << std::endl;
    for(int i=0;i<num_CU;i++){
        OCL_CHECK(err, cipher_kernel[i] = cl::Kernel(program, "Cipher_top", &err));

        OCL_CHECK(err, err = cipher_kernel[i].setArg(0, buffer_roundkey[i]));
        OCL_CHECK(err, err = cipher_kernel[i].setArg(1, buffer_plain[i]));
        OCL_CHECK(err, err = cipher_kernel[i].setArg(2, buffer_cipher[i]));

        OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_plain[i]}, 0 /* 0 means from host */, nullptr, &write_event[i]));
    }
    OCL_CHECK(err, err = cl::Event::waitForEvents(write_event));
    set_callback(write_event[i], "write_event_queue");
}
```

執行每個 kernel 後，把每個 buffer 的內容搬回 Host 中，並在確定每個 buffer 都已經搬完之後，在 Host 端將這些資料寫進另外一個檔案，確認上述步驟都執行完畢後，才開始另外一個迴圈。

```
vector<cl::Event> flagWait;
vector<cl::Event> read_events(num_CU);
std::cout << "Getting Cipher Results (Device to Host)..." << std::endl;
for(int i=0;i<num_CU;i++){
    OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_cipher[i]}, CL_MIGRATE_MEM_OBJECT_HOST, &eventList, &read_events[i]));
    flagWait.push_back(read_events[i]);
}
OCL_CHECK(err, err = cl::Event::waitForEvents(flagWait));
set_callback(read_events[i], "total_queue");

printf("Write Cipher...\n");
for(int i=0;i<num_CU;i++){
    fwrite(cipher[i].data(),1,NUM_WORDS*DATA_SIZE,fp2);
}

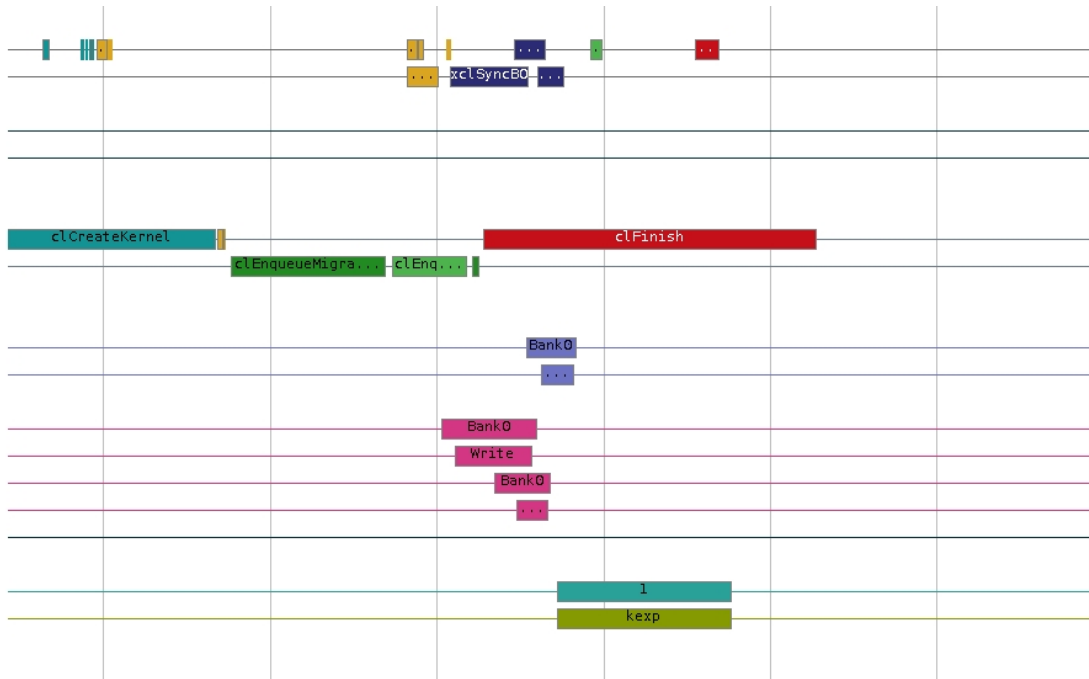
printf("Releasing Cipher Buffer...\n");
for(int i=0;i<num_CU;i++){
    buffer_roundkey[i] = nullptr;
    buffer_plain[i] = nullptr;
    buffer_cipher[i] = nullptr;
    cipher_events[i] = nullptr;
}

printf("Finish...\n");
OCL_CHECK(err, err = q.finish());
```

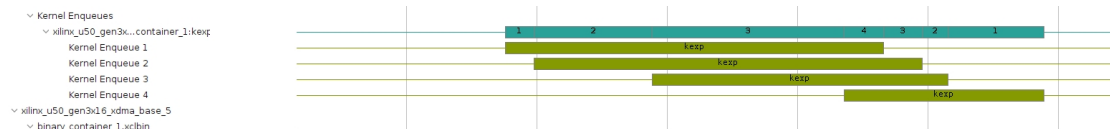
3. Result

i. Key expansion (Single compute unit)

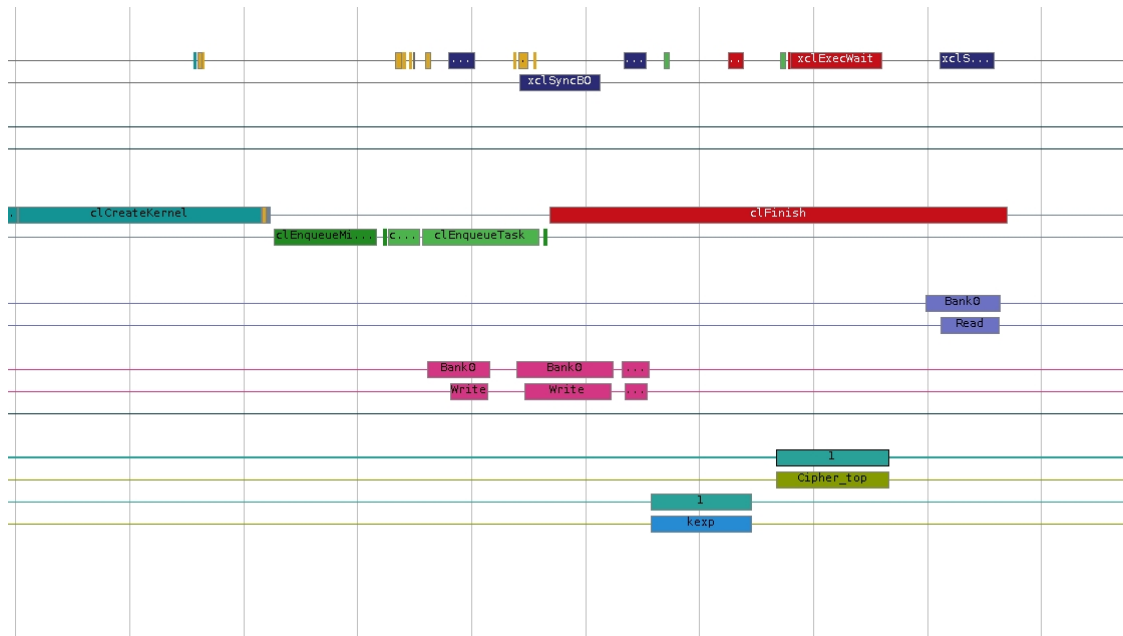
在 key expansion 中，我們並未計算總共的運算時間，僅觀察 run summary 中的 timeline 確定是否有正常運作。



ii. Key expansion (multiple compute unit)



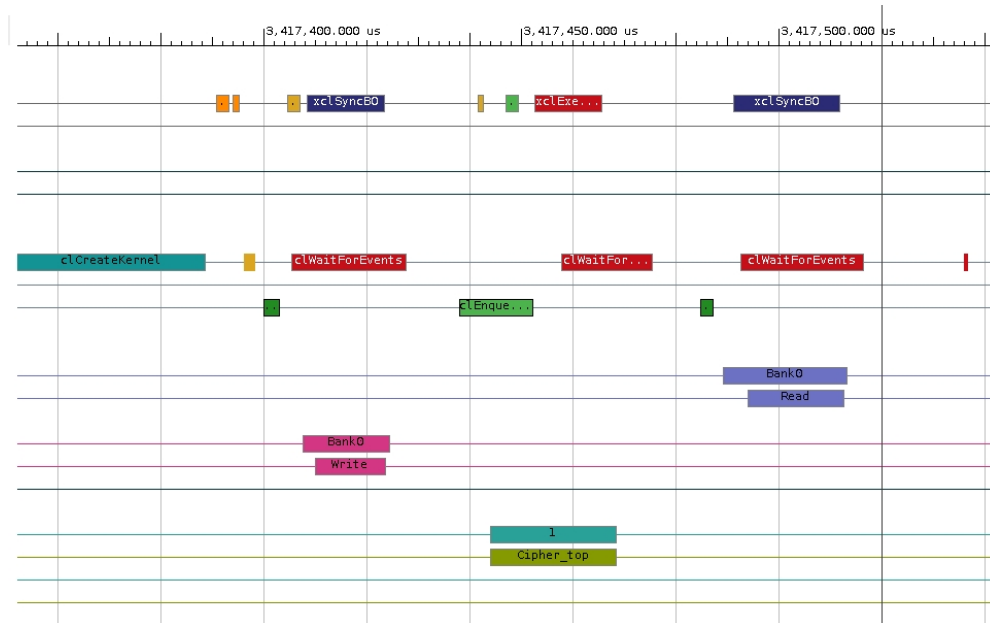
iii. System(two kernel , 16KB)



```
|total time(s):| 0.00126428 |
|key expansion time(s):| 2.9377e-05 |
|cipher time(s):| 0.000103843 |
```

iv. System(two kernel , 16MB 、16G)

16MB :



```
|total time(s):|    0.408606 |
|key expansion time(s):|  4.3837e-05 |
|cipher time(s):|    0.0497739 |
```

16GB :

```
|total time(s):|    492.178 |
|key expansion time(s):|  4.8716e-05 |
|cipher time(s):|    59.6024 |
```

v. System(two kernel , 16G , multiple compute unit)

由於在使用多個 kernel 時會出現 malloc(): unsorted double linked list corrupted 的問題，導致在使用 vitis IDE 執行 Host Code 時無法產生 run summary，因此只能透過 std::chrono 計算運算時間。

1. Compute Unit=2



```
|total time(s):|      387.745 |  
|key expansion time(s):|  6.5237e-05 |  
|cipher time(s):|      37.1302 |
```

2. Compute Unit=4

```
|total time(s):|      360.801 |  
|key expansion time(s):|  4.6742e-05 |  
|cipher time(s):|      28.5813 |
```

3. Compute Unit=8

```
|total time(s):|      353.445 |  
|key expansion time(s):|  4.4729e-05 |  
|cipher time(s):|      24.6704 |
```